UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Centro de e-Learning

Diplomatura en programación web full stack con React JS

Centro de e-Learning SCEU UTN - BA.
Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

Módulo 5: React Inicial

Unidad 2: Componentes, estados y eventos

Presentación:

Los componentes son las estructuras base de las aplicaciones React, los estados y eventos las formas de comunicación entre los componentes. En esta unidad vamos a aprender acerca de estos conceptos fundamentales!

Objetivos:

Que los participantes:

- Comprendan los conceptos de componente, estado y evento.
- Sean capaces de implementar dichos conceptos.

Bloques temáticos:

1. Concepto de componente
2. Creación de componentes
3. Comunicación en entre componentes
4. Estados
5. Eventos
6. Ejemplo
7. Trabajo Práctico

Consignas para el aprendizaje colaborativo

En esta Unidad los participantes se encontrarán con diferentes tipos de actividades que, en el marco de los fundamentos del MEC*, los referenciarán a tres comunidades de aprendizaje, que pondremos en funcionamiento en esta instancia de formación, a los efectos de aprovecharlas pedagógicamente:

- Los foros proactivos asociados a cada una de las unidades.
- La Web 2.0.
- Los contextos de desempeño de los participantes.

Es importante que todos los participantes realicen algunas de las actividades sugeridas y compartan en los foros los resultados obtenidos.

Además, también se propondrán reflexiones, notas especiales y vinculaciones a bibliografía y sitios web.

El carácter constructivista y colaborativo del MEC nos exige que todas las actividades realizadas por los participantes sean compartidas en los foros.

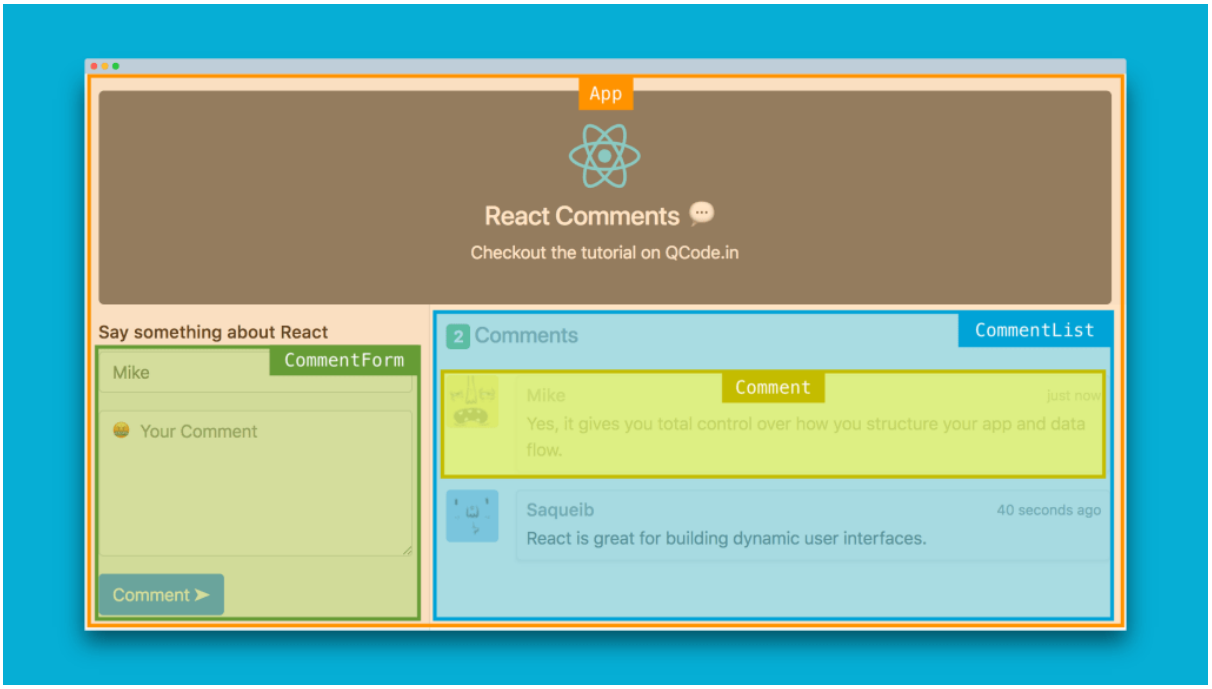
**** El MEC es el modelo de E-learning colaborativo de nuestro Centro.***

