APLICACIONES MÓVILES MULTIPLATAFORMA

LABORATORIO N° 04

Componentes en React Native

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

Criterio de Evaluación	*Excelente (4pts)*	*Bueno (3pts)*	*Requiere mejora (2pts)*	*No acept. (0pts)*	*Puntaje Logrado*
Instala con éxito React Native					
Utiliza Next-Gen Javascript					
Desarrolla aplicaciones con React Native					
Realiza con éxito lo propuesto en el laboratorio					
Es puntual y redacta el informe adecuadamente					

Laboratorio 04:
Componentes en React Native

Objetivos:

Al finalizar el laboratorio el estudiante será capaz de:

- Entender el funcionamiento de React Native
- Desarrollar aplicaciones web enfocadas a componentes
- Implementación correcta de Next-Gen Javascript

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 lab04
```

   1.2. En caso de estar usando expo, utilice:

```
>expo init lab04
```

   1.3. En caso de estar usando expo, seleccione la opción blank:

```
C:\Users\favio\code\tecsup>expo init lab04Expo
There is a new version of expo-cli available (2.13.0).
You are currently using expo-cli 2.6.11
Run `npm install -g expo-cli` to get the latest version
? Choose a template:
> blank
  minimum dependencies to run and an empty root component
  tabs
  several example screens and tabs using react-navigation
```

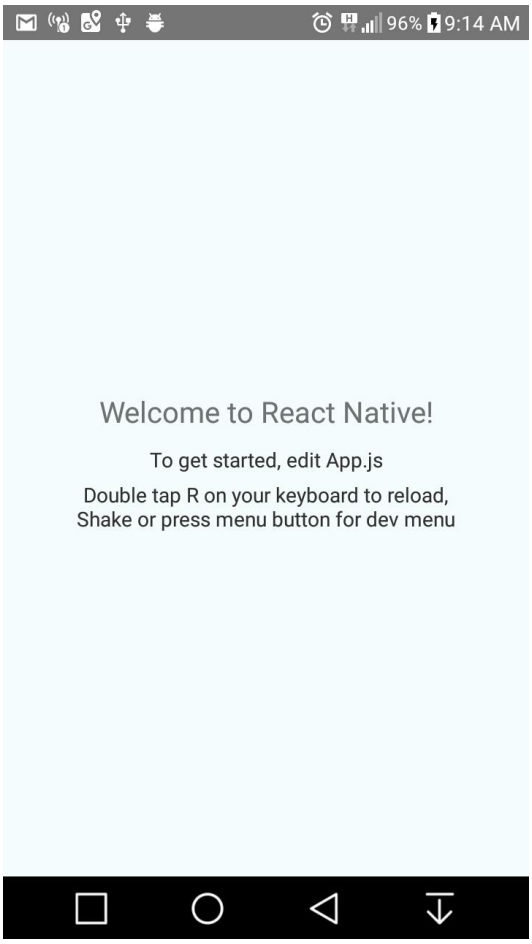
   1.4. Iniciemos nuestro proyecto. En caso de usar la instalación nativa, utilizar:

```
cd C:\Users\favio\code\tecsup\lab04 && react-native run-android
```

1.5. En caso de haber usado expo, utilizar:

```
cd lab04Expo
npm start
```

1.6. Si todo ha salido bien, deberías tener una pantalla como la siguiente. En caso contrario, consulta con el profesor.



1.7. Cambiaremos el contenido del archivo App.js por el siguiente:

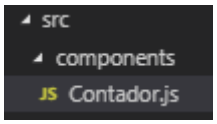
```
import React, {Component} from 'react';
import {StyleSheet, Text, View} from 'react-native';

export default class App extends Component{
    render() {
        return (
            <View style={styles.container}>
                <Text style={styles.welcome}>Nuestro primer componente</Text>
            </View>
        );
    }
}

const styles = StyleSheet.create({
    container: {
        flex: 1,
        justifyContent: 'center',
        alignItems: 'center',
        backgroundColor: '#F5FCFF',
    }
});
```

2. Creación de un componente

2.1. Crearemos la carpeta src, que contendrá todo nuestro código fuente. Dentro crearemos la carpeta components para nuestros componentes y finalmente crearemos el archivo Contador.js



2.2. Contador.js deberá tener la siguiente estructura:

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

class Contador extends Component{
    render(){
        return (<View>
            <Text>Mi contador: </Text>
        </View>);
    }
}

export default Contador;
```

