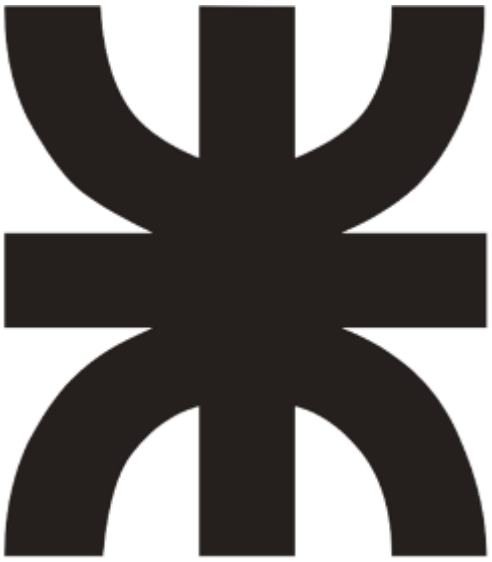
Universidad Tecnológica Nacional

Facultad Regional Rosario

Tecnicatura Universitaria en Programación



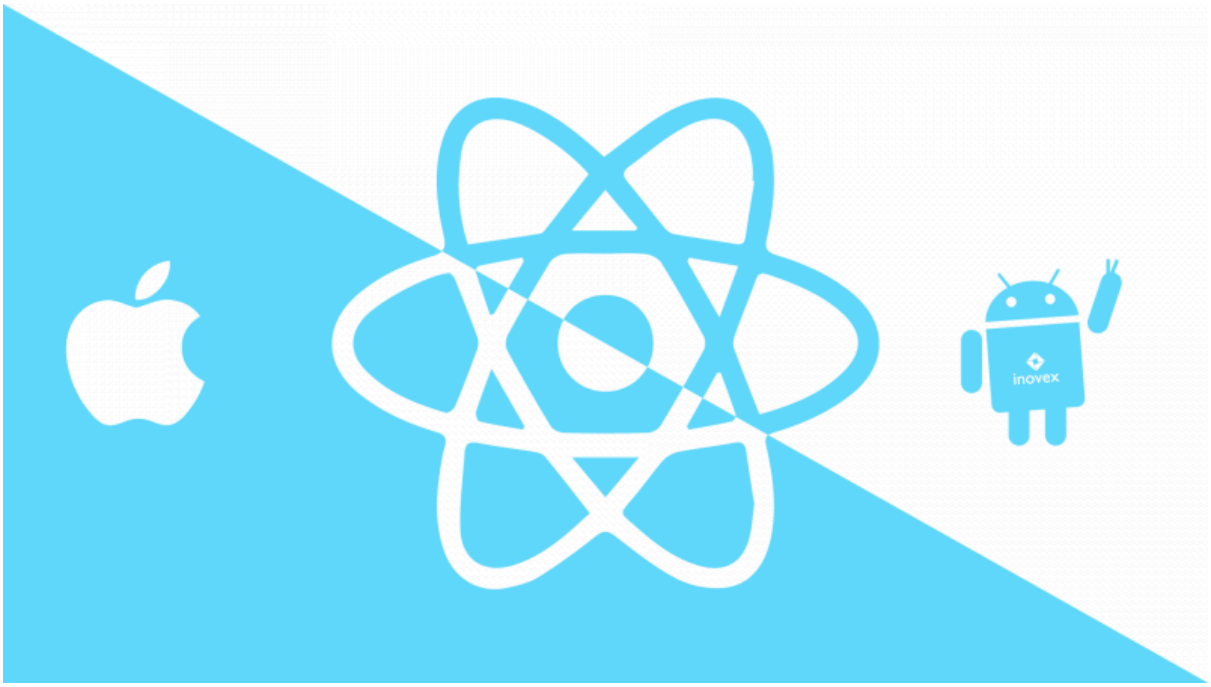
Programación III

Unidad 5.1

Introducción a React Native

Introducción a React Native

¿Qué es React Native?



React Native, junto a React.js (que hemos visto que es una librería de javascript que nos permite construir interfaces de usuario modernas) nos va a dar la posibilidad de construir **aplicaciones nativas para IOS y Android**.

De la parte de React, es necesario notar que React en sí es **agnóstico a su plataforma**, esto quiere decir, su utilización para crear componentes y manejar el estado de un árbol virtual de elementos no necesariamente es obligatorio de usar en la web. Por eso es que para poder correrlo en nuestros navegadores le agregamos la librería *react-dom* que nos hace el trabajo por nosotros.

Entonces, React Native es justamente una **alternativa** a *react-dom*, la cual nos da una colección de componentes especiales que luego serán compilados en elementos nativos de las interfaces de usuario móviles (sea de IOS o Android). A su vez, React Native nos permite acceder a las API 's de los teléfonos (tales por ejemplo como la cámara) y utilizarlas dentro de nuestra aplicación.

Pero, ¿Cómo lo hace?

Veamos a continuación un código de un componente simple, escrito en React + React Native:

```
const App = () => {
        return (
                <View>
                        <Text> ¡Buenos días!</Text>
                </View>
                );
};
```

Este código luego va a ser compilado a una **verdadera aplicación nativa**. Esto varía según la plataforma en la que estemos compilando el código.

