Curso de React y React Native

Clase 11

Agenda de la clase

Agenda

- Repaso.
- ScrollView.
- List Views.
- Ejercicios.

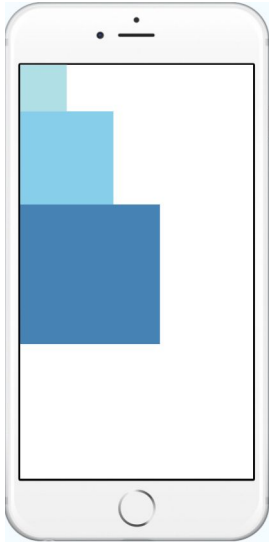
Repaso

REPASO

React Native – Height y Width

El `height` y `width` de un componente en React Native determinan su tamaño en pantalla. Todas las dimensiones en React Native son sin unidad, representan píxeles independientes a la densidad del celular.

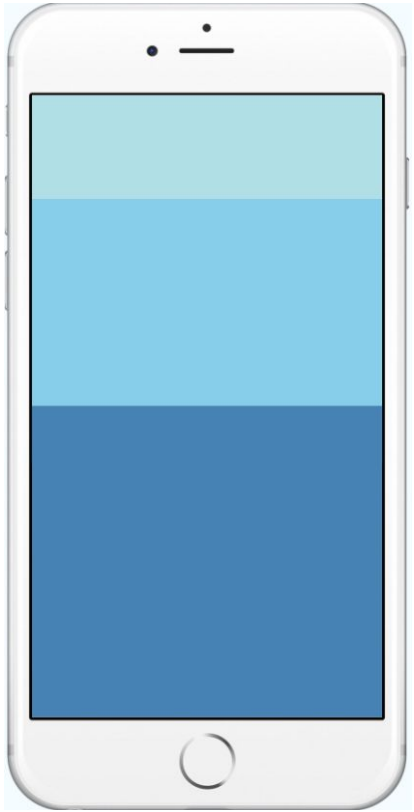
```
export default class FixedDimensionsBasics extends Component {
 render() {
   return (
     <View>
       <View style={{width: 50, height: 50, backgroundColor: 'powderblue'}} />
       <View style={{width: 100, height: 100, backgroundColor: 'skyblue'}} />
       <View style={{width: 150, height: 150, backgroundColor: 'steelblue'}} />
     </View>
   );
 }
}
```



REPASO

React Native – Flex

```
export default class FlexDimensionsBasics extends Component {
 render() {
   return (
     <View style={{flex: 1}}>
       <View style={{flex: 1, backgroundColor: 'powderblue'}} />
       <View style={{flex: 2, backgroundColor: 'skyblue'}} />
       <View style={{flex: 3, backgroundColor: 'steelblue'}} />
     </View>
   );
 }
}
```



Si el View padre no tiene `flex: 1`, no se dibuja nada, y si tuviera un `height` fijo, se distribuyen los hijos en ese `height`.

REPASO

React Native – Flexbox

En cada componente podemos utilizar Flexbox (Guide to Flexbox css)para determinar el layout de los hijos, de forma de garantizar un layout que se mantenga consistente entre distintas pantallas.

Las propiedades que más comúnmente se utilizan son:

- `flexDirection`: Determina el eje principal, si es horizontal (`row`) o vertical (`column`). Por defecto es vertical.
- `justifyContent`: Determina la distribución de los hijos a lo largo del eje principal, ya sea al comienzo, al final, en el centro, separados equitativamente, etc.
- `alignItems`: Determina la distribución de los hijos a lo largo del eje secundario.

REPASO

React Native – TextInput

Para crear inputs en React Native se utiliza el componente TextInput.

A diferencia de la web, donde se tiene el evento `onChange` para los `<input>`, al utilizar `TextInput` se cuenta con el evento `onChangeText`.

También existen otras propiedades comunes como `placeholder`, `multiline`, `numberOfLines`, `secureTextEntry`, `keyboardType`, etc.