2.3. Para utilizar el componente recién creado, debemos importarlo e invocarlo. Modificaremos App.js para importar Contador desde su ruta (fíjese que no es necesario poner la extensión cuando se trata de un archivo JavaScript) y luego debemos invocarlo como un elemento HTML con su nombre propio (<Contador />)

```
import Contador from './src/components/Contador';

export default class App extends Component{
    render() {
        return (
            <View style={styles.container}>
                <Text style={styles.welcome}>Nuestro
                <Contador />
            </View>
        );
    }
}
```

2.4. Guarde los cambios y verifique la aparición del contador.

2.5. Podemos invocar todas las veces que necesitemos a Contador. Agreguemos dos iteraciones más del mismo en App.js

```
return (
    <View style={styles.c
        <Text style={style
        <Contador />
        <Contador />
        <Contador />
    </View>
);
```

2.6. Al igual que los elementos HTML, los componentes en React pueden recibir “propiedades” que les indican cómo comportarse. Para invocar estas propiedades, basta con llamar a la variable interna **this.props**

Modificaremos primero Contador.js

```
class Contador extends Component{
    render(){
        return (<View>
            <Text>Mi contador: {this.props.valor}</Text>
        </View>);
```

Ahora modificaremos App.js

```
render() {
    return (
        <View style={styles.container}>
            <Text style={styles.welcome}>Nuestro primer componente</Text>
            <Contador valor='1' />
            <Contador valor='2' />
            <Contador valor='3' />
        </View>
    );
```

Guardemos los cambios y veamos el resultado.

Nótese que para invocar a una variable Javascript, es decir, su valor, debimos usar llaves para que React sepa diferenciar entre una cadena de texto y un valor dinámico.

3. Estado del componente

3.1. Así como los componentes reciben propiedades externas que les indican cómo comportarse, cada uno puede tener sus características propias que le permiten diferenciarse. Esto se llama estado y es una de las características más útiles de React. Podemos acceder al estado a través de la variable reservada **this.state**

Modifiquemos nuestro archivo Contador.js

```
class Contador extends Component{
    state = {
        valor: 0
    }
    render(){
        return (<View>
            <Text>Mi contador: {this.props.valor}</Text>
            <Text>Otro valor: {this.state.valor}</Text>
        </View>);
    }
```

Guarde los cambios y vea la diferencia.

3.2. Para modificar el estado, debemos hacer uso de la función **this.setState**

Para disparar este evento, vamos a crear dos botones, uno que incremente el valor, y otro que lo disminuya. Modifiquemos nuevamente el código de Contador.js

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

class Contador extends Component{
    state = {
        valor: 0
    }
    render(){
        return (<View>
            <Text>Mi contador: {this.props.valor}</Text>
            <Text>Otro valor: {this.state.valor}</Text>
            <Button
                title='Disminuir'
                />
            <Button
                title='Incrementar'
                color="#841584"
                />
        </View>);
```

Para conocer más de las propiedades de Button (como title y color), podemos ahondar en la documentación de React Native: https://facebook.github.io/react-native/docs/button.html

Note que a diferencia de los otros elementos, Button está escrito con saltos de línea. Eso no genera ningún problema de compilación y suele ser utilizado para crear código más legible, podríamos haber escrito el mismo código en una sola línea como por ejemplo:

```
return (<View>
    <Text>Mi contador: {this.props.valor}</Text>
    <Text>Otro valor: {this.state.valor}</Text>
    <Button title='Disminuir' />
    <Button title='Incrementar' color="#841584" />
</View>);
```

Guarde los cambios y verifique el resultado

3.3. Procederemos a agregar los eventos que modifican el estado. Para esto, el componente Button tiene una propiedad llamada **onPress**, que se dispara cada vez que el usuario presiona el componente en la pantalla.
Modificaremos Contador.js nuevamente.