Observemos que además tenemos dos componentes que no conocíamos del desarrollo web: *View* y *Text*. Estos son dos de esos componentes “especiales” que sí van a ser compilados a lenguaje nativo por React Native. Notar que la lógica de Javascript **no va a ser compilada a la aplicación nativa**, sino solo el JSX.

Veamos esta tabla comparativa de ejemplo:

Web browser (*react-dom*)	**Native component** (Android)	**Native componente** (IOS)	**React Native JSX**
	android.View	UIView	<View>
<input>	EditText	UITextField	<TextInput>

La lógica Javascript (por fuera del JSX) se encuentra en un hilo de Javascript manejado por la aplicación nativa (la app construida con React Native). Básicamente, cuando hacemos *build* de nuestra app de React Native, se va a correr un subproceso en nuestra app que se encargará de la traducción de esa lógica Javascript para que el teléfono pueda realizar las funciones y especificaciones deseadas.

Instalando React Native

Para poder instalar React Native, nos dirigiremos a su página oficial, a *Environment Setup*. Allí tenemos dos opciones: la instalación mediante expo o la instalación con react native directamente. Ambos sirven para levantar entornos de desarrollo y crear aplicaciones de React Native. Entonces, ¿Cuál es la diferencia?

Expo CLI (por *command-line interface*), o sencillamente expo, es un servicio *third-party* que es gratuito (aunque tiene opciones *premium*) que nos permitirá realizar un desarrollo controlado de nuestra aplicación de celulares. Esto quiere decir que nos proveerá de facilidades a lo largo del desarrollo que generarán menos fricción a la hora de escribir y publicar nuestras *apps*.

Otra ventaja es que podemos salir del ecosistema expo cuando nosotros queramos.¿Cuál es la ventaja de salir de expo e ir a React Native puro? En general, ya que el desarrollo allí es más directo (aunque más complejo), las integraciones con el código nativo de celular (*swift* en *IOS* o *Java/kotlin* en *Android*) son mucho más simples que con expo.

Durante estas clases de introducción utilizaremos expo debido a los puntos anteriores.

Para poder comenzar la instalación, debemos tener **Node.js y npm instalado**.

Luego corremos el siguiente comando en nuestra terminal preferida:

npx create-expo-app --template blank book-champions-mobile

Ya que no utilizaremos typescript y tampoco queremos pantallas predefinidas, eligiremos la opción *blank*.

Analizando el proyecto creado

Analicemos las carpetas que se nos generaron automáticamente:

- *assets*: esto va a ser importante más adelante, ya que esta carpeta contendrá las imágenes de nuestra app.
- *node_modules*: contiene todos los paquetes externos y sus dependencias, que van a conformar nuestra aplicación.
- *package.json*: contiene las especificaciones de los paquetes principales y las dependencias. Remarquemos que expo además de ser un CLI es un paquete de node.
- *app.json*: es un archivo de configuración que expo va a utilizar cuando pre visualicemos o construyamos nuestra app.
- *app.js*: ¡El primer archivo de código de nuestra aplicación! Es un componente normal de React que utiliza los componentes especiales de React Native, junto a un estilizado un poco diferente a lo que estamos acostumbrados.

Correr nuestra primera app

Para poder correr nuestra app rápidamente, podemos descargar (sea en IOS o en Android) la app de **expo go**.

Mientras descargamos, abrimos una terminal en el directorio de nuestro proyecto y corremos *npm start*. Eso nos dará un código QR dentro de la terminal y nos abrirá una nueva pestaña en el navegador con información similar. Si escaneamos con la aplicación expo go ese código QR, nos compilará el código y veremos nuestra app en el celular.

Podemos cambiar el texto de la app, guardar los cambios y veremos que se actualiza automáticamente en nuestro celular. Así vemos lo fácil que es desarrollar y testear estas aplicaciones con expo.

Setear el entorno de desarrollo local

Para poder correr nuestra aplicación *mobile* en nuestra terminal y poder previsualizarla, en el caso de que tengamos Windows o Linux utilizaremos *Android Studio*. En el caso que poseamos una Macbook, utilizaremos tanto android studio como xCode. Esto es debido a que es imposible la previsualización de aplicaciones de *IOS* en terminales que no posean el sistema operativo MacOS.

Link de descarga de Android Studio

Una vez descargado e instalado, lo inicializamos y seleccionamos la opción *virtual device manager* (dentro de *more actions*). Allí, elegiremos el dispositivo que deseamos virtualizar (es importante que tenga el icono de *Play Store*, ya que vamos a necesitar descargar la aplicación de *expo*). Le damos adelante, seleccionamos la API de Android más moderna (la descargamos si hace falta) y luego le damos al botón de *play* para inicializarla.

Luego de que se haya inicializado el dispositivo virtual, si nos dirigimos a nuestra terminal veremos que apretando la letra tecla **a** nos empieza a generar la aplicación en ese mismo dispositivo. Esto lo usaremos como alternativa para poder probar nuestra app en distintos entornos.

Opcional: correr nuestra app en la web

Para aquellos que quizás la aplicación de Android Studio les sea demasiado pesada en términos de costos de procesamiento, pueden optar por correr la app en el puerto 8081 de la web y luego mediante las DevTools de Chrome (F12 o click derecho → inspeccionar) elegir la opción de visualización *mobile* para efectuar los testeos correspondientes.