Tomen nota:

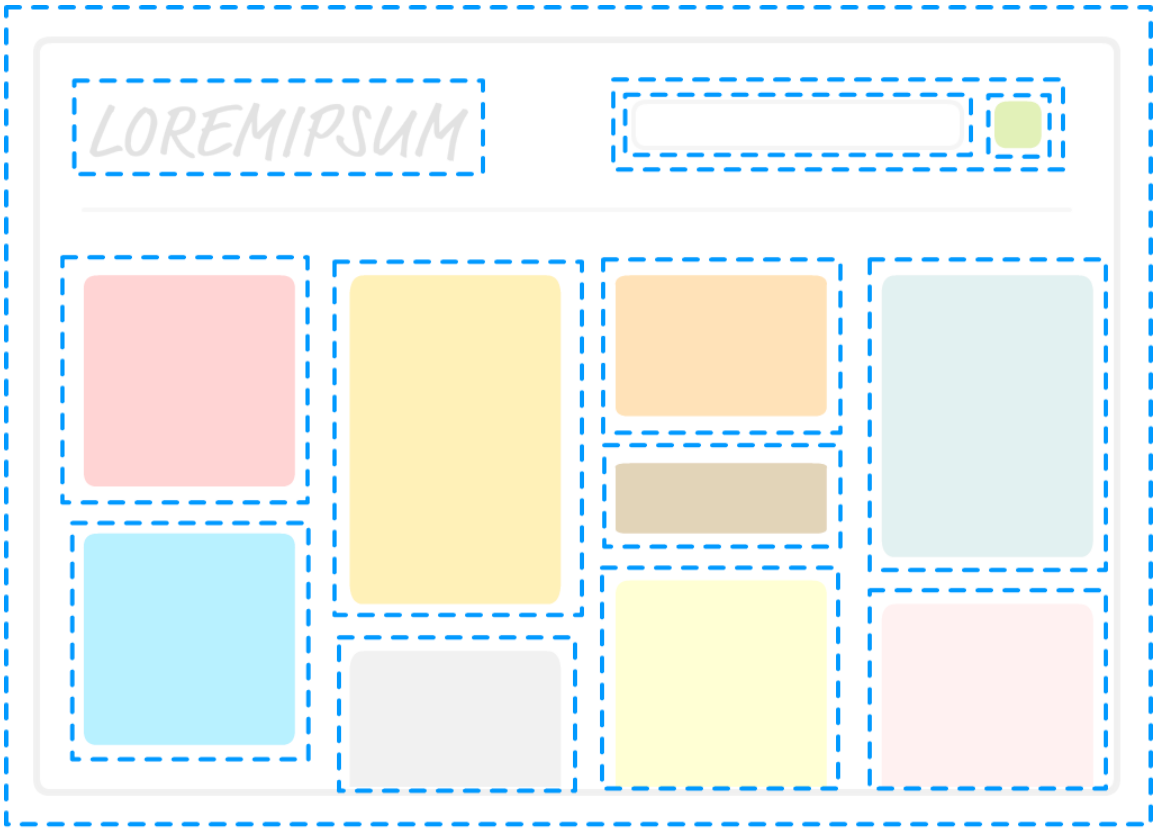
Las actividades son opcionales y pueden realizarse en forma individual, pero siempre es deseable que se las realice en equipo, con la finalidad de estimular y favorecer el trabajo colaborativo y el aprendizaje entre pares. Tenga en cuenta que, si bien las actividades son opcionales, su realización es de vital importancia para el logro de los objetivos de aprendizaje de esta instancia de formación. Si su tiempo no le permite realizar todas las actividades, por lo menos realice alguna, es fundamental que lo haga. Si cada uno de los participantes realiza alguna, el foro, que es una instancia clave en este tipo de cursos, tendrá una actividad muy enriquecedora.