REPASO

React Native – Button

En React Native tenemos un componente `Button` que se dibujara de forma distinta para cada plataforma, siguiendo los estándares de la plataforma.

```
import { AppRegistry, View, Button, Alert } from "react-native";
export default class ButtonExample extends Component {
 render() {
   return (
     <View style={{ padding: 10 }}>
       <Button onPress={() => { Alert.alert("You tapped the button!"); }} title="Press Me" />
     </View>
   );
 }
}
```

ScrollView

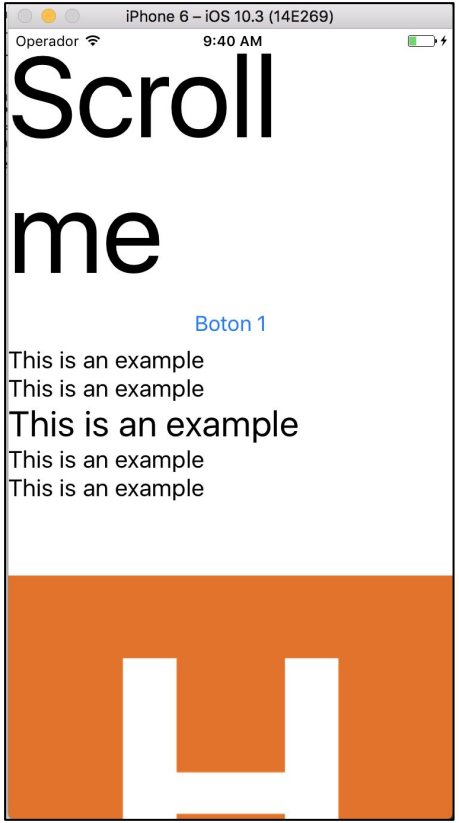
ScrollView

En React Native se encuentra el componente `ScrollView` que permite envolver una lista de componentes (no necesariamente homogéneos) de forma de que el contenido sea scrolleable, sin importar que el tamaño exceda el tamaño de pantalla.

El `ScrollView` es ideal para mostrar una cantidad pequeña de elementos que no entran en la pantalla de un dispositivo. No debe ser utilizado cuando es una lista muy grande de elementos ya que **todos** los hijos del `ScrollView` **serán renderizados**, no importa que no estén en pantalla.

ScrollView

```
const deviceWidth = Dimensions.get("window").width;
export default class ScrollExample extends Component {
   render() {
     return (
       <ScrollView>
         <Text style={{ fontSize: 96 }}>Scroll me</Text>
         <Button title="Boton 1" onPress={() => Alert.alert("Press")}/>
         <Text style={{ fontSize: 20 }}>This is an example</Text>
         <Image resizeMode="contain" style={{ width: deviceWidth }}
                source={require("./img/ha.png")} />
       </ScrollView>
     );
   }
}
```



ScrollView

El Scroll puede ser también horizontal, con la propiedad `horizontal`.

```
<ScrollView horizontal>
   <Text style={{ fontSize: 96 }}>Scroll me</Text>
   <Button title="Boton 1" onPress={() => Alert.alert("Press")} />
   <Text style={{ fontSize: 20 }}>This is an example</Text>
   <Image resizeMode="contain" style={{ width: deviceWidth }} source={require("./img/ha.png")} />
</ScrollView>
```

Si utilizamos un ScrollView con un solo hijo podemos utilizarlo para hacer zoom en el contenido, pero necesitamos pasarle dos nuevas *props*: `minimumZoomScale` y `maximumZoomScale`

ScrollView

En `iPhone`, si utilizamos un ScrollView con un solo hijo podemos utilizarlo para hacer zoom en el contenido, pero necesitamos pasarle dos nuevas *props*: `minimumZoomScale` y `maximumZoomScale`

```
export default class ScrollExample extends Component {
   render() {
     return (
       <ScrollView minimumZoomScale={1} maximumZoomScale={10}
         contentContainerStyle={styles.scrollViewContainer}
       >
         <Image source={require("./img/ha.png")} />
       </ScrollView>
     );
   }
}
```

List Views

Lists Views

En React Native contamos con dos componentes para mostrar listas de elementos: FlatList y SectionList.

`FlatList` es utilizada para mostrar una lista de datos scrolleables que son distintos, pero de estructura similar. Es buena para mostrar mucha cantidad de elementos, ya que a diferencia de `ScrollView`, `FlatList` sólo renderiza los elementos que son visibles en la pantalla.

`FlatList` necesita dos *props* principales: `data` y `renderItem`.

`data` es la fuente de información para la lista. `renderItem` toma un elemento de los datos y retorna un componente formateado para dibujar.

