APLICACIONES MÓVILES MULTIPLATAFORMA

LABORATORIO N° 05

Flatlist y Axios

Alumno(s):		***Nota***	
Grupo:		***Ciclo:V***	

Criterio de Evaluación	*Excelente (4pts)*	*Bueno (3pts)*	*Requiere mejora (2pts)*	*No acept. (0pts)*	*Puntaje Logrado*
Agregar componentes de listados dinámicos					
Instalar y utilizar axios					
Consumir APIs de terceros					
Realiza con éxito lo propuesto en el laboratorio					
Es puntual y redacta el informe adecuadamente					

Laboratorio 05:
Flatlist y Axios

Objetivos:

Al finalizar el laboratorio el estudiante será capaz de:

- Entender el funcionamiento del componente Flatlist
- Desarrollar aplicaciones web enfocadas a componentes
- Manejar de manera básica axios para consumir APIs

Seguridad:

- Ubicar maletines y/o mochilas en el gabinete del aula de Laboratorio.
- No ingresar con líquidos, ni comida al aula de Laboratorio.
- Al culminar la sesión de laboratorio apagar correctamente la computadora y la pantalla, y ordenar las sillas utilizadas.

Equipos y Materiales:

- Una computadora con:
  - Windows 7 o superior
  - VMware Workstation 10+ o VMware Player 7+
  - Conexión a la red del laboratorio
- Máquinas virtuales:
  - Windows 7 Pro 64bits Español - Plantilla
- Instalador de node.js

Procedimiento:

Lab Setup

1. **Configuración de proyecto**
   1.1. En su carpeta de trabajo, inicie un nuevo proyecto de react-native. En caso de usar una instalación nativa, utilice:

```
>react-native init lab05
```

   1.2. Iniciemos nuestro proyecto. En caso de usar la instalación nativa, utilizar:

```
>cd lab05

\lab05>react-native run-android
```

   1.3. Ahora que conocemos el ciclo de vida, hagamos una pequeña prueba del inicio de montaje. Cambiaremos el contenido del archivo App.js:

```
export default class App extends Component<Props> {
  state = {
    contador: 1
  }
  componentDidMount(){
    setInterval(() => this.setState({contador: this.state.contador + 1}),1000);
  }
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>Veamos nuestro contador autónomo</Text>
        <Text style={styles.instructions}>{this.state.contador}</Text>
      </View>
    );
```

setInterval es una función nativa de JavaScript que recibe dos parámetros, primero una función a repetir, y luego el tiempo en milisegundos a ser repetida. En nuestro ejemplo, hacemos que nuestro contador se incremente a sí mismo con cada segundo que transcurre. Adjunte una captura en GIF de su resultado.

1.4. Podemos aprovechar esto para diversas cosas, como crear por ejemplo un reloj.

```
export default class App extends Component<Props> {
  state = {
    fecha: new Date()
  }
  componentDidMount(){
    setInterval(() => this.setState({fecha: new Date()}),1000);
  }
  render() {
    const fecha = this.state.fecha.getHours()+':'+this.state.fecha.getMinutes()+
      ':'+this.state.fecha.getSeconds();
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>Veamos nuestro reloj</Text>
        <Text style={styles.instructions}>{fecha}</Text>
      </View>
```

Adjunte captura en GIF de su resultado

2. Listados en React Native

2.1. Hasta ahora, hemos visto componentes básicos como botones, textos e imágenes. Pasaremos a utilizar uno de los más utilizados: el componente **FlatList**. Modificaremos una vez más nuestro archivo App.js con el siguiente código.