Asimismo, también tengan en cuenta cuando trabajen en la Web, que en ella hay de todo, cosas excelentes, muy buenas, buenas, regulares, malas y muy malas. Por eso, es necesario aplicar filtros críticos para que las investigaciones y búsquedas se encaminen a la excelencia. Si tienen dudas con alguno de los datos recolectados, no dejen de consultar al profesor-tutor. También aprovechen en el foro proactivo las opiniones de sus compañeros de curso y colegas.

1. Concepto de componente



Son fragmentos visuales de la página web



That's a lot of COMPONENTS!

Si, son muchos componentes, pero que hacen operaciones muy específicas:
- Mostrar un formulario
- Mostrar un comentario
- Recorrer una lista de comentarios y llamar al componente de Mostrar un comentario

…

.

2. Creación de componentes

Al igual que en NodeJS, utilizamos el Visual Studio Code.

Paso 1

Creamos un archivo con el nombre del componente y como extensión .js

Ejemplo

primerComponente.jsx

Paso 2

Escribimos el código del componente. A este punto hay que aclarar que ReactJS es muy versátil y permite armar los componentes basados en funciones o basados en clases (paradigma de objetos).

En este ejemplo utilizamos funciones que es la forma más nueva de trabajar React

```
import './App.css';
import React from 'react';

export default function PrimerComponente() {
  const name = 'Juan Perez';
  const element = <h1>Hola {name}</h1>;
  return element;
}
```

Realizamos las importaciones necesarias.

Escribimos el código teniendo en cuenta que se debe tener obligatoriamente un return que retorne el JSX que se debe mostrar.

En este otro ejemplo utilizaremos una función que incluye el componente que acabamos de crear.

```
import PrimerComponente from './primerComponente';
export default function App() {
  return (
    <div className="App">
      <PrimerComponente  />
    </div>
  )
}
```

Se considera que el archivo “primerComponente.jsx” se encuentra en la misma carpeta que App.js

Este es el componente principal (el que se ejecuta inicialmente) donde podemos ver que sigue la forma del componente anterior: importación y luego en este caso una función que incluye un return dónde va el código a mostrar.

Se puede visualizar la forma en que se llama a un componente desde otro dentro del return de esta función. Cada componente importado se incluye como una etiqueta.

<PrimerComponente />

La barra detrás del nombre del componente cierra la etiqueta.

Return de la función

Caso especial - Sin elemento contenedor

Recordemos que el “return” de la función siempre debe retornar el JSX que se genera el componente. Hasta ahora vimos que para retornar el JSX debíamos incluirlo necesariamente en una etiqueta HTML, pero en algunos casos podemos no querer agregar etiquetas extras (ya que pueden no ser necesarias para el componente). En estos casos podemos hacer uso de una etiqueta particular: React.Fragment (envolviendo nuestro código en esta etiqueta, la cual es solo a los fines de “agrupar” la respuesta, y no generará ninguna etiqueta HTML en particular)

Veamos un ejemplo del método render con esta etiqueta.

```
return (
  <React.Fragment>
    <td>Hello</td>
    <td>World</td>
  </React.Fragment>
);
```

Existe también una versión “resumida” de la etiqueta que podemos utilizar, la cual podemos reemplazar React.Fragment por una etiqueta de apertura y cierre (sin contenido)

```
return (
  <>
    <td>Hello</td>
    <td>World</td>
  </>
);
```