Para poder correr la app en web, debemos apretar la tecla **w** luego de **npm start**. Nos solicitará la siguiente instalación:

npx expo install react-dom react-native-web @expo/metro-runtime

Luego de realizarla, repetimos el proceso.

Nota: desde la cátedra, sin embargo, recomendamos sobre esta opción el hecho de escanear el QR con el celular propio e ir testeando en nuestros propios celulares.

Conceptos básicos

Una vez que tengamos el ecosistema de desarrollo andando, podemos empezar a construir nuestra primera aplicación de prueba, donde veremos los primeros conceptos básicos de React Native.

```
1   import { StatusBar } from 'expo-status-bar';
2   import { StyleSheet, Text, View } from 'react-native';
3
4   const App = () => {
5     return (
6       <View style={styles.container}>
7         <Text>Open up App.js to start working on your app!</Text>
8         <StatusBar style="auto" />
9       </View>
10    );
11  }
12
13  const styles = StyleSheet.create({
14    container: {
15      flex: 1,
16      backgroundColor: '#fff',
17      alignItems: 'center',
18      justifyContent: 'center',
19    },
20  });
21
22  export default App;
23
```

En el archivo *App.js* default que nos crea, podemos ver que poseemos varios *imports* hacia librerías y un componente funcional sencillo llamado App. También se ve un *import* llamado *Stylesheet* que veremos más adelante.

Observemos que, dentro de estos *imports* de React Native se encuentra *Text* y *View*. Estos son dos de los componentes especiales de Native más importantes que vamos a tener a nuestra disposición. Recordemos que **no podemos utilizar elementos JSX directamente que si utilizamos en React** (como por ejemplo <*div*> o <*p*>) ya que no estamos en un navegador, sino en un entorno nativo de celular.

Podemos ver que componentes nos aporta React Native aquí.

Las interfaces de usuario de React Native las vamos a construir combinando estos *core components* junto a nuestro estilizado de esos componentes. Pero, ¿Cómo estilizamos si no existe el *CSS* en entornos nativos de celular?

Para poder estilizar los componentes tenemos dos opciones principales: utilizar *inline-styles* dentro de los componentes o **utilizar el *Stylesheet object*** que nos provee React Native. Es decir, todo el código css va a ser también escrito en **javascript**.

Core components

Eliminemos el componente *Status Bar* y borremos las etiquetas de *Text* que rodean a "*Open up App.js to start working on your app!"*, luego escribamos "*Hello World!* sin ninguna etiqueta que lo contenga:

```
import { StatusBar } from "expo-status-bar";
import { StyleSheet, Text, View } from "react-native";

const App = () => {
  return <View style={styles.container}>Hello World!</View>;
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: "#fff",
    alignItems: "center",
    justifyContent: "center",
  },
});

export default App;
```

Si realizamos eso, veremos un error en la consola:

```
Error: Text strings must be rendered within a <Text> component.

This error is located at:
    in RCTView (created by View)
    in View (created by App)
    in App (created by ExpoRoot)
    in ExpoRoot
    in RCTView (created by View)
    in View (created by AppContainer)
    in RCTView (created by View)
    in View (created by AppContainer)
    in AppContainer
    in main(RootComponent)
```

Y también un error en el emulador o en el dispositivo.

Esto nos significa que si bien *View* es el “equivalente” al *div* en web, no funciona exactamente igual. En este caso, *View* lo utilizaremos para poder organizar los contenedores o “cajas” que poseen cada una de las pantallas del celular. En caso de que quisiéramos, imprimir un texto, si o si necesitamos que el mismo esté encuadrado en la etiqueta *Text*.

Entonces, el componente *View* nos servirá para poder encapsular a los otros componentes y además proveer estilizado a los bloques de nuestras pantallas.

Estilizado en React Native

Como dijimos antes, no existe el CSS como lo conocemos en React Native. Esto significa que algunas funcionalidades de CSS no las vamos a poder utilizar y algunos nombres conocidos de las mismas van a ser cambiados.

Para estilizar vamos a usar la *prop style* que no se encuentra en todos los *core components*, pero sí en algunos de ellos, como por ejemplo, *View* o *Text*.

style es una *prop* que espera un objeto de javascript. Ese objeto va a tener las propiedades de estilizado que deseemos pasarle al componente, como *margin* o *backgroundColor*. Los valores de las propiedades van a depender según la propiedad. Por ejemplo, *margin* espera un número, y ese número luego va a ser traducido a pixeles.

```
export default function App() {
  return (
    <View style={styles.container}>
      <Text
        style={{ margin: 16, borderWidth: 2, borderColor: "red", padding: 16 }}
      >
        Hello World!
      </Text>
      <StatusBar style="auto" />
    </View>
  );
}
```

Si bien este tipo de estilizado está permitido, sabemos de cuando aprendimos React que el *inline-styling* se considera en muchos casos una **mala práctica** debido a lo tedioso que resulta estilizar por cada componente (no poseemos clases que nos permita replicar el estilizado) y además de la pérdida de trazabilidad que nos genera.