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

type Props = {};
export default class App extends Component<Props> {
  render() {
    return (<View>
      <FlatList
        data={[{key: 'a'}, {key: 'b'}, {key: 'c'}]}
        renderItem={({item}) => <Text>{item.key}</Text>}
      />
    </View>);
  }
}
```

Las partes más importantes de este snippet (porción de código) son:

- La importación de FlatList al inicio del script
- La propiedad función renderItem dentro de Flatlist, que se encargará de renderizar cada ítem del listado.
- La propiedad objeto data, que tiene que contener un array de elementos.

De esta forma, **Flatlist** podrá renderizar los arrays que reciba en forma de listado, lo cual nos será muy útil para mostrar contenido de bases de datos o contenido estático.

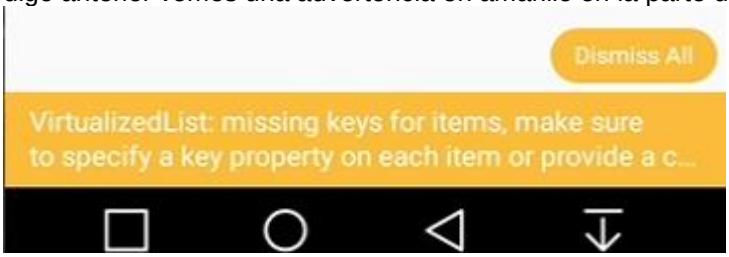
Fíjese que **renderItem** tiene una propiedad (en este caso llamada ítem, puede tener cualquier nombre) la cual recibe el elemento a iterar.

2.2. En el ejemplo anterior vimos que el componente recibía un listado de objetos. Sin embargo, también se puede utilizar un array convencional.

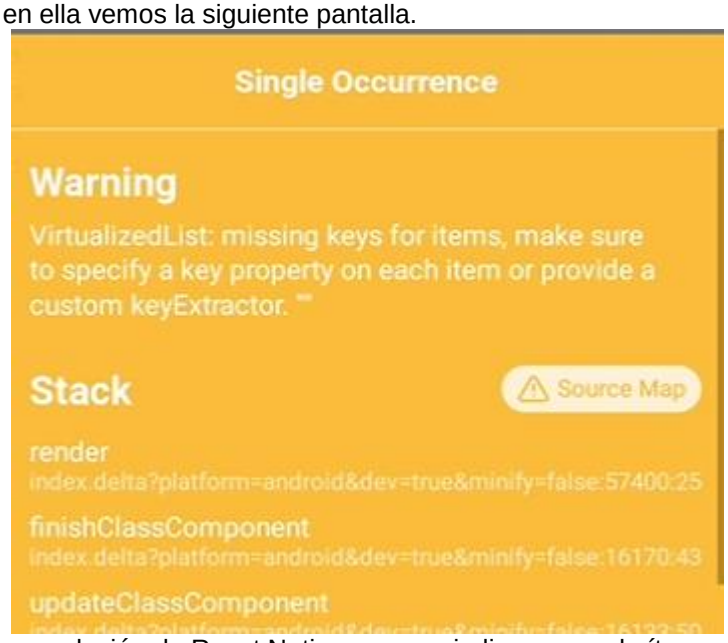
```
export default class App extends Component<Props> {
  render() {
    return (<View>
      <FlatList
        data={['a','b','c','d']}
        renderItem={({item}) => <Text>{item}</Text>}
      />
    </View>);
```

Adjunte una captura de su resultado.

2.3. Al ejecutar el código anterior vemos una advertencia en amarillo en la parte de abajo



Y al hacer click en ella vemos la siguiente pantalla.



Esta es una recomendación de React Native que nos indica que cada ítem en un listado debe tener una propiedad **key**, con la cual podrá identificar rápidamente cada nodo y renderizar el correcto en caso de ser necesario. Como nuestro nuevo arreglo no es un objeto ni tiene la propiedad key, deberemos utilizar la función **keyExtractor** que nos permite indicarle a React Native como comportarse en ese caso.

`keyExtractor`

```
(item: object, index: number) => string;
```

Used to extract a unique key for a given item at the specified index. Key is used for caching and as the react key to track item re-ordering. The default extractor checks `item.key`, then falls back to using the index, like React does.

Esto no impide que el programa se ejecute, pero si causa una baja en el rendimiento, que será corregida con agregar la función mencionada. Fíjese que, según la documentación oficial, nosotros debemos devolver un string, por lo que después de obtener el índice del elemento le concatenaremos algo vacío para forzarlo a cadena de texto.

```
return (<View>
  <FlatList
    keyExtractor={(item,index) => index+''}
    data={['a','b','c','d']}
    renderItem={({item}) => <Text>{item}</Text>}
  />
</View>);
```

Adjunte una captura de su resultado sin advertencias.

2.4. Hasta ahora nuestro listado es muy básico. Vamos a crear primero los estilos para diferencias una celda de la otra. Esto podemos hacerlo en la parte inferior del script.