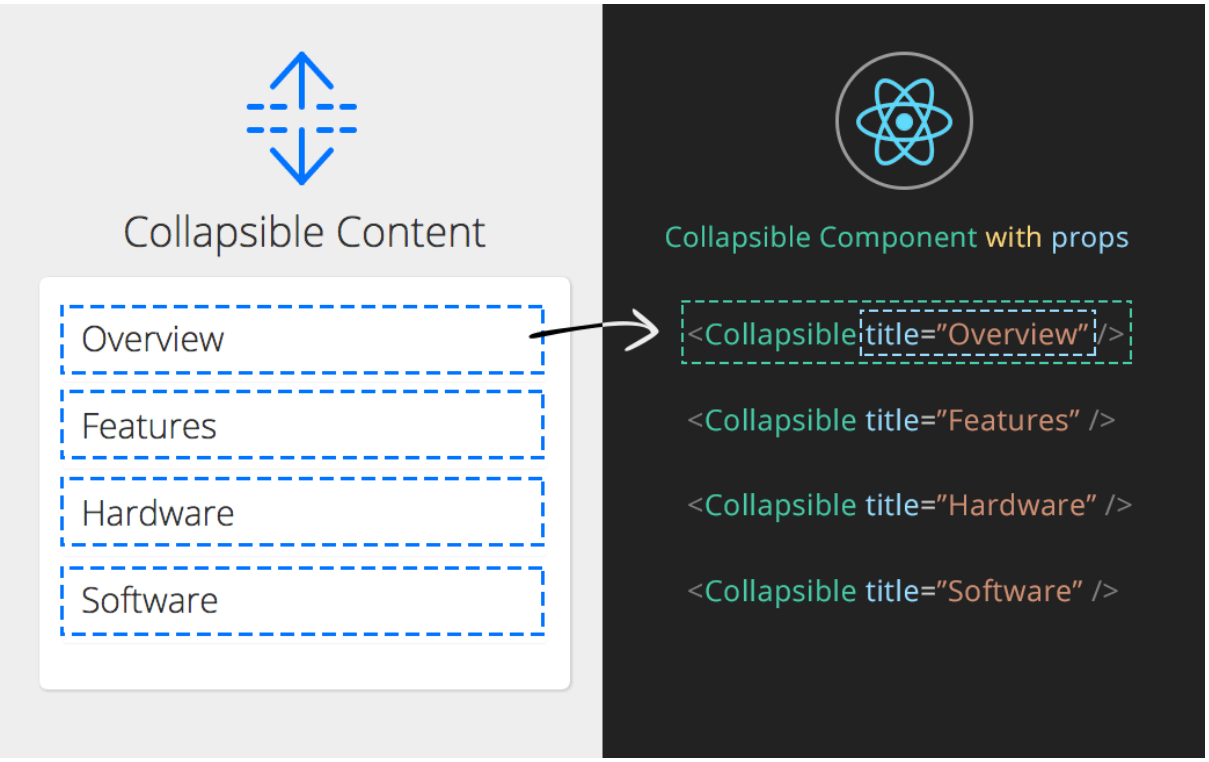
3. Comunicación en entre componentes

Paso de parámetros a componentes

Hasta el momento trabajamos con componentes que siempre muestran el mismo valor, independientemente de su entorno. A partir de ahora comenzaremos a trabajar con componentes que podamos cambiar su comportamiento, similar a los parámetros de las funciones, pero orientado a la visualización y comportamiento de cada componente.

Tomemos como ejemplo la siguiente imagen que muestra del lado izquierdo como se ve el componente Collapsible, y del lado derecho como llamamos a dicho componente para poder cambiar la información que muestra.

Ejemplo



Veamos un ejemplo completo de paso de parámetros a un componente. Comencemos con la llamada a un componente llamado **PrimerComponente** cuya función es mostrar el nombre que le pasamos como parámetro.

La llamada al componente puede ser de la siguiente forma

```
<PrimerComponente name="Juancito" />
```

Como vemos, el paso de un parámetro al componente se realiza por medio del agregado de un atributo “HTML” en la etiqueta del componente que hemos definido.

Luego en el componente, podemos acceder al parámetro por medio de **props.<nombre del parámetro>**. Siendo “props” el parámetro que recibe la función (el nombre es una convención). Veamos el ejemplo

```
import React from 'react';

export default function PrimerComponente(props) {
  return (
    <h1>Hola {props.name}</h1>
  )
}
```

El parámetro (atributo) es **name**, por lo que para poder acceder a dicho parámetro hacemos uso de **props.name**.