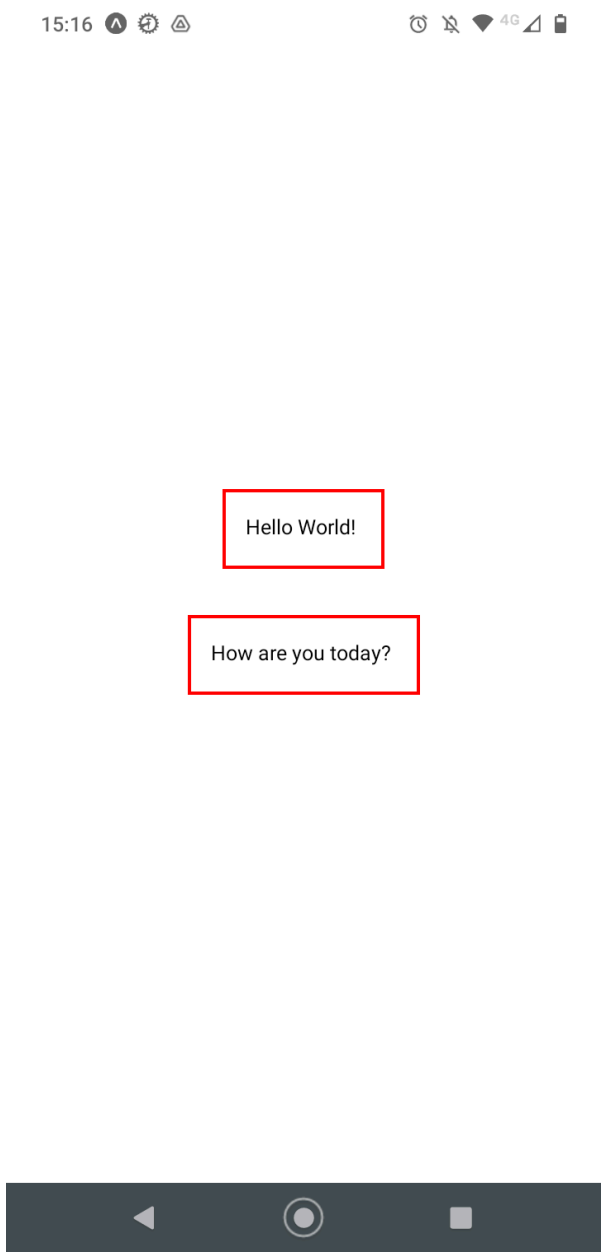
Para poder utilizar algo similar a las clases de CSS que utilizamos en web, usaremos la funcionalidad Stylesheet que nos proveé React Native:

```
17   const styles = StyleSheet.create({
18     container: {
19       flex: 1,
20       backgroundColor: "#fff",
21       alignItems: "center",
22       justifyContent: "center",
23     },
24   });
25
```

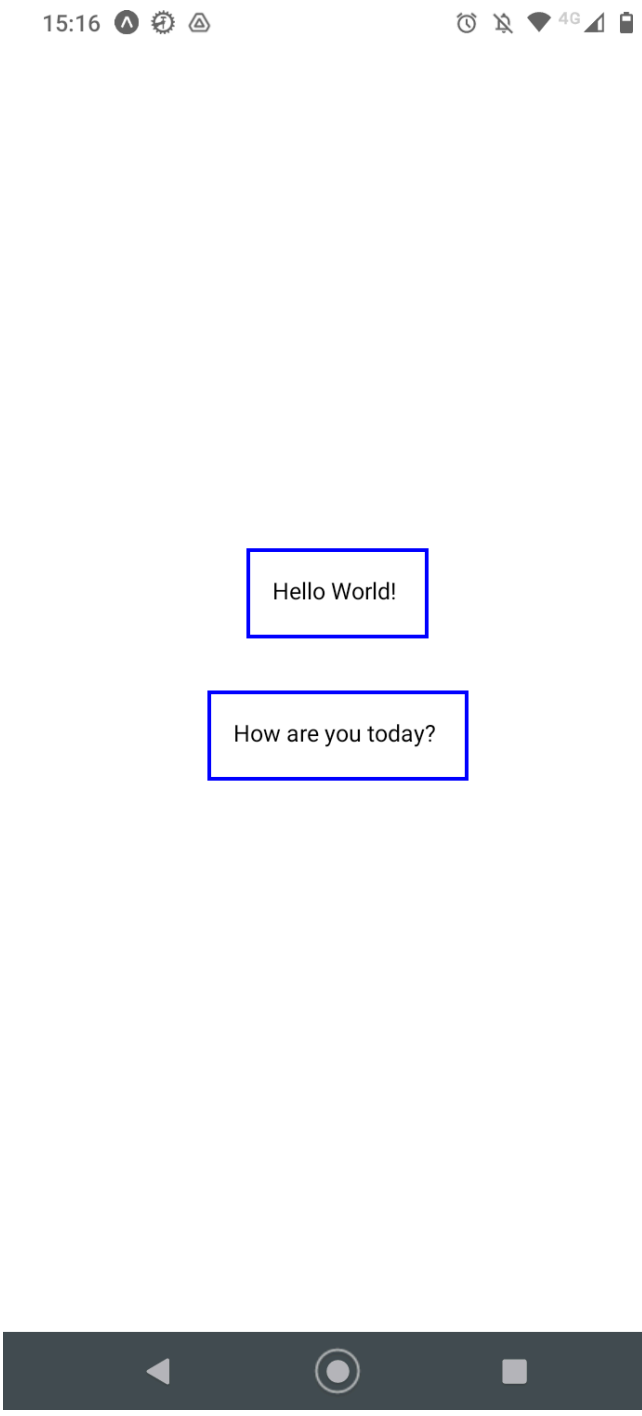
Ahi observamos que en el primer *View* se aplica la propiedad *container*. Para nosotros agregar nuestras propiedades *custom*, basta con tal de agregar una propiedad nueva con un objeto que posea el estilizado:

```
export default function App() {
  return (
    <View style={styles.container}>
      <Text style={styles.welcomeText}>Hello World!</Text>
      <Text style={styles.welcomeText}>How are you today?</Text>

      <StatusBar style="auto" />
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: "#fff",
    alignItems: "center",
    justifyContent: "center",
  },
  welcomeText: {
    margin: 16,
    borderWidth: 2,
    borderColor: "red",
    padding: 16,
  },
});
```