```
    state = {
        valor: 0
    }
    disminuirHandler = () => {
        this.setState({
            valor: this.state.valor - 1
        });
    }
    incrementarHandler = () => {
        this.setState({
            valor: this.state.valor - 1
        });
    }
    render(){
        return (<View>
            <Text>Mi contador: {this.props.valor}</Text>
            <Text>Otro valor: {this.state.valor}</Text>
            <Button
                title='Disminuir'
                onPress={this.disminuirHandler}
                />
            <Button
                title='Incrementar'
                color="#841584"
                onPress={this.incrementarHandler}
                />
        </View>);
```

Guarde los cambios y verifique el funcionamiento de nuestros botones.
Aquí han sucedido varias cosas:

- Hemos invocado a la propiedad **onPress**, quien a la vez llama a un evento personalizado que hemos creado nosotros. Fíjese que el nombre es libre, yo le puse la acción (incrementar o disminuir) seguido por la palabra handler (manejador) pero usted puede colocar cualquier nombre.
- El evento ha sido declarado antes del **render**, puede ser declarado después pero es necesario que sea invocado como un método de la clase Contador.
- Para modificar el estado, utilizamos la función **this.setState** ya que esta invoca internamente al ciclo de vida del componente. Esta es quizás la parte más importante del ejercicio.

3.4. Modifiquemos Contador.js, ahora queremos declarar el valor por defecto de su estado inicial. Para esto modificaremos una parte del state y otra del render.