Nada nos impide de llamar al componente varias veces en nuestra aplicación, por ejemplo de la siguiente forma

```
  return (
    <div className="App">
      <PrimerComponente name="Juan Carlos" />
      <PrimerComponente name="Maria" />
      <PrimerComponente name="Victoria" />
    </div>
  )
```

De esta forma, se llamará al componente 3 veces, la primera vez el parámetro **name** tendrá como valor **Juan Carlos**, la segunda vez **Maria**, y por último el valor **Victoria**.

Esto provocará que el componente se muestre 3 veces en la aplicación.

Uso de vectores

De manera habitual, no llamaremos a los componentes con datos fijos, sino que utilizaremos información que hemos guardado previamente en un vector.

Supongamos que tenemos el siguiente vector de nombres

```
const vectorNombres = [
    'Juan Carlos',
    'Maria',
    'Gerardo'
];
```

Y que deseamos llamar al componente que muestra los nombres (PrimerComponente) para cada uno de los elementos del vector (para mostrar todos los valores).

Disponemos de varias formas de trabajar con este tipo de vectores en JavaScript y React. Presentaremos la forma más popular de realizar esta operación que es mediante el uso del método map que tienen todos los vectores. El método map, permite convertir los elementos de un vector, a otro formato, para el ejemplo que estamos realizando, convertiremos los elementos del vector en componentes React, más precisamente en componentes PrimerComponente, y cada uno de estos componentes recibirán como parámetro el nombre de la persona que debe mostrar.

```
const vectorNombres = [
    'Juan Carlos',
    'Maria',
    'Gerardo'
];

const respuesta = vectorNombres.map(unItem => {
    return <PrimerComponente name={unItem} />
})

return (
    <div className="App">
        {respuesta}
    </div>
)
```

Analicemos el código:

- Recorreremos vectorNombres y por cada elemento
  - Convertiremos el nombre al componente PrimerComponente pasándole como parámetro el nombre de la persona (elemento del vector)
  - Cada elemento del vector nuevo (vector de PrimerComponente) se almacena en la variable respuesta
- Mostramos la variable respuesta (el vector de componentes)

4. Estados

Manejo de estados en la aplicación

En los casos que mostramos una propiedad del componente (el return del componente) el mismo se ejecuta una vez de manera predeterminada, al momento de tener que mostrar el componente. En los casos que debemos modificar la propiedad luego de mostrarse el componente, este cambio no se verá reflejado en la interfaz (ya que el return ya fue llamado).

Para poder reflejar el cambio de una propiedad del componente, debemos hacer uso del **estado**. El estado es una propiedad especial dentro de los componentes React, las propiedades que figuran en el estado, son aquellas que al ser modificadas fuerzan una actualización de la interfaz. Para poder hacer uso del estado en React debemos utilizar una función especial **useState**. Debemos utilizar esta función para todas aquellas propiedades que deseamos que al momento de ser modificadas, provoquen un cambio en la interfaz.

Veamos un ejemplo de la propiedad state, que dentro tiene una propiedad fecha

```
const [fecha, setFecha] = React.useState(new Date())
```

Al momento de crearse el componente, la variable fecha (que está dentro de state) toma como valor la fecha actual de la computadora (new Date()).

La llamada a la función **useState** retorna un vector de 2 posiciones, en la primera posición la variable de la cual vamos a “leer” el estado, y la segunda posición una función que nos permite modificar el estado. El nombre de la variable y función pueden ser las que el programador prefiera, aunque por convención se utiliza nombreDeVariable, setNombreDeVariable.

La propiedad fecha, por estar dentro de state automáticamente es una propiedad que si cambiamos su valor, generará una actualización de la interfaz (se vuelve a ejecutar la función). Pero la forma de realizar el cambio del estado se realiza por medio de la llamada a un método especial **setFecha()**. Si nosotros deseamos actualizar la fecha y que la misma se vea reflejada en la interfaz, debemos utilizar el siguiente comando

```
setFecha(new Date())
```

La función setFecha recibe como parámetro cualquier tipo válido en JavaScript (Objeto, Vector, entero, string, etc), con las propiedades que deseamos cambiar.

En React, todo aquello que implique una actualización de la interfaz, debe estar definido por medio de la llamada a useState.

Se puede tener un único useState por función, y dentro una estructura con todos los elementos, o tener varios useState por función con estructuras más sencillas.