Luego colocamos el estilizado en la *prop* style de cada uno de los Textos (agregamos uno para visualizar que usando la misma propiedad obtenemos el estilizado):



También podemos probar cambiando el color del estilizado para ver que afecta a ambos textos:


15:16
Hello World!
How are you today?

Comenzando nuestra App: *layout* principal

Primero antes de empezar a codear nuestra aplicación compañera de la biblioteca, vamos a limpiar lo que tenemos actualmente, borrando todo el objeto de *styles* (pero dejando el método *createStyleSheet*) y en el *return* dejaremos un sólo *View* sin estilizado.

Nuestra aplicación donde podremos, en primera medida, cargar las lecturas que ya leímos en el 2025, tendrá dentro de ese *View* principal dos *Views* más, en donde en el primero pondremos un *TextInput* (importado de React Native) junto con un *Button* para el ingreso de los nombres de los libros, y en el segundo por ahora solo un *Text* que diga “Lista de lecturas…”:

Como observamos, no vendría mal agregarle un poco de padding a nuestra *View* principal:

```
17  const styles = StyleSheet.create({
18    appContainer: {
19      padding: 50,
20    }
21  });
```

Usando *flexbox* para nuestro *Layout*

Todos los componentes *View* de React Native vienen con el *display:flexbox* incorporado dentro de su estilizado interno. Vamos a aprovechar esta facilidad para poder estilizar un poco nuestra aplicación, y acomodar los bloques que vemos en la pantalla:

```jsx
import { StyleSheet, Text, View, TextInput, Button } from 'react-native';

export default function App() {
  return (
    <View style={styles.appContainer}>
      <View style={styles.inputContainer}>
        <TextInput style={styles.textInput} placeholder='Ingrese el nombre del libro' />
        <View style={styles.buttonInput}>
          <Button title='Agregar lectura' />
        </View>
      </View>
      <View>
        <Text>Lista de lecturas...</Text>
      </View>
    </View>
  );
}
```

```jsx
const styles = StyleSheet.create({
  appContainer: {
    padding: 50,
  },
  inputContainer: {
    justifyContent: "center"
  },
  textInput: {
    width: '100%',
    marginRight: 8,
    marginVertical: 15,
    padding: 8,
    borderWidth: 1,
    borderColor: "#CCCCCC",
  },
  buttonInput: {
    width: "50%",
    alignSelf: "flex-end"
  }
});
```

Nota: el componente *Button* no posee una *prop* de style, por ende estilizaremos una *View* que encierre al botón.

Manejando eventos dentro de la app

Actualmente nuestra aplicación no posee lógica que permita interactuar con el usuario, de manera que este pueda agregar las lecturas que va ingresando en el *input*. En definitiva, deberíamos escuchar cada carácter que el usuario ingresa en el input y permitirle, al tocar el botón, agregar la nueva lectura a la lista de lecturas.

Para eso, tengamos en cuenta que las facilidades que nos aportaba React en cuanto al manejo de interfaces de usuario, no está solo en el apartado UI sino también en el apartado UX de React Native. Funcionalidades nativas de React como *hooks, props* y manejo de *state* están presentes en RN y nos simplificarán el desarrollo en gran medida.

Crearemos dos funciones llamadas: *handleReadingInput* y *handleAddReading*. Luego, en *TextInput* agregamos la *prop onChangeText*, que funciona similar al *onChange* de React y le pasaremos la función *handleReadingInput* .

```
5    const handleReadingInput = (enteredText) => {
6      console.log(enteredText);
7    };
```

El parámetro *enteredText* es un parámetro automático que nos envía React Native que toma el valor del texto actual que se encuentra en el input (similar a como en React utilizabamos el *event.target.value*).

Veremos que a medida que ingresamos el texto en el *TextInput*, en VSC, en la consola, nos va imprimiendo el texto actualizado de lo que el usuario ingresa.

Ahora, para utilizar ese texto en la función *handleAddReading* lo vamos a agregar al estado del componente. El componente *Button* no posee un evento *onClick* pero si un evento *onPress* que nos permitirá relacionar a la función con el botón:

```
1  import { useState } from 'react';
2  import { StyleSheet, Text, View, TextInput, Button } from 'react-native';
3
4  export default function App() {
5    const [text, setText] = useState("")
6
7    const handleReadingInput = (enteredText) => {
8      setText(enteredText)
9    };
10
11   const handleAddReading = () => {
12     console.log(text);
13   }
14
15   return (
16     <View style={styles.appContainer}>
17       <View style={styles.inputContainer}>
18         <TextInput
19           style={styles.textInput}
20           placeholder='Ingrese el nombre del libro'
21           onChangeText={handleReadingInput}
22         />
23         <View style={styles.buttonInput}>
24           <Button title='Agregar lectura' onPress={handleAddReading} />
25         </View>
26       </View>
27       <View>
28         <Text>Lista de lecturas...</Text>
29       </View>
30     </View>
31   );
32 }
33
```

Observamos que cuando presionamos el botón de agregar lectura, se imprime en consola el texto escrito.

