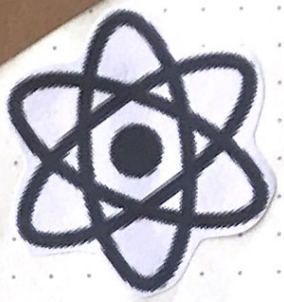
react



¿QUÉ ES?

Una biblioteca de JavaScript para construir interfaces de usuario.

```
ReactDOM.render(
<h1>Hola, mundo!</h1>
document.getElementById('root')
);
```

* Este pequeño ejemplo muestra un encabezado con el texto:

Hola, mundo!

JSX.

extensión de la sintaxis de JavaScript. para uso con React.

JSX produce "elementos" de React.

- ELEMENTO: Bloque más pequeño de las aplicaciones de React.
- Describe lo que quieres ver en la pantalla.

Considerando las siguientes variables:

```
const name = 'Elon Musk';
const element = <h1> Hello, {name} </h1> ;
```

- Para renderizarlo:

```
ReactDOM.render (
  element,
  document.getElementById('root')
);
```

SE PASAN AMBOS A RENDER

→ Imagina que hay un en alguna parte de tu HTML

, siendo un nodo raiz.

> muestra: Hello, Elon Musk.

Se puede utilizar JSX dentro de declaraciones IF y bucles FOR, asignarlo a variables, aceptarlo como argumento y retornarlo desde dentro de funciones.

@MayoLedes

atributos con jsx.

- Puedes utilizar comillas para especificar strings como atributos:

```
const element = <div tabIndex="0"></div>;
```

- También usar llaves para insertar una expresión JS:

```
const element = <img src={user.avatarURL}></img>;
```

⚠ No utilices comillas rodeando llaves.

"comillas": valores de los strings
{llaves}: expresiones JavaScript

- Si una etiqueta está vacia, puedes cerrarla con /> :

```
const element = <img src={user.avatarUrl} />
```

JSX

representa objetos

↓

llamados "ELEMENTOS DE REACT"

React lee estos objetos y los usa para construir el DOM

éstos ejemplos son idéntices

```
const element = (
<h1 className = 'greeting'>
  Hola, mundo!
</h1>
);
```

```
const element = React.createElement(
'h1',
{ className: 'greeting' },
'Hola, mundo!'
);
```

renderizando elementos

ELEMENTO ? BLOQUE MÁS PEQUEÑO DE LAS APLICACIONES DE REACT.

Describe lo que quieres ver en la pantalla.

```
const element = <h1>Hello, world</h1>;
```

Imagina que hay un `<div>` en alguna parte del HTML

```
<div id="root"></div>
```

Para renderizar un elemento de React en un nodo raiz del DOM, pasa ambos a `ReactDOM.render()`

```
const element = <h1>Hello, world</h1>;
ReactDOM.render(element, document.getElementById('root'));
```

ACTUALIZANDO EL ELEMENTO :

React solo actualiza lo que es necesario.

ReactDOM compara el elemento y sus hijos con el elemento anterior, y solo aplica las actualizaciones del DOM que son necesarias para que el DOM esté en el estado deseado.

componentes

permiten separar la interfaz de usuario en piezas independientes.

Los componentes aceptan entradas arbitrarias, llamadas "props", y devuelven a React elementos que describen lo que se debe ver en la pantalla.

La forma más sencilla de definir un componente es escribir una función de JavaScript

```
function Welcome (props) {
return <h1> Hello, {props.name}
</h1>;
}
```

ACEPTA UN ARGUMENTO DE OBJETO 'PROPS' (que proviene de propiedades) CON DATOS Y DEVUELVE UN ELEMENTO DE REACT.

Estos componentes son "Funcionales" porque literalmente son funciones de JavaScript

renderizando componentes

Además de etiquetas del DOM, los elementos también pueden representar componentes definidos por el usuario:

```
function Welcome (props) {
return <h1> Hello, {props.name} </h1> ;
}

const element = <Welcome name = 'Elon' />
ReactDOM.render (
 element ,
document.getElementById ('root') );
```

1: Llamamos a ReactDOM.render() con el elemento <Welcome name = "Elon". />

2: React llama al componente Welcome con {name : 'Elon'} como 'props'.

3: El componente Welcome devuelve un elemento <h1> Hello, Elon </h1> como resultado

4: React DOM actualiza el DOM para que coincida con <h1> Hello, Elon </h1>

react.component.

React te permite definir componentes como funciones o clases.

Para definir una clase de componente React, necesitas extender React.Component:

```
class Welcome extends React.Component {
 render() {
  return <h1>Hello, {this.props.name}</h1>
 }
}
```

Si aún no utilizas ES6, puedes utilizar el módulo create-react-class:

```
var createReactClass = require('create-react-class');
var Greeting = createReactClass({
    render: function() {
      return <h1>Hello, {this.props.name}</h1>
    }
});
```

métodos de ciclo de vida

- comúnmente usados -

montaje

Estos métodos se llaman cuando se crea una instancia de un componente y se inserta en el DOM.

- constructor():

SI NO INICIALIZAS EL ESTADO Y NO ENLAZAS LOS MÉTODOS, NO NECESITAS IMPLEMENTAR UN CONSTRUCTOR PARA TU COMPONENTE REACT.

Normalmente, se utiliza para dos propósitos:

→ Para inicializar un estado local asignando un objeto al `this.state`

→ Para enlazar manejadores de eventos a una instancia

- render():

ES EL ÚNICO MÉTODO REQUERIDO EN UN COMPONENTE DE CLASE.