Opción 1

```
const [estado, setEstado] = React.useState({
    persona: {
        nombre: 'Pilar',
        apellido: 'Geijo'
    },
    tareas: [
        {id: 1, nombre: 'Terminar unidad', done: false},
        {id: 2, nombre: 'Hacer TP', done: false},
    ]
})
```

Opción 2

```
const [persona, setPersona] = React.useState({
    nombre: 'Pilar',
    apellido: 'Geijo'
})

const [tareas, setTareas] = React.useState([
    {id: 1, nombre: 'Terminar unidad', done: false},
    {id: 2, nombre: 'Hacer TP', done: false},
])
```

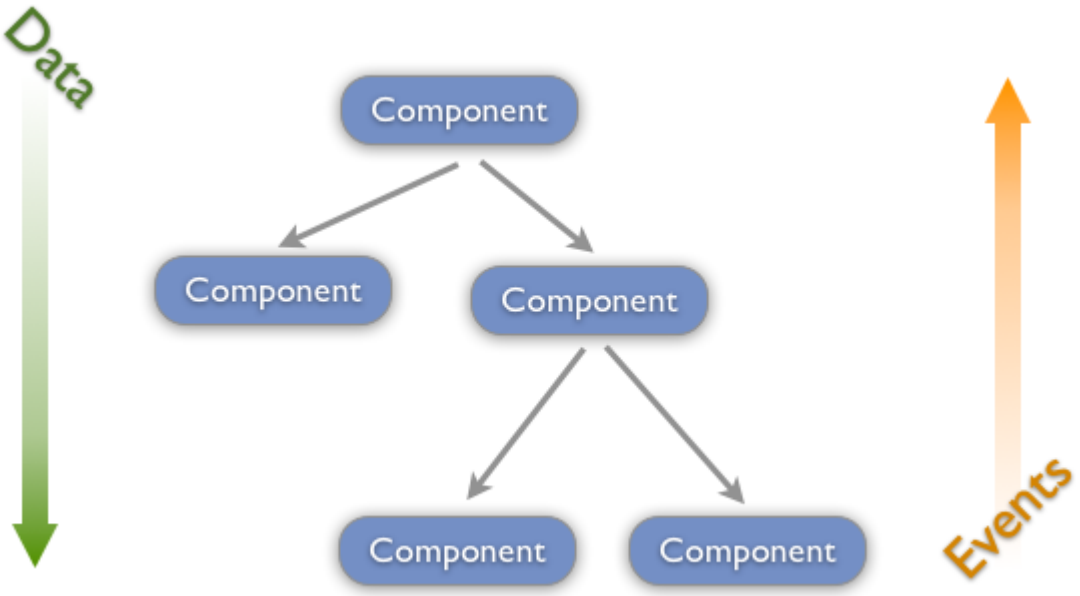
En React se acostumbra a utilizar la opción 2.

5. Eventos

Manejo de eventos en la aplicación

Cuando debemos responder ante acciones del usuario, como la modificación de un campo en un formulario, envío de un formulario, click en un botón, etc. debemos hacer uso de los eventos que nos provee JavaScript.

En el siguiente gráfico podemos apreciar que los datos pueden ser pasados de un componente padre a un componente hijo. Mientras que los eventos pueden ser enviados de un componente hijo a un componente padre.



Supongamos que deseamos mostrar un mensaje cuando el usuario realiza click sobre un botón, para ello podemos utilizar el siguiente código

```
const onPresionoBoton = () => {
    console.log('Presiono botón')
}

return (
    <div>
        <button onClick={onPresionoBoton}>Soy un botón</button>
    </div>
)
```

Veamos el flujo de operaciones en el código:

- Se ejecuta el return el cual muestra un botón para que el usuario pueda hacer click sobre el mismo.
- Cuando el usuario hace click sobre el botón (evento)
    - Se ejecuta la acción especificada en el atributo onClick, en este caso es llamar a la función onPresionoBoton (nótese que no llevan paréntesis en React, es decir, la llamada no es onPresionoBoton(). En caso de poner paréntesis en atributo onClick, el método onPresionoBoton se ejecutará al momento de carga de la interfaz (y nosotros deseamos que solo se ejecute cuando el usuario hace click sobre el mismo). Estamos pasando al evento onClick una callback (una función que se ejecutará cuando el usuario haga click)
        - La función onPresionoBoton() solamente muestra el mensaje 'Presiono botón'