Listas en React Native

Para poder armar la lista de recetas, setearemos un estado que contenga inicialmente un arreglo donde guardaremos cada una de esas lecturas. Luego, en la función *handleAddReading* tendremos dos formas de actualizar el estado:

```
5    const [text, setText] = useState("");
6    const [readings, setReadings] = useState([]);
7
8    const handleReadingInput = (enteredText) => {
9      setText(enteredText)
10   };
11
12   const handleAddReading = () => {
13     setReadings([...readings, text]);
14   }
```

O sino, de una mejor manera (ya que aquí estamos actualizando el estado de React en base a un estado anterior):

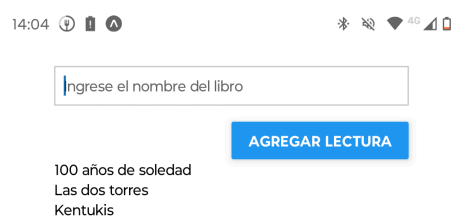
```
5    const [text, setText] = useState("");
6    const [readings, setReadings] = useState([]);
7
8    const handleReadingInput = (enteredText) => {
9      setText(enteredText)
10   };
11
12   const handleAddReading = () => {
13     setReadings((prevReadings) => [...prevReadings, text]);
14   }
```

Ambas van a funcionar, pero la segunda forma nos permite asegurarnos que vamos a utilizar correctamente el estado anterior y evitar problemas de sincronización de estados.

¿Cómo podemos mostrar esta lista, una vez que ya la tenemos guardada en el estado?

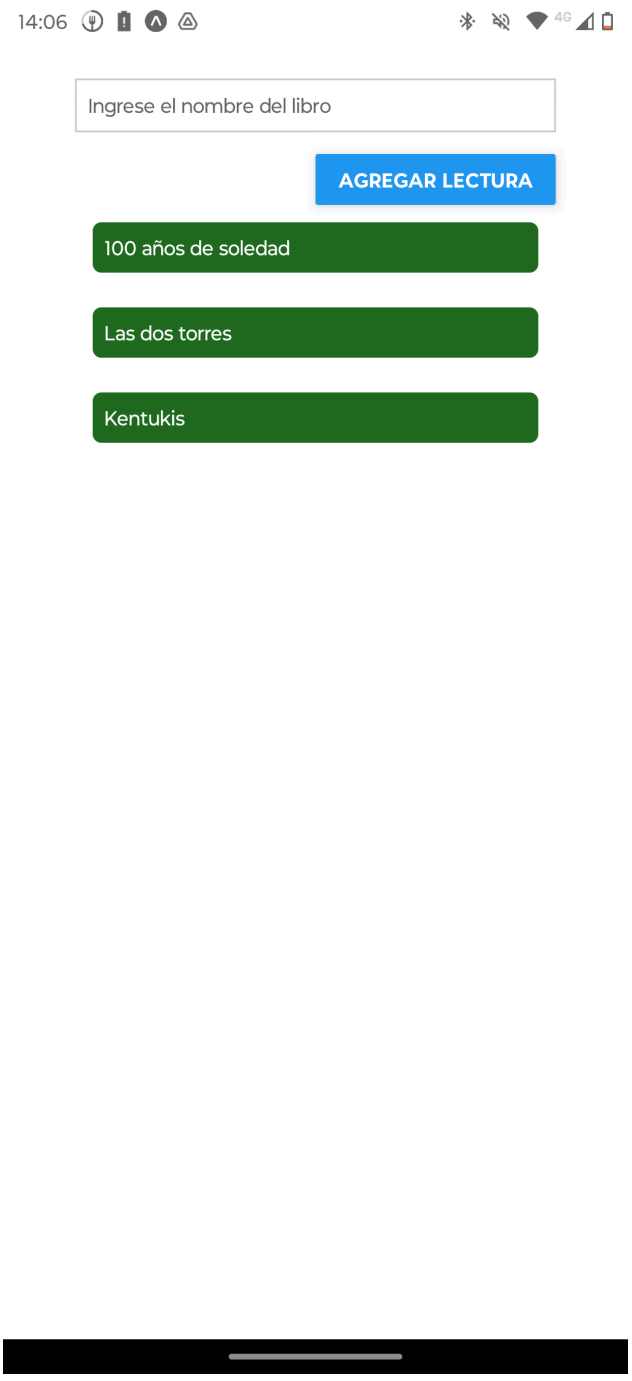
Al ser una lista de *strings*, podremos mapear de manera sencilla cada uno de los valores de *readings* en un componente *Text* (utilizaremos el mismo *string* como *key*, si bien esto claramente no va a ser único, nos permite salir del apuro para esta demo de aplicación)

```
16    const mappedReadings = readings.map((reading) =>
17      <Text key={reading}>{reading}</Text>
18    )
19
20    return (
21      <View style={styles.appContainer}>
22        <View style={styles.inputContainer}>
23          <TextInput
24            style={styles.textInput}
25            placeholder='Ingrese el nombre del libro'
26            onChangeText={handleReadingInput}
27          />
28          <View style={styles.buttonInput}>
29            <Button title='Agregar lectura' onPress={handleAddReading} />
30          </View>
31        </View>
32        <View>
33          {mappedReadings}
34        </View>
35      </View>
```