```
const styles = StyleSheet.create({
  ItemEven: {
    color: '#2B4B6F'
  },
  ItemUneven: {
    backgroundColor: '#D46A6A',
    color: 'white'
  }
});
```

Ahora procedemos a modificar el código del método **render**.

```
render() {
  const people = [
    {name: 'Carlos',lastname: 'Amezquita'},
    {name: 'Marta', lastname: 'Chavez'},
    {name: 'Pedro', lastname: 'Picapiedra'},
    {name: 'Lucía', lastname: 'Gonzalez'}
  ]
  return (<View>
    <FlatList
      keyExtractor={(item,index) => index+''}
      data={people}
      renderItem={({item,index}) => {
        return (<Text style={index%2===0?styles.ItemEven:styles.ItemUneven}>
          {item.name}
        </Text>);
      }}
    />
  </View>);
}
```

Adjunte una captura de su resultado

Vemos varias cosas importantes, como el hecho de que data de **Flatlist** puede recibir una variable, no solamente tener objetos declarados dentro de sí como habíamos visto hasta el momento.

Así mismo, al momento de declarar el estilo del elemento **Text** hacemos un **if** en una sola línea, preguntando si el índice (recibido como segundo parámetro en **renderItem**) es modular entre dos, es decir, si es par o impar. En caso de ser par, utilizará los estilos definidos como **ItemEven**, y en caso de ser impar utilizará los estilos **ItemUneven**.

2.5. Añadiremos capacidades interactivas al listado con el componente TouchableOpacity. Primero, lo importaremos en la invocación de nuestros elementos de React Native

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

Modificaremos la función renderItem dentro del FlatList, encapsulando nuestro componente anterior con dicho componente.

```
renderItem={({item,index}) => {
  return (<TouchableOpacity onPress={() => this.onPressHandler(item)}>
    <Text style={index%2===0?styles.ItemEven:styles.ItemUneven}>
      {item.name}
    </Text>
  </TouchableOpacity>);
}}
```

Ahora procederemos a crear una función onPressHandler, que se disparará en cada ítem. En nuestro caso la crearemos antes de la invocación del render.

```
export default class App extends Component<Props> {
  onPressHandler = item => {
    alert('El apellido es '+item.lastname);
  }
  render() {
```

Adjunte un gif con su resultado.

2.6. Agregaremos una cabecera para poder filtrar nuestros resultados. Primero vamos a cambiar la invocación de la información, declarando nuestro arreglo como una variable externa al componente. Realice los siguientes cambios y verifique el funcionamiento del componente.

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

const people = [
  {name: 'Carlos',lastname: 'Amezquita'},
  {name: 'Marta', lastname: 'Chavez'},
  {name: 'Pedro', lastname: 'Picapiedra'},
  {name: 'Lucía', lastname: 'Gonzalez'}
];

type Props = {};
export default class App extends Component<Props> {
  state = {
    data: people
  }
  onPressHandler = item => {
    alert('El apellido es '+item.lastname);
  }
  render() {
    return (<View>
      <FlatList
        keyExtractor={(item,index) => index+''}
        data={this.state.data}
        renderItem={({item,index}) => {
          return (<TouchableOpacity onPress={() => this.onPressHandler(item)}>
            <Text style={index%2===0?styles.ItemEven:styles.ItemUneven}>
```

Hacemos esta manipulación para poder seguir contando con las personas originales a pesar de filtrarlas. Lo ideal sería hacerlo en un componente aparte.

2.7. Agregaremos la propiedad ListHeaderComponente para agregar un componente de cabecera a nuestro listado.

```
<FlatList
  keyExtractor={(item,index) => index+''}
  data={this.state.data}
  renderItem={({item,index}) => {
    return (<TouchableOpacity onPress={() => this.onPressHandler(item)}>
      <Text style={index%2===0?styles.ItemEven:styles.ItemUneven}>
        {item.name}
      </Text>
    </TouchableOpacity>);
  }}
  ListHeaderComponent={this.renderHeader}
/>
```

Como hace mención a un método renderHeader dentro de nuestro componente (this), pasaremos a crear este método.

```
  onPressHandler = item => {
    alert('El apellido es '+item.lastname);
  }
  renderHeader = () => {
    return (<TextInput
      style={{height: 40, borderColor: 'gray', borderWidth: 1}}
      onChangeText={this.searchHandler}
      value={this.state.text}
    />);
  }
  render() {
    return (<View>
      <FlatList
```