Ejemplo recibiendo un nombre por props

Veamos el mismo ejemplo anterior, pero en el cual, el componente MiComponente, recibe como parámetro (del componente padre), el nombre a mostrar

```
export default function MiComponente(props) {
    const onPresionoBoton = () => {
        console.log('Presiono botón')
    }

    return (
        <div>
            <span>{props.nombre}</span>
            <button onClick={onPresionoBoton}>Soy un botón</button>
        </div>
    )
}
```

Propagación de evento al componente padre

Muchas veces el componente en el cual se produce el evento, no es el componente que va a realizar la operación sobre el mismo. Tomemos como ejemplo un formulario que debe agregar una tarea en un listado. Si disponemos de la siguiente estructura de componentes

- AppTareas
  - Listado
  - Formulario

El evento que se produce en el Formulario, no puede actualizar el componente Formulario, por lo que debe propagar el evento al componente padre de los dos AppTareas, para que este se encargue de notificar al componente Listado.

Veamos como el componente Formulario puede propagar un evento a AppTareas.

Supongamos que el código de AppTareas (simplificado) es el siguiente

```
import React from 'react';
import Listado from './Listado';
import Formulario from './Formulario';

export default function AppTareas(props) {

    const agregarDesdeFormulario = (valor) => {
        // Lógica de agregar valor
    }

    render() {
        return (
            <div >
                <Listado />
                <Formulario onAgregar={props.agregarDesdeFormulario} />
            </div>
        )
    }
}
```

Y el código de Formulario es el siguiente

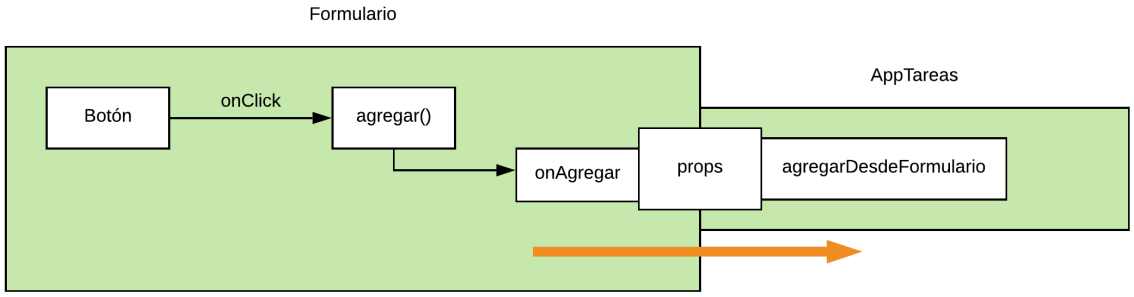
```
import React from 'react';

export default function Formulario() {

    const agregar = () => {
        const valorAAgregar = 'Prueba';
        props.onAgregar(valorAAgregar);
    }

    render() {
        return (
            <div >
                <button onClick={agregar}>Agregar</button>
            </div>
        )
    }
}
```

Cuando el usuario hace click sobre el botón del componente Formulario, se sucede el siguiente flujo de eventos.