Cada uno de los items de `data` debería tener una propiedad `key` única.

```
export default class FlatListBasics extends Component {
   render() {
       const dataHarcoded = [ { key: 0, name: "Diego" },
           { key: 1, name: "Jorge" }, { key: 2, name: "Andres" },
           { key: 3, name: "Michael" }, { key: 4, name: "Sebastian" },
           { key: 5, name: "Federico" }, { key: 6, name: "Gaston" }
       ];
       return (
           <View style={styles.container}>
               <FlatList data={dataHarcoded}
                       renderItem={(listItem) => {
                           const item = listItem.item;
                           const index =  listItem.index;
                           return  <Text style={styles.item}>
                                   {item.name}
                               </Text>
                       }}
               />
           </View>
       );
   }
}
```

```
export default class FlatListBasics extends Component {
   render() {
       const dataHarcoded = [ { key: 0, name: "Diego" },
           { key: 1, name: "Jorge" }, { key: 2, name: "Andres" },
           { key: 3, name: "Michael" }, { key: 4, name: "Sebastian" },
           { key: 5, name: "Federico" }, { key: 6, name: "Gaston" }
       ];
       return (
           <View style={styles.container}>
               <FlatList data={dataHarcoded}
                       renderItem={function({item, index}) {
                           return  <Text style={styles.item}>
                                   {item.name}
                               </Text>
                       }}
               />
           </View>
       );
   }
}
```

Lists Views

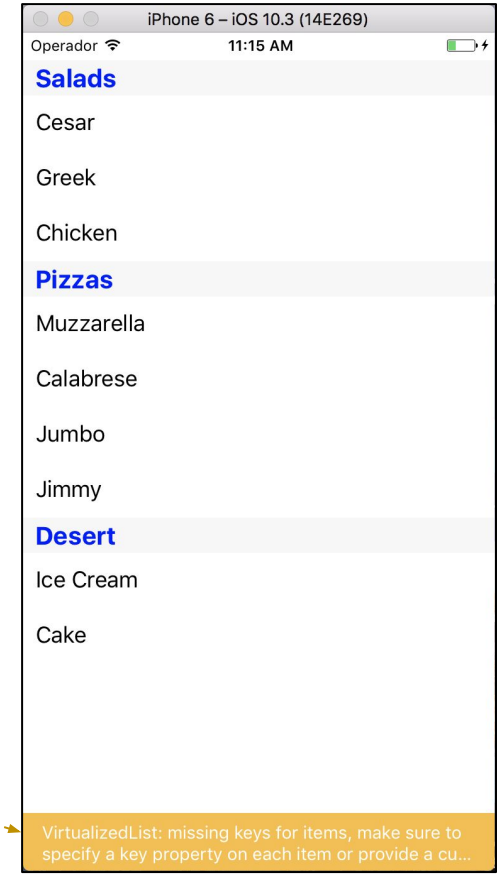
SectionList sirve para mostrar una lista de elementos, separados en secciones, y para cada sección se dibujara un "header".

Las principales props que recibe son tres:

- `sections`: Es el array de datos, un item para cada sección, pero a diferencia de FlatList, cada sección deberá tener dentro además el array de datos para esa sección. Por ejemplo:
- `renderItem`: Función para determinar cómo renderizar un item de una sección.
- `renderSectionHeader`: Función para determinar cómo renderizar un header de una sección.

```
export default class SectionListBasics extends Component {
   render() {
     const dataHarcoded = [
       { title: "Salads", data: ["Cesar", "Greek", "Chicken"] },
       { title: "Pizzas",
         data: ["Muzzarella", "Calabrese", "Jumbo", "Jimmy"] },
       { title: "Desert", data: ["Ice Cream", "Cake"] }
     ];
     return  <View style={styles.container}>
         <SectionList
           sections={dataHarcoded}
           keyExtractor={(item, index) => item.title + index}
           renderItem={({ item, index }) =>
                <Text style={styles.item}>{item}</Text>}
           renderSectionHeader={({ section, index }) =>
             <Text style={styles.sectionHeader}>{section.title}</Text>}
         />
       </View>
     );
   }
}
```

Los items también deben tener keys únicas.



Ejercicio

Ejercicios (1)

1. Crear un proyecto de react-native
2. Usando `FlatList` imprimir en pantalla una lista *scrolleable* de todas las `platforms` de la API de libraries.io.
3. Cada ítem de la lista deberá tener un `fontSize` de 60 y ser representado en el color que nos provee la API para cada platform.

Ejercicios (2)

1. Crear un proyecto de react-native
2. Usando `SectionList` imprimir en pantalla los resultados de buscar “react” en cada una de las platforms. Naturalmente, tendremos *una section para cada platform*.

3. *Opcionalmente*, pueden hacer la búsqueda sujeta al valor de un TextInput (que colocaremos a tope de pantalla), en lugar de tener “react” hardcodeado.