TextInput hace referencia a la variable text dentro del state por lo que agregaremos ese estado

```
export default class App extends Component<Props> {
  state = {
    data: people,
    text: ''
  }
  onPressHandler = item => {
```

Finalmente, TextInput hace referencia a searchHandler en su propiedad onChangeText, la cual se disparará cada vez que escribamos algo y ejecutará el filtrado de nombres. Crearemos esa función a continuación.

```
renderHeader = () => {
  return (<TextInput
    style={{height: 40, borderColor: 'gray', borderWidth: 1}}
    onChangeText={this.searchHandler}
    value={this.state.text}
  />);

searchHandler = text => {
  this.setState({
    text: text
  }, () => {
    const newData = people.filter(item => {
      const itemData = `${item.name.toUpperCase()}`;
      const textData = text.toUpperCase();
      return itemData.indexOf(textData) > -1;
    });
    this.setState({
      data: newData
    });
  });
}
render() {
  return (<View>
    <FlatList
```

Aquí pasan varias cosas:

- Asignamos el nuevo valor del input a nuestro state text
- this.setState tiene en realidad dos argumentos, el primero es el objeto que modifica al estado interno del componente, y el segundo es una función callback, es decir, se ejecuta después de modificado el estado. Esto nos sirve cuando queremos que las funciones se ejecuten en el orden que deseamos.
- Realizamos un people.filter para poder filtrar nuestro arreglo. Dentro ejecutamos la función indexOf que pregunta si lo ingresado (textData) se encuentra dentro de itemData. En caso la respuesta sea mayor a -1 significa que la cadena de texto se encuentra dentro del ítem comparado, y pasará el filtro.
- Hacemos un toUpperCase a los strings, para compararlos en mayúsculas sin importarnos como estaba el texto original. Si deseas que la comparación se realice sin este truco, funcionará siempre que se escriba el nombre tal cual.

Adjunte un gif de su resultado.

2.8. Flatlist tiene más propiedades y métodos útiles para nuestros listados. Los iremos explorando a lo largo del curso, pero siempre puedes ahondar en la documentación.

https://facebook.github.io/react-native/docs/flatlist

3. Axios

3.1. Tenemos la API de consulta de películas YTS. Haremos uso de esta para ejemplificar como acceder a una consulta remota y renderizar la data.

https://yts.am/api

3.2. Procedemos a instalar axios en nuestro proyecto, deteniendo la consola y luego ejecutando el siguiente comando.

```
a>npm install --save axios
```