```
class Contador extends Component{
    state = {
        valor: this.props.valor
    }
```

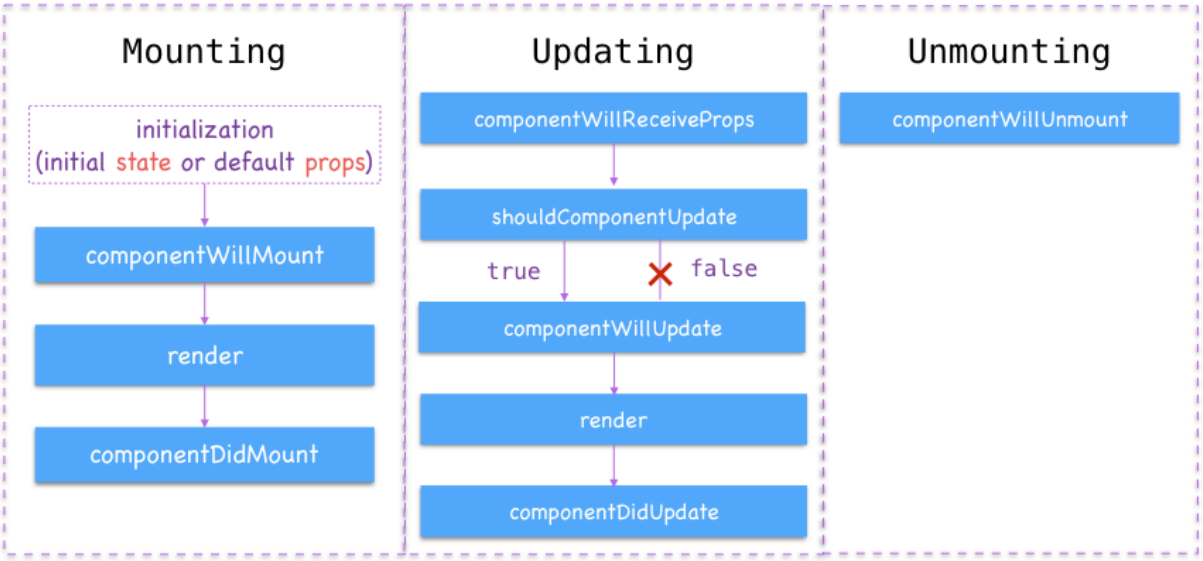
```
return (<View>
    <Text>Mi contador: {this.state.valor}</Text>
    <Button
```

Esto significa que podemos utilizar la propiedad recibida inicialmente para configurar nuestro estado. Verifique el funcionamiento actual de los tres contadores y comente al respecto.

4. Manejo de valores en controles
   4.1. Vamos a agregar un control que me permita setear el valor inicial de los contadores. Para esto, modificaremos el componente App.js

   4.2. asdfds

5. Ciclo de vida del componente
   5.1. Los componentes en React tienen un ciclo de vida de tres etapas: Montado, Actualización y Desmontado.



- El montado se da la primera vez que se renderiza el componente. Como se puede apreciar en la gráfica, se invocará al método render y luego al método componentDidMount. Es en este método donde generalmente haremos llamadas asíncronas, solamente cuando el componente se ha terminado de montar.
- La etapa de actualización se da siempre que haya un cambio de propiedades o de estado interno. Esta etapa, al igual que la de montado, invoca nuevamente al render ya que de acuerdo al cambio de propiedades o estado, nuestro componente debe renderizarse y mostrar una nueva configuración.
- La última etapa, de desmontado, se invoca cuando el componente va a desaparecer de la pantalla, digamos porque nosotros mismos lo hacemos desaparecer por un estado o por cambiar de pantalla.

   5.2. Vamos a agregar un control que me permita setear el valor inicial de los contadores. Para esto, modificaremos el componente App.js

   En esta parte veremos varias cosas importantes, por ejemplo, el componente TextInput tiene la propiedad onChangeText que se dispara cuando modificamos el texto del mismo. Así mismo, como vamos a modificar el estado libremente, puede que el usuario ingrese un valor no numérico, por lo que haremos una verificación de si es un número válido o no con la función isNaN (is not a number) que en caso sea un texto no numérico, devolverá verdadero.
   Fíjese en el render en el cómo podemos hacer un if condicional de forma inline, es decir, de una sola línea e incluyendo componentes de React.

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

import Contador from './src/components/Contador';

export default class App extends Component{
    state = {
        valorInicial: '0'
    }
    iniciarContadoresHandler = (texto) => {
        this.setState({
            valorInicial: texto
        });
    }
    render() {
        return (
            <View style={styles.container}>
                <Text style={styles.welcome}>Nuestro primer componente</Text>
                <TextInput
                    value={this.state.valorInicial}
                    onChangeText={this.iniciarContadoresHandler}
                    />
                {!isNaN(this.state.valorInicial)?(<View>
                    <Contador valor={parseInt(this.state.valorInicial)} />
                    <Contador valor={parseInt(this.state.valorInicial)+1} />
                    <Contador valor={parseInt(this.state.valorInicial)+2} />
                </View>):(<Text>Debe ingresar un número!</Text>)}
            </View>
```

5.3. Sin embargo, al ejecutar el código, vemos que nuestro input puede cambiar de valor pero no modifica a los contadores. Esto se debe a que el valor del contador se setea solamente la primera vez que se carga el componente, en la etapa de montado.
Vamos a utilizar la etapa de actualización para hacer funcionar nuestros contadores con el nuevo input, así que modifiquemos Contador.js

```
class Contador extends Component{
    state = {
        valor: this.props.valor
    }
    componentDidUpdate(oldProps,oldState){
        if(oldProps.valor!==this.props.valor && !isNaN(this.props.valor)){
            this.setState({
                valor: this.props.valor
            });
        }
    }
    disminuirHandler = () => {
        this.setState({
```

Lo que hemos hecho es validar si el componente se está actualizando a través de la etapa de actualización con el método componentDidUpdate. Esta función recibe dos parámetros, las propiedades anteriores a la modificación (oldProps) y el estado anterior a la modificación

(oldState) por lo que basta para que nosotros preguntemos si el valor es distinto al valor actual y sobre todo, preguntar si el valor es un número.

6. Ejercicio propuesto
   6.1. Realizar la creación de una calculadora con lo aprendido.

7. **Finalizar la sesión**
   7.1. Apagar el equipo virtual
   7.2. Apagar el equipo

Conclusiones:

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