Cuando se llama, examina a `this.props` y `this.state`, y devuelve uno de los siguientes tipos:

→ Elementos de React (Ej:)

→ Arrays y fragmentos: Permiten que puedas devolver múltiples elementos desde el render.

→ Portales: Permiten renderizar hijos a otro subárbol del DOM.

→ String y números

→ Booleanos o nulos: No renderizan nada.

- componentDidMount()

Se invoca inmediatamente después de que un componente se monte (se inserte en el árbol).

Este método es un buen lugar para establecer cualquier suscripción. Si lo haces, no olvides darle de baja en componentWillUnmount().

Actualización

Puede ser causada por cambios en los props o el estado.

- render()
- componentDidUpdate():

componentDidUpdate(prevProps, prevState, snapshot)

Se invoca inmediatamente después de que la actualización ocurra.
Este método no es llamado para el renderizador inicial.
Es un buen lugar para hacer solicitudes de red siempre y cuando compare los accesorios actuales con los anteriores.

Desmontaje

- componentWillUnmount()

Se invoca inmediatamente antes de desmontar y destruir un componente. Realiza las tareas de limpieza necesarias en este método.

No debes llamar setState() en componentWillUnmount() porque el componente nunca será vuelto a renderizar.

manejando eventos

→ Los eventos de React se nombran usando camelCase, en vez de minúsculas.

→ Con JSX pasas una función como el manejador del evento, en vez de un string.

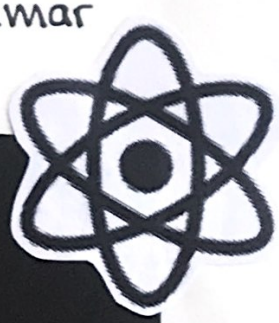
HTML

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

REACT

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

En React no puedes retornar `false` para prevenir el comportamiento por defecto. Debes llamar `preventDefault`.



```
Function ActionLink() {
 function handleClick(e) {
   e.preventDefault();
   console.log('The link was clicked.'); }
 return (
<a href="#" onClick=
{handleClick}>
      Click me </a>
 ); }
```

Los **EVENTOS** en React se definen de manera declarativa, producidos con JSX en el método render().

Para definirlos

- Se indica el tipo de evento.
- Indicamos el método que hará de manejador de evento.

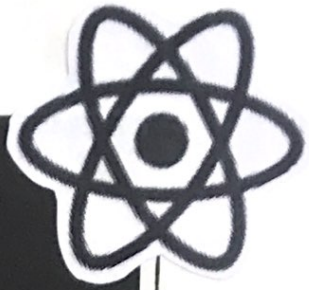
Un manejador de evento es, básicamente, una función que define la funcionalidad que se ejecutará al dispararse un evento.

```
render(){
 return(
<p>
<button onClick={this.toggleSlider}>Mostrar / Ocultar Slider
</button>
</p>
 );
}
```

Las declaraciones de eventos en React tienen siempre la forma:

"on" + tipo de evento (onClick, onInput...)

Luego, definimos el manejador:



```
toggleSlider() {
  alert('Haz click');
}
```

Al implementar un manejador de eventos es habitual que queramos acceder a las props o métodos del componente.
Pero en las funciones no es posible acceder a this como referencia al objeto sobre el que se invoca el método.
Se necesita 'bindear' el contexto:

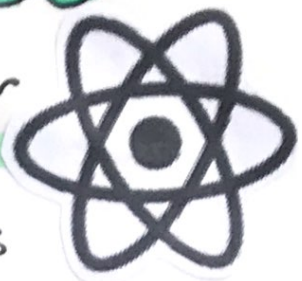
- Directamente en el constructor del componente

```
constructor(props) {
  super(props);
  this.state = {
    showSlider: false
  };
  this.toggleSlider = this.toggleSlider.bind(this); }
```

- En el template.

```
<button onClick={thrs.toggleSlider.bind(this)}>Mostrar/Ocultar Slider</button>
```

renderizando
~ condicional ~



En React, puedes crear distintos componentes que encapsulan el comportamiento que necesitas. Entonces, puedes renderizar solamente algunos de ellos, dependiendo del estado de tu aplicación.

El renderizado condicional usa operadores de Js como **if** o el **operador condicional** para crear elementos representando el estado actual.

Considera estos dos componentes

```
Function UserGreeting(props){
  return <h1> Welcome back!</h1>
}
```

```
Function GuestGreeting(props){
  return <h1> Please sign up.</h1>
}
```

Creamos un componente **Greeting**

```
Function Greeting(Props){
  const isLoggedIn =props.isLoggedIn;
  if(isLoggedIn){
    return <UserGreeting />;
  }
  return <guest greeting />;
}
ReactDOM.render(
  // Intentar cambiando isLoggedIn={true}:
  <Greeting isLoggedIn={False} />;
  document.getElementById('root')
);
```

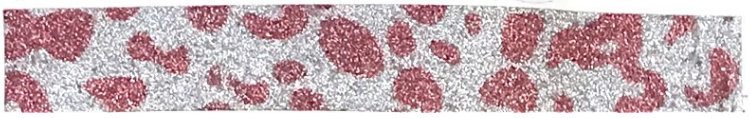
evitar que el componente se renderice

Es posible que desees que un componente se oculte a sí mismo aunque haya sido renderizado por otro componente.
Para hacer esto, devuelve `null`

```
Function WarningBanner(props) {
  if (!props.warn) {
    return null;
  }
  return (
    <div className="warning">
      Warning
    </div>
  );
}
```

*Si el valor del prop es `false`, entonces el componente no se renderiza.

El devolver `null` desde el método `render` de un componente no influye en la activación de los métodos del ciclo de vida del componente. Por ejemplo, `