3.3. Agregaremos la dependencia de axios y modificaremos el método componentDidMount (mejor dicho, crearemos uno)

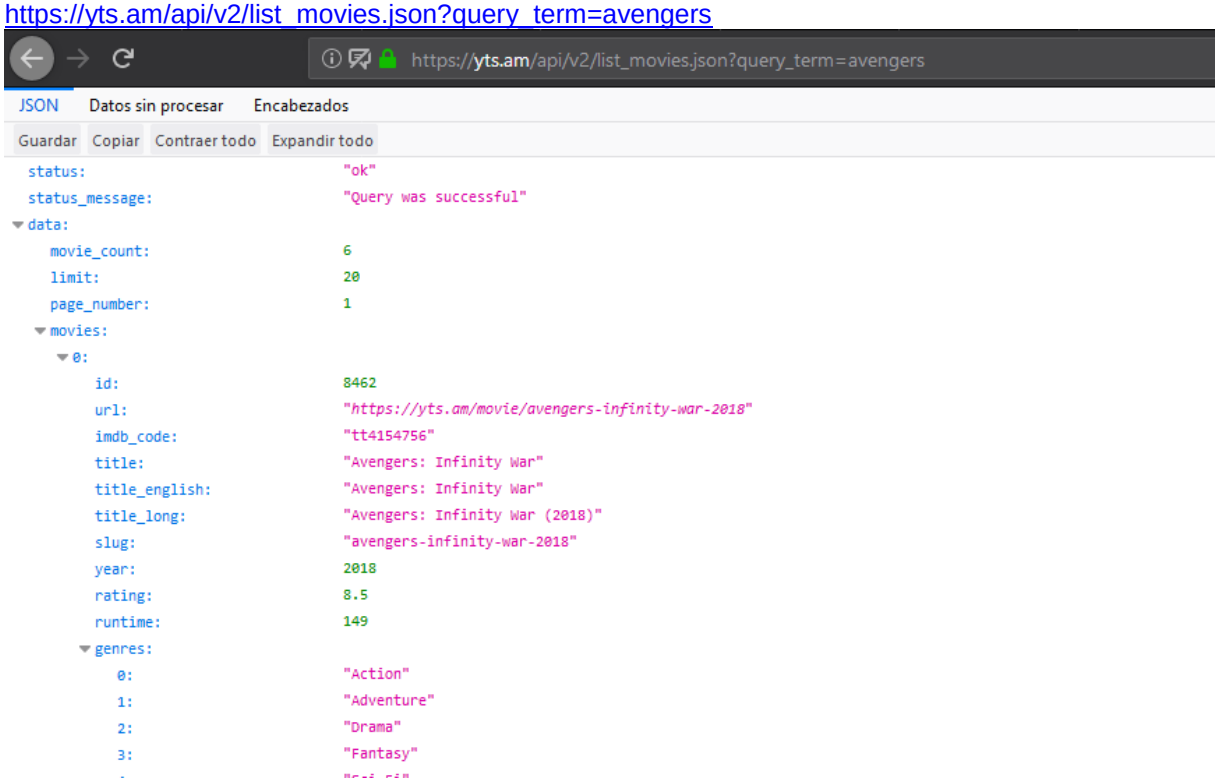
```
import React, {Component} from 'react';
import {StyleSheet, Text, View, FlatList, TouchableOpacity, TextInput} from 'react-native';
import axios from 'axios';

type Props = {};
export default class App extends Component<Props> {
  state = {
    loading: false,
    data: []
  }
  componentDidMount(){
    this.setState({loading: true});
    axios({
      method: 'GET',
      url: 'https://yts.am/api/v2/list_movies.json'
    }).then(response => {
      this.setState({
        loading: false,
        data: response.data.data.movies
      });
    }).catch(err => {
      this.setState({loading: false});
      console.warn(err);
    })
  }
  onPressHandler = item => {
```

3.4. Modificaremos de igual forma nuestro render y el onPressHandler, para que muestren el título de la película y la descripción de la misma al hacer click.

```
onPressHandler = item => {
  alert(item.description_full);
}
render() {
  let contenido = (<Text>
    Cargando, espere por favor...
  </Text>);
  if(!this.state.loading){
    contenido = (<FlatList
      keyExtractor={(item,index) => index+''}
      data={this.state.data}
      renderItem={({item,index}) => {
        return (<TouchableOpacity onPress={() => this.onPressHandler(item)}>
          <Text style={index%2===0?styles.ItemEven:styles.ItemUneven}>
            {item.title_long}
          </Text>
        </TouchableOpacity>);
      }}
      ListHeaderComponent={this.renderHeader}
    />);
  }
  return (<View>
    {contenido}
  </View>);
}
```

Fíjese en la variable contenido. Inicialmente recibe un componente de texto, a menos que le digamos que ya nuestro componente no está "cargando". Esto sucede en componentDidMount cuando obtenemos una respuesta del servidor.
Axios realiza una consulta por método GET a la API de YTS y recibe una respuesta de películas en JSON, lo cual facilita nuestro trabajo. Luego nosotros debemos enviar esa información a nuestro state por lo que la introducimos en nuestro atributo data del setState, al mismo tiempo que le decimos al componente que ya terminó de cargar.

4. Ejercicio propuesto
   4.1. La misma URL a la que hemos consultado tiene el parámetro query_term. Este parámetro permite filtrar los resultados según el término que le enviemos, buscando entre Título, Código IMDB, Director o Actores de la película.
   https://yts.am/api/v2/list_movies.json?query_term=avengers



   Agregue una cabecera a la lista creada que pueda buscar entre películas y devolver estos resultados.
   Pista: La ejecución de la búsqueda ya no se realizará en el componentDidMount, sino en la función que se ejecute después de actualizar el estado del buscador, como en nuestro ejemplo anterior, en searchHandler, lo cual ayudará a que cada vez que escribamos, se realice dicha búsqueda en internet.

5. **Finalizar la sesión**
   5.1. Apagar el equipo virtual
   5.2. Apagar el equipo

Conclusiones:

Indicar las conclusiones que llegó después de los temas tratados de manera práctica en este laboratorio.