Podemos agregarle un poco de estilizado a esos componentes *Text*:

```
58    readingItem: {
59      margin: 12,
60      padding: 8,
61      borderRadius: 6,
62      backgroundColor: "#206B21FF",
63      color: "#FFF"
64    }
```



Componente *ScrollView*

Si comenzamos a agregar una variedad de lecturas a nuestra aplicación, veremos que eventualmente la lista sobrepasa el margen inferior y no nos permite desplazarnos hacia abajo para ver su contenido. En web, ligado al concepto de *overflow*, cuando esto sucedía

nos aparecían automáticamente unas barras de desplazamiento para *scrollear* y ver el contenido completo.

En el caso de React Native, para que una *View* sea desplazable, debemos explicitarlo en el código. Como es un caso de uso frecuente, existe un componente (llamado *ScrollView*) que realiza el trabajo por nosotros. Probemos en agregarlo, previamente cambiando la *key* a algo tipo *reading* + *index*:

```
32      <ScrollView>
33        {mappedReadings}
34      </ScrollView>
```

Vemos que la lista ahora si es desplazable.

ScrollView vs *FlatList*

Entonces con *ScrollView* podemos hacer contenido dentro de la aplicación desplazable. En este caso lo hemos usado para una lista pero también se podría usar por sobre cualquier vista que creamos que generaría un *overflow* en el dispositivo (pensemos quizás, un artículo de un portal digital)

Ahora, imaginemos que el usuario es una biblioteca reconocida de Rosario, y no tiene solo 15 libros o lecturas, sino que sabe poseer más de 5000 libros y variaciones de los mismos. La lista entonces nos quedaría de más de 30 elementos.

El problema con *ScrollView* es que renderiza todos los elementos dentro suyo **sin tener en cuenta si se ven o no, sin importar tampoco su cantidad**. Esto podría generar problemas de *performance* en el *booteo* de la app, en donde tendremos que renderizar listas de +1000 ítems por ejemplo, pero el usuario solo visualiza 10 máximo al mismo tiempo.

Entonces, *ScrollView* nos va a servir para elementos con un fin definido. En el caso de listas dinámicas donde no controlamos en teoría la cantidad de elementos posibles a renderizar, una mejor solución sería utilizar el componente de React Native ***FlatList***.

FlatList nos permite mantener el desplazamiento en la lista, y solo nos va a renderizar los ítems de la lista que se vean en un momento dado, renderizando los nuevos a medida que el usuario se desplaza por la lista. Veamos cómo podemos aplicarlo:

```
36      <FlatList data={readings}
37        renderItem={renderReadingItem}
38        keyExtractor={(item) => item.id}
39      />
```

```
16    const renderReadingItem = (itemData) => (
17      <Text style={styles.readingItem}>{itemData.item.text}</Text>
18    )
```

```
12  const handleAddReading = () => {
13    setReadings((prevReadings) => [...prevReadings, { id: Math.random().toString(), text }]);
14  }
15
```

1. Creamos la etiqueta cerrada *FlatList*.
2. *FlatList* posee una *prop* llamada ***data***, donde le pasaremos la lista de elementos que deseamos renderizar.
3. ***renderItem*** es una *prop* que espera una función, donde por cada item va a renderizar el jsx que nosotros deseemos. Notamos que el ítem no es solo el texto original, sino que convierte al item en un objeto (con metadata agregada) y guarda el valor original en la propiedad *item*.
4. A su vez y como ejemplo, *FlatList* reconoce la propiedad *key* de cada uno de los elementos de la lista automáticamente (es decir, si el objeto original tiene una propiedad *key*, *FlatList* la detecta sin explicitarlo). En el caso de que no sea el caso,como si en vez de la propiedad *key* tenemos una propiedad *id*, podemos utilizar la *prop* ***keyExtractor***, que como indica su nombre, por cada item va a extraer la key de la propiedad que le indiquemos.
5. Por último, modificamos el *handleAddReading* para que estos cambios estén correctamente reflejados.

Ejercicio de clase

Hacer composición en el componente *App* y dividir al mismo en *ReadingInput y ReadingList* de manera que el código resulte más prolijo.

Agregando interacciones a elementos

Si quisiéramos agregar funcionalidades a elementos de nuestra aplicación cuando el usuario los presione, no podremos sencillamente hacerlo como en web y agregar un *onClick* (en este caso, un *onPress*) al componente *View*, ya que no es una *prop* aceptada por el mismo.

Lo que sí podemos hacer es utilizar componentes que nos provee React Native específicos que solucionan este problema. Antiguamente se utilizaba una diversa librería (*Touchable, TouchableOpacity*) pero actualmente (y a futuro) se está utilizando principalmente el componente *Pressable*.