1. El usuario hace click sobre el botón
    a. Se dispara el evento **onClick**
    b. Se llama a la función **agregar**

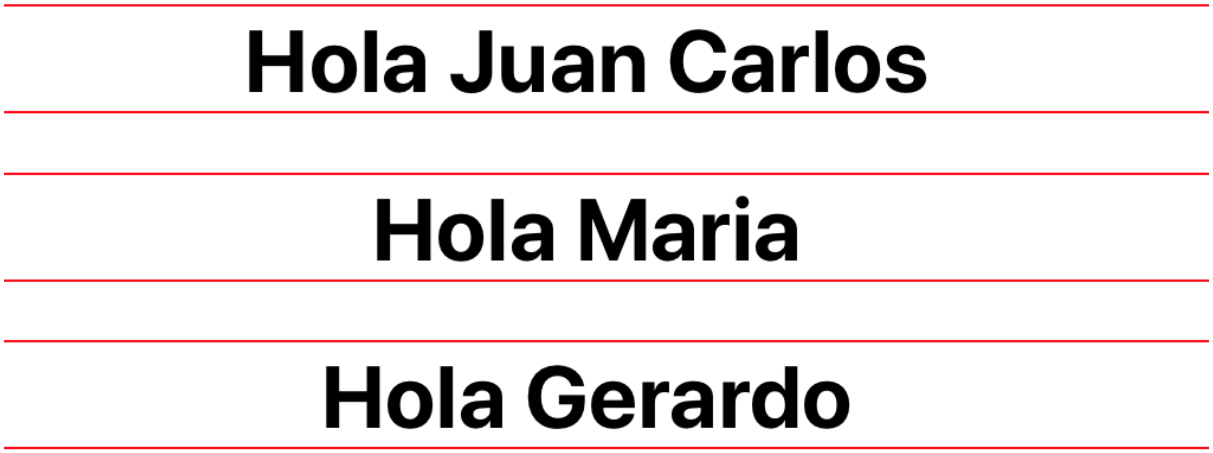
c. La función **agregar()** llama a la función que se haya asignado a **props.onAgregar**
d. El evento que se asignó a **props.onAgregar** es **agregarDesdeFormulario()**, por lo que se llama a dicha función.

Habitualmente el método agregarDesdeFormulario() generará un cambio de estado que hará que todos los componentes hijos actualicen sus propiedades (en caso que haga falta)

6. Ejemplo

A los componentes realizados anteriormente, se le debe pasar como parámetro la información a mostrar…

- img: la dirección de la imagen (componente imagen)
- title: el título a mostrar (componente título)
- description: la descripción a mostrar (componente descripción)



Que es generado por las siguientes llamadas al componente PrimerComponente.

```
<div className="App">
    <PrimerComponente name="Juan Carlos" />
    <PrimerComponente name="Maria" />
    <PrimerComponente name="Gerardo" />
</div>
```

Todo comienza por un vector

Generalmente cuando debemos mostrar un componente varias veces, el origen de los datos es un vector…

```
const vectorNombres = [
    'Juan Carlos',
    'Maria',
    'Gerardo'
];
```

Opción 1: uso de array.forEach

```
let respuesta = [];
vectorNombres.forEach(unItem => {
    respuesta.push(<PrimerComponente name={unItem} />)
})

return (
    <div className="App">
        {respuesta}
    </div>
)
```

Opción 2: uso de array.map()

```
const respuesta = vectorNombres.map(unItem => <PrimerComponente
name={unItem} />)

return (
    <div className="App">
        {respuesta}
    </div>
)
```

o una versión más reducida

```
return (
    <div className="App">
        {vectorNombres.map(unItem => <PrimerComponente
name={unItem} />)}
    </div>
)
```

7. Trabajo Práctico

- Ahora se debe mostrar el siguiente vector en el componente

```
vector = [
        { url: 'https://placeimg.com/80/80/people?id=1', title: 'Titulo 1', description:
'Descripcion imagen 1'},
        { url: 'https://placeimg.com/80/80/people?id=2', title: 'Titulo 2', description:
'Descripcion imagen 2'},
        { url: 'https://placeimg.com/80/80/people?id=3', title: 'Titulo 3', description:
'Descripcion imagen 3'},
        { url: 'https://placeimg.com/80/80/people?id=4', title: 'Titulo 4', description:
'Descripcion imagen 4'}
]
```

Bibliografía utilizada y sugerida

Node JS (n.d.) Recuperado de: https://nodejs.org/es/

React JS (n.d.) Recuperado de: https://es.reactjs.org/

Wikipedia (n.d.) Recuperado de: https://es.wikipedia.org/wiki/React

Mozilla Developer Network (n.d.) Recuperado de: https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Array

Lo que vimos:

Componentes, estados y eventos.

Lo que viene:

Redux.