Material de apoyo a las clases.

¿Qué es ReactJS?

Es una librería para el desarrollo de interfaces de usuario interactivas.

Las interfases son definida usando JavaScript que genera dinámica y eficientemente las páginas web directamente en el navegador.

El funcionamiento normal de un navegador es bajar la página web como un archivo de HTML, este archivo es analizado (parse) y convertido a una representación de memoria (DOM - Document Object Model) para luego ser mostrada (render) en la pantalla.

El DOM puede ser consultado y modificado desde JavaScript. Cualquier cambio en realizado en el DOM es luego reflejando inmediatamente en pantalla.

React aprovecha esta funcionalidad de los navegadores para generar dinámicamente el contenido eliminado la necesidad de bajar el archivo HTML y convertirlo luego a DOM. En su lugar permite generar directamente el DOM; además incorpora mecanismos para que cuando se realice una modificación del contenido se actualice la página inteligentemente afectando solo los elementos alterados.

Este mecanismos permite una forma nueva de programar las interfaces conocida como *programación reactiva*:

> El programador crean componentes para mostrar el estado de la aplicación, luego programa los cambios de estado y React se encarga de mostrar automáticamente y eficientemente la pantalla.

¿Cómo funciona ReactJS?

Una página web tradicional podría implementar una 'agenda' así:

```
<div id="agenda">
        <h1>Agenda</h1>
        <ul>
                <li>Alejandro</li>
                <li>Beatriz</l1>
                <li>Carlos</li>
        </ul>
</div>
```

El navegador genera el DOM que puede ser consultado y modificado usando JavaScript

```
// Obtener el elemento agenda (document es el DOM de la página)
let agenda = document.getElementById('agenda');

```

```
// Obtener el contenido de la lista
let contactos = agenda.getElementsByTagName('li');
for (let i = 0; i < contactos.length; i++) {
        console.log(contactos[i].innerHTML);
}

// Cambiar el nombre de 'Beatriz' por 'Belen'
contactos[1].innerHTML = 'Belen';

// Cambiar el estilo de 'Carlos'
contactos[2].style.color = 'red';

// Crear un nuevo elemento
let nuevoElemento = document.createElement('li');
nuevoElemento.innerHTML = 'David';
let lista = agenda.getElementsByTagName('ul')[0]
lista.appendChild(nuevoElemento);
```

La agenda podría ser generar desde JavaScript usando el DOM de la siguiente manera:

```
// Crear el elemento h1
let h1 = document.createElement('h1');
let h1Text = document.createTextNode('Agenda');
h1.appendChild(h1Text);

// Crear el elemento ul
let ul = document.createElement('ul');

// Crear los elementos li
let liAlejandro = document.createElement('li');
let liAlejandroText = document.createTextNode('Alejandro');
liAlejandro.appendChild(liAlejandroText);

let liBeatriz = document.createElement('li');
let liBeatrizText = document.createTextNode('Beatriz');
liBeatriz.appendChild(liBeatrizText);

let liCarlos = document.createElement('li');
let liCarlosText = document.createTextNode('Carlos');
liCarlos.appendChild(liCarlosText);

// Añadir los elementos li al ul
ul.appendChild(liAlejandro);
ul.appendChild(liBeatriz);
ul.appendChild(liCarlos);

// Añadir h1 y ul al cuerpo del documento
const agenda = document.getElementById('agenda')
agenda.appendChild(h1);
agenda.appendChild(ul);
```

Es evidentemente mas complicado crear la 'agenda' usando JavaScript que usando HTML pero creando una función auxiliar en JavaScript puro podemos simplificar el proceso.

```
function createElement(tag, props, ...children) {
        const element = document.createElement(tag);
        Object.assign(element, props)
        for(let child in children) {
                if (typeof child === 'string') {
                        element.appendChild(document.createTextNode(child));
                } else {
                        element.appendChild(child)
                }
        }
        return element
}
```

La creación de la 'agenda' ahora quedaría de la siguiente manera.

```
const agenda =
        createElement('div', {},
                createElement('h1', {}, 'Agenda'),
                createElement('ul', {},
                        createElement('li', {}, 'Beatriz'),
                        createElement('li', {}, 'Alejandro'),
                        createElement('li', {}, 'Carlos')
                )
        )

const root = document.getElementById('agenda')
root.replaceChildren(agenda)
```