Entonces, si quisiéramos eliminar las lecturas agregadas tocando los bloques de colores, primero deberíamos envolver cada receta en ese *Pressable*.

```
const ReadingList = ({
    readings = [],
    onDelete = () => { }
}) => {
    const renderReadingItem = (itemData) => (
        <Pressable onPress={onDelete}>
            <Text style={styles.readingItem}>{itemData.item.text}</Text>
        </Pressable>
    )

    return (
        <FlatList data={readings}
            renderItem={renderReadingItem}
            keyExtractor={(item) => item.id}
        />
    )
}
```

onDelete es una función declarada en *App.js* que por ahora solo hace un *console.log* con un mensaje para comprobar que funcione.

Ejercicio de clase

Completar la función de *onDelete* que toma el id de la receta como parámetro y la elimina del arreglo.

Agregando estilizado a los *Pressable*

Para poder dar un *feedback* visual al usuario, haciéndole entender que ha presionado el botón, el componente *Pressable* acepta una *prop* llamada *android_ripple*, que espera un objeto con la propiedad *color* que le agregara el efecto *ripple* con el color que nosotros pongamos en esa propiedad (en este caso, un naranja más oscuro)

Para lograr que el estilizado no sobrepase los límites establecidos por el componente *Text*, deberemos agregar un componente *View* que encierre a ambos y modificar el estilizado de la siguiente manera:

```
29  const styles = StyleSheet.create({
30      readingContainer: {
31          overflow: 'hidden',
32          borderRadius: 6,
33      },
34      readingItem: {
35          margin: 12,
36          padding: 8,
37          borderRadius: 6,
38          backgroundColor: "#206B21FF",
39          color: "#FFF"
40      },
41      readingText: {
42          color: 'white'
43      }
44  })
```

```
3   const ReadingList = ({
4       readings = [],
5       onDelete = () => { }
6   }) => {
7       const renderReadingItem = (itemData) => (
8           <View style={styles.readingContainer}>
9               <Pressable
10                  onPress={() => onDelete(itemData.item.id)}
11                  style={styles.readingItem}
12                  android_ripple={{ color: '#A64A08' }}>
13                  <Text >{itemData.item.text}</Text>
14              </Pressable>
15          </View>
16      )
17
18      return (
19          <FlatList data={readings}
20              renderItem={renderReadingItem}
21              keyExtractor={(item) => item.id}
22          />
23      )
24  }
25
```

Ahora observamos que luego de presionar, se distribuye por el componente el color marrón más oscuro con el efecto *ripple*.

Componente *Modal* en React Native

Supongamos que el cliente desea que el *ReadingInput* no esté siempre visible, sino que el mismo aparezca en un modal (es decir, un bloque que aparece en pantalla ante una interacción del usuario).

Afortunadamente, React Native viene incorporado con el componente *Modal*, que nos permitirá encerrar a *ReadingInput* entre sus etiquetas de la siguiente manera:

```
7     return (
8         <Modal>
9             <View style={styles.inputContainer}>
10                <TextInput
11                    style={styles.textInput}
12                    placeholder='Ingrese el nombre del libro'
13                    onChangeText={onChangeText}
14                />
15                <View style={styles.buttonInput}>
16                    <Button title='Agregar lectura' onPress={onPress} />
17                </View>
18            </View>
19        </Modal>
```

Si probamos ahora la aplicación, vemos que el modal no funciona correctamente ¿Por qué es eso? Bueno, debido a que el mismo está cubriendo toda la pantalla y está siempre visible.

Para que el Modal “aparezca y desaparezca” agregaremos un componente *Button* en *App.js* que maneje el estado de la visibilidad del mismo:

```
const App = () => {
  const [text, setText] = useState("");
  const [readings, setReadings] = useState([]);
  const [showModal, setShowModal] = useState(false);

  const handleReadingInput = (enteredText) => {
    setText(enteredText)
  };

  const handleAddReading = () => {
    setReadings((prevReadings) => [...prevReadings, { id: Math.random().toString(), text }]);
    setShowModal(false);
  }

  const handleDeleteReading = (id) => {
    setReadings(prevReadings => prevReadings.filter(reading => reading.id !== id))
  };

  const handleShowModal = () => {
    setShowModal(true);
  }

  const handleHideModal = () => {
    setShowModal(false);
  }
```

Con el seguimiento del valor de la visibilidad, se lo vamos a pasar por *props* a *ReadingInput* y utilizaremos las *props* nativas de *Modal; visible* y *animationType:*

```
const ReadingInput = ({
    visible = false,
    onChangeText = () => { },
    onPress = () => { },
    onCancel = () => { }
}) => {
    return (
        <Modal visible={visible} animationType='slide'>
            <View style={styles.inputContainer}>
                <TextInput
                    style={styles.textInput}
                    placeholder='Ingrese el nombre del libro'
                    onChangeText={onChangeText}
                />
                <View style={styles.buttonInputContainer}>
                    <View style={styles.buttonInput}>
                        <Button title='Cancelar' onPress={onCancel} />
                    </View>
                    <View style={styles.buttonInput}>
                        <Button title='Agregar lectura' onPress={onPress} />
                    </View>
                </View>
            </View>
        </Modal>
    )
}
```

Ahora si, el *Modal* responde al botón que agregamos.

Ejercicio de clase

Agregar botón cancelar al modal que nos permita cancelar el mismo. Re-estilizar el modal para que el *input* se encuentre por arriba de los dos botones, de la siguiente manera:

Fecha	Versionado actual	Autor	Observaciones
19/09/2022	1.0.0	Gabriel Golzman	Primera versión
11/04/2025	2.0.0	Gabriel Golzman	Segunda versión