Este mismo código puede ser generado usando React, en particular usando una forma mas simple de crear los elementos.

```
import React from 'react' // Importa desde React
import ReactDOM from 'react-dom'

// Generando la 'agenda' usando ReactJS puro
const agenda =
        React.createElement('div', {},
                React.createElement('h1', {}, 'Agenda'),
                React.createElement( 'ul', {},
                        React.createElement('li', {}, 'Alejandro'),
                        React.createElement('li', {}, 'Beatriz'),
                        React.createElement('li', {}, 'Carlos')
                )
        )

ReactDOM.render(agenda, document.getElementById('agenda'));
```

Este código es mas simple que crear las páginas con JavaScript puro pero es menos claro y mas extenso que usar HTML. Es por eso que se creo un ampliación del lenguaje que permite escribir código símil HTML dentro de JavaScript: Los archivos JSX.
Este tipo de archivo no lo entiende los navegadores por lo que para ser usado debe ser convertidos a código React antes de ser usado.

```
/// Genera la 'agenda' usando ReactJS mediante JSX
const agenda = (
    <div>
        <h1>Agenda</h1>
        <ul>
            <li>Beatriz</li>
            <li>Alejandro</li>
            <li>Carlos</li>
        </ul>
    </div>
)

ReactDOM.render(agenda, document.getElementById('agenda'));
```

Para poder usar JSX en el navegador se debe usar una herramienta adicional a React: Babel.
Una aplicación compleja requiere la creación de un proyecto que configura e instala la herramientas que facilita el desarrollo en React, pero realizar prototipos o pequeñas aplicaciones se puede configurar una página para que funcione autónomamente.

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <!-- Importar librerias React, ReactDOM y Babel -->
    <script src="https://unpkg.com/react@18/umd/react.development.js"></script>
    <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js">
</script>
    <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
</head>
<body>
    <div id="root"></div>
    <script scr="main.jsx" type="text/babel"></script>
</body>
</html>
```

El código se puede ingresar en 'main.jsx' o bien dentro de un

```
// file: main.jsx
const App = () => // Ejemplo de JSX
        <div>
            <h1>¡Hola, mundo!</h1>
            <p>¡Bienvenidos a React!</p>
        <div/>

const root = ReactDOM.createRoot(document.getElementById('root'))
```

```
root.render(<App />);
```

Hacer una aplicación en React consiste en hacer componentes que se use para crear otros componentes, que forman páginas que forman sitios. Los componentes encapsulan el comportamiento.

Hacer un componente en React es definir una función que recibe un objeto como argumento y retornar una representación en pantalla (un trozo de DOM). Normalmente esa representación se expresa en JSX.

JSX permite usar dentro de JavaScript expresiones que tiene el formato similar a HTML.
En cualquier lugar que pueda ingresar una expresión puede poner una expresión JSX, puede ser asignado a una variable, ser retornado por una función o ser pasado como parámetro a otros componentes.

Cuando se encuentra una expresión JSX los elementos en minúsculas generan un componente HTML, cuando son en mayúsculas crean componente en definidos en React.

Con los atributos que hay dentro de un elemento se conforma un objeto que es pasado como argumento al componente invocado. Cuando se encuentra llaves dentro de un JSX en el interior debe ir una expresión de JavaScript, esta expresión es evaluada en el momento de la creación del componente.

JSX parece HTML pero no lo es, usa una sintaxis similar pero tiene varias diferencias:

1. Siempre debe estar bien formado. (Todos los elementos deben ser cerrados)
2. Los atributos algunas veces son diferentes a HTML (en particular en lugar de 'class' se debe usar 'className', 'for' debe ser 'htmlFor' y los eventos se deben escribir usando camelCase, en lugar de 'onclick' se debe escribir 'onClick')
3. Siempre debe conformar una jerarquía estricta si debe retornar mas de un elemento deben ser colocado dentro de <> </> (Conocido como un fragmento)
4. Pasa pasar un estilo a un componente se le puede pasar un objeto con reglas de css. En este caso las clave de los objetos se deben expresar en camelCase en lugar de palabras separadas por guiones, 'font-size' se debe escribir 'fontSize' y los valores siempre van entre comillas.

Los componentes retorna un trozo de pantalla.

```
function Titulo(){
        return <h1>Demo React con JSX</h1>
}

// Se usa <Titulo />
```

Pueden recibir un objeto con las propiedad

```
function Saludo(props){
        return <div>Hola {props.nombre}!</div>
}

// Se usa <Saludo nombre="Juan" />
```

Se puede usar la desestructuración para facilitar el uso.

```
function Campo({etiqueta, valor}){
        const estilo = { color: 'gray', fontSize: '14px' }
        return (
        <>
                <label style={estilo}>{etiqueta} htmlFor="{etiqueta}</label>
                <input type="text" value={valor} name={etiqueta}/>
        </>
        )
}
// se usa <Campo etiqueta="Nombre" valor="Juan" />
```

También puede recibir otros componentes anidados en la propiedad 'children'

```
function Formulario({titulo, children}){
        return (
        <form className="editar">
                <h3>{titulo}</h3>
                {children}
        </form>
        )
}

function EditarContacto({nombre, apellido, telefono}){
        return (
        <Formulario titulo="Editar Contacto">
                <Campo etiqueta="Nombre" valor={nombre} />
                <Campo etiqueta="Apellido" valor={apellido} />
                <Campo etiqueta="Teléfono" valor={telefono} />
        </Formulario>
        )
}
```

Al final todo código de JSX se traduce a JavaScript

```
function App(){ // Ejemplo de JSX
        return (
        <div>
            {Titulo()}
            <Saludo nombre="Juan" />
                <EditarContacto nombre="Juan" apellido="Perez" telefono="5343458"
/>
        </div>
        )
```

```
}
// -- JSX se convierte en JavaScript --
//  React.createElement('div', {},
//      Titulo(),
//      Saludo({nombre: 'Juan'}),
//      EditarContacto({nombre: 'Juan', apellido='Perez' telefono='5343458'})
//  )
```

Los componentes son funciones normales, en ello se puede ejecutar cualquier funcionalidad necesaria, por ejemplo se podría generar el resultado en forma condicional.

```

// Usando expresion booleana
function Producto({nombre, precio, esOferta}){
        return (
        <div>
                {nombre} <b>${precio}</b>
                {esOferta && <span style={{color:'red'}}>30% descuento</span>}
        </div>
        )
}

// Usnado operador ternario
function Favorito({marcado}){
        return (
        <span>{marcado ? <IconoMarcado /> : <InonoNormal />}</span>
        )
}
```

Los componente pueden mostrar una lista de elementos, para ello debe identificar cada elemento con una clave única. Se pueden pasar todas las propiedad de un objeto se lo pone lo pone {...objeto} (tres puntos entre llaves)

```
function Contacto( {nombre, telefono}){
        return <li></li>
}

function Agenda({datos}){
        return (
                <ul>
                {datos.map(c => <Contacto key = {c.id} { ...c} />)}
                </ul>
        )
}

let contactos = [
        {id: 1, nombre:'Jose',  telefono: '400-1234'},
        {id: 2, nombre:'Maria', telefono: '400-1235'},
        {id: 3, nombre:'Juan',  telefono: '400-1236'},
]

<Agenda datos={contactos} />
```

Hasta el momento todos los componentes que creamos fueron componentes pasivos, solo mostraban información que recibían del padre. Pero se pueden hacer componentes interactivos que en los que el padre reciba información producida en el hijo, para ello se debe definir que el componte pueda emitir un evento del usuario el cual puede traer la información generada.

Para implementar un componente que emita eventos se le debe pasar una función (un callback) como parámetro que será ejecutada cuando el usuario interactúe con el componente.

El nombre de la función debe (por convención) comenzar con el prefijo 'on' para identificar un evento: p. e 'onEdit', 'onCancel', etc

El componente en este caso deberá escuchar un evento que produzca el usuario y eventualmente emitir un evento (llamar a la función pasada) con la información necesaria.

Por ejemplo podría hacer un componente que confirme una operación, el mismo debería emitir un evento indicando si la operación fue aceptada o cancelada. En esté caso en particular el componente escucha dos posibles eventos del usuario, hacer click en el botón aceptar o en el botón cancelar. Dependiente del caso emitirá (llamará a la función con el valor a devolver) el evento 'onConfirm' con el valor true o false respectivamente.

```
function Confirmar({mensaje, onConfirm}){
        const aceptar  = (e) => onConfirm(true)
        const cancelar = (e) => onConfirm(false)
        return (
                <div>
                        <h2>Confirmar</h2>
                        <p className="pregunta">{mensaje}</p>
                        <div className="respuestas">
                                <Boton texto="Aceptar"  onClick={aceptar} />
                                <Boton texto="Cancelar" principal={false} onClick=
{cancelar} />
```

```
                        </div>
                </div>
        )
}

function Boton({texto, principal = true,  ...props}){
    return (
        <button className={`btn btn-${principal ? 'primary' : 'secondary'}`}
                { ...props} >
            {texto}
        </button>
    );
}

function informar(acepto){
        if(acepto) {
                alert('La operación fue exitosa')
        } else {
                alert('La operación no fue aceptada')
        }
}

<Confirmar mensaje="¿Está seguro que quiere comprar?" onConfirm={informar} />
```

Note que en el caso del componente Botón se pasa dos propiedades especiales: en principal se la da un valor por defecto, en *...props* se indica que todas las propiedades extras (además de texto y principal) serán pasadas al componente *Button* mediante *{...props}*

Componentes con estados

Hay veces que se necesita que los componentes pueda realizar alguna operación en respuesta a las acciones del usuario. Esta acciones deben producir cambios en el estado de la aplicación por los que los componente deben tener un mecanismo para preservar la información generada al tiempo que le debe informar a React que al haberse producido un cambio de estado se debe actualizar la parte de la pantalla afectada.

Si bien los componentes son funciones y pueden usar variables internamente las misma no son de utilidad porque cada vez que la función es llamada para ser mostrada en pantalla las variables se vuelven a inicializar nuevamente. Se podría usar variables globales para persistir los cambios pero aun en ese caso no seria de utilidad porque React no tienen forma de saber que una variable ha sido modificada. Para resolver estos problema se han creado los hooks o ganchos. Los hooks son funciones de librería que permite hacer componentes que reserven el estado ejecuten acciones secundaria, etc.

El primer hooks es *useState*. Esta función al ser llamada por primera vez inicializa el valor de estado, la subsiguiente veces recupera el valor previamente registrado. A la vez la función retorna dos valores al ser llamada, el primer valor consiste de una variable con el contenido del estado, esta variable se debe considerar de *solo lectura* por no que no debe ser modificada directamente, el segundo valor es una función que se debe usar para cambiar el estado que se vera reflejando en la próxima vez que

dibuje la pantalla. Esta función se encarga además de verificar que el estado cambio en cuyo caso va a pedirte a React que vuelva a dibujar la pantalla para reflejar los campos.

```
function Contador({inicial:0}){
        const [contador, setContador] = useState(inicial)

        let incrementar = () => setContador( contador + 1)
        let decrementar = () => setContador( contador - 1)

        return <>
                <button onClick={decrementar}>-1</button>
                <span>Contador: {contador}<span>
                <button onClick={incrementar}>+1</button>
        </>
}

function App(){
        return <>
                <h1>Contando Votos</h1>
                <div>Candidato 1 <Contador /></div>
                <div>Candidato 2 <Contador /></div>
                <div>Candidato 3 <Contador /></div>
        </>
}
```

El nombre que se le pone al estado puede ser cualquiera pero es convención que la función para cambiar el estado sea el nombre de la variable precedido por *set*, si queremos guardar el estado de un teléfono, por ejemplo deberíamos usar [telefono, setTelefono]

Establecer el valor de un estado no significa necesariamente que el estado cambie, React va a inferir si el estado cambio y por lo tanto es necesario refrescar la pantalla comparando el nuevo estado con el estado original, de hay la importancia que el estado devuelto por *useState* no se modifique nunca. Cuando el estado corresponde a un tipo básico (Number, String o Boolean) esta condición, la de trabajar el estado como inmutable es automática. En cambio cuando tra
bajamos con tipos compuestos como array u objetos se de trabajar sobre una copia, es decir se deben usar el patrón inmutable para registrar los cambios.

Veamos por ejemplo cuando se usa el estado para preservar un tipo complejo como un objeto.

```
function DetalleCompra({ compra }) {
        const [item, setItem] = useState(compra)

        function agregar() {
                let copia = { ...item, cantidad: item.cantidad + 1}
                setItem(copia)
        }

        const importe = item.cantidad * item.precio
        return (
                <div>
```

```
                        {item.cantidad}u {item.nombre} a ${item.precio} =
    ${importe}
                        <button onClick={agregar}>+</button>
                </div>
        )
}
```

La forma mas simple de actualizar un objeto de forma inmutable es usar el operador spread ('...') para copiar los datos del objeto original y poner la propiedad que se quiere modificar a continuación.

Cuando queremos actualizar varias propiedades de un objeto simultáneamente se puede usar el patrón {...original, ...cambio} supongamos que tenemos un producto completo y queremos modificar solo algunas propiedades...

```
const producto = { id: 1, nombre: 'Coca Cola', precio: 200, cantidad: 40 }
const compra = { cantidad: 10, precio: 250 }

// Cambiar varias propiedades
const nuevo = { ...producto, ...compra}
//> nuevo == {id: 1, nombre: 'Coca Cola', precio: 250, cantidad: 10}

// Cambiar propiedades individuales
const nuevoPrecio = { ...producto, precio: 300 }
//> nuevoPrecio == {id: 1, nombre: 'Coca Cola', precio: 300, cantidad: 10}
```

Cuando trabajamos con array las operaciones de agregar, borrar o modificar elementos se cambian:

```
const lista = ['Alejandro', 'Beatriz', 'Carlos']

/// -- Agregar datos -- ///

// Agregamos 'David' al final
const agregarFinal = [...lista, 'David']
//> ['Alejandro', 'Beatriz', 'Carlos', 'David']

// Agregamos 'David' al inicio
const agregarInicio = ['David', ...lista]
//> ['David', 'Alejandro', 'Beatriz', 'Carlos']

/// -- Borrando datos -- ///

// Borrar la posición 2
const borrado2 = lista.filter( (item, i) => i != 2)
//> ['Alejandro', 'Beatriz']

// Borrar el elemento 'Beatriz'
const borradoBeatriz = lista.filter((item) => item != 'Beatriz')
// > ['Alejandro', 'Carlos']

/// -- Modificar datos ---

// Modificar el elemento 'Beatriz' por 'Belen'
```

```
const modificado = lista.map( (item) => item == 'Beatriz' ? 'Belen' : item)
// > ['Alejandro', 'Belen', 'Carlos']
```

Un uso normal de este patrón es cuando tenemos un array de objetos y los objetos tienen un campo que actúa como identificador.
La ABM sobre este tipo de objeto se suele implementar de la siguiente manera:

```
// Datos originales
const productos = [
        {id: 1, nombre: 'Coca Cola', cantidad: 1, precio: 100},
        {id: 2, nombre: 'Pepsi', cantidad: 3, precio: 90},
        {id: 3, nombre: 'Fanta', cantidad: 2, precio: 80},
        {id: 4, nombre: 'Sprite', cantidad: 4, precio: 70}
]

// Alta
const nuevo = {id: 5, nombre: '7up', precio: 60, cantidad: 5}
const productoAgregado = [ ...productos, nuevo]

// Baja
const borrar = {id: 3, /**/}
const productoEliminado = productos.filter(p => p.id !== borrar.id)

// Modificación (solo nombre y precio)
const cambios = {id: 2, nombre: 'Pepsi Cola', precio: 95}
const productoModificado = productos.map(p =>
                                                    p.id === cambios.id
? {...p,  ...cambios} : p)
```

Por ejemplo, al implementar un carrito de compra podríamos tener las operaciones implementada con este patrón: (nota: La alta esta simulada por simplicidad)

```
const { useState } = React
const { createRoot } = ReactDOM

const DatosIniciales = [
        { id: 1, nombre: 'Coca Cola', precio: 100, cantidad: 1 },
        { id: 2, nombre: 'Pepsi', precio: 90, cantidad: 3 },
        { id: 3, nombre: 'Fanta', precio: 80, cantidad: 2 },
        { id: 4, nombre: 'Sprite', precio: 70, cantidad: 4 }
]

function moneda(valor) {
        const opciones = {style: 'currency', currency: 'ARS'}
        return new Intl.NumberFormat('es-AR', opciones).format(valor)
}

function sumar(valores) {
        return valores.reduce((suma, valor) => suma + valor, 0)
}

function Boton({ texto, grande = false, onClick }) {
```

```
        return (
                <button
                        className={grande ? 'grande' : ''}
                        onClick={onClick}>{texto}
                </button>
        )
}

function Totales({ productos }) {
        let unidades = sumar(productos.map(p => p.cantidad))
        let total = sumar(productos.map(p => + p.precio * p.cantidad))

        if(unidades === 0) return null

        return (
                <>
                        <br/>
                        {unidades} {unidades == 1 ? 'unidad' : 'unidades'}
                        <div className="destacar">Total: {moneda(total)}</div>
                </>
        )
}

function Producto({ producto, alSumar, alRestar }) {
        const { id, nombre, cantidad, precio } = producto
        return (
                <li>
                        <span>
                                <div className='destacar'>{nombre}</div>
                                {cantidad}u x {moneda(precio)} =
                                <b>{moneda(precio * cantidad)}</b>
                        </span>
                        <span>
                                <Boton texto='Agregar' onClick={() => alSumar(id)}
/>
                                <br />
                                <Boton texto='Quitar' onClick={() => alRestar(id)}
/>
                        </span>
                </li>
        )
}

function App() {
        const [productos, setProductos] = useState(DatosIniciales)

        // Funciones utilitarias
        const proximoID = () => Math.max( ...productos.map(p => p.id)) + 1
        const buscarProducto = (id) => productos.find(p => p.id === id)

        // Manejo de eventos
        const agregar = () => { // Alta
                const id = proximoID()
```

```
                const nuevo = { id, nombre: '7up', precio: 60, cantidad: 1 }

                setProductos([ ...productos, nuevo])
        }

        const borrar = (id) => { // Baja
                setProductos(productos.filter(p => p.id !== id))
        }

        const sumar = (id) => { // Modificación
                const cantidad = buscarProducto(id).cantidad + 1
                setProductos(productos.map(
                        p => p.id === id ? {  ...p, cantidad } : p))
        }

        const restar = (id) => { // Modificación
                const cantidad = buscarProducto(id).cantidad - 1
                if (cantidad > 0) {
                        setProductos(productos.map(
                                p => p.id === id ? {  ...p, cantidad } : p))
                } else {
                        borrar(id)
                }
        }

        return (
                <div id="main">
                        <h2>Productos</h2>
                        <ul>
                                <Boton texto='Nuevo Producto'
                                        onClick={agregar} grande />
                                {productos.map(p =>
                                        <Producto key={p.id}
                                                producto={p}
                                                alSumar={sumar}
                                                alRestar={restar} />)}
                                <Totales productos={productos} />
                        </ul>
                </div>
        )
}

const root = createRoot(document.getElementById('root'))
root.render(<App />);
```

Otro ejemplo de uso de estado es cuando queremos editar datos con un formulario.

Los datos a editar se deben guardar en un estado, se debe pasar el estado en la propiedad value del elemento input y actualizando el valor de la variable en el evento onChange

```
const [nombre, setNombre] = useState('')
//...
const campo = <input type="text"
```

```
                value={nombre}
                                        onChange={e => setNombre(e.target.value)}
/>
```

Aplicado a, por ejemplo, un formulario de *login* nos queda algo así como:

```
const Login = ({ onLogin }) => {
        const [usuario, setUsuario] = useState('')
        const [contraseña, setContraseña] = useState('')
        const [error, setError] = useState(false)

        // Simula un usuario
        const contraseñaValida = () =>
                usuario === 'admin' && contraseña === '1234'

        const cambiarUsuario = (e) => {
                setUsuario(e.target.value)
                setError(false)
        }

        const cambiarContraseña = (e) => {
                setContraseña(e.target.value)
                setError(false)
        }

        const ingresar = (e) => {
                if (contraseñaValida()) {
                        onLogin(usuario, contraseña)
                } else {
                        setError(true)
                }
                e.preventDefault()
        }

        return (
                <form>
                        <h3>Iniciar Sesión</h3>

                        <label>Usuario:</label>
                        <input type="text"
                                value={usuario}
                                onChange={cambiarUsuario} autofocus />

                        <label>Contraseña:</label>
                        <input type="password"
                                value={contraseña}
                                onChange={cambiarContraseña} />

                        <div className="error">
                                {error && "La contraseña y/o el usuario son
incorrectos"}
                        </div>

```

```
                        <div className="acciones">
                                <button onClick={ingresar}>Ingresar</button>
                        </div>
                </form>
        )
}

const App = () => {
        const [usuario, setUsuario] = useState('')
        const onLogin = (usuario, contraseña) => setUsuario(usuario)

    return usuario ?
                            <h2>Bienvenido {usuario}</h2> :
                            <Login onLogin={onLogin} />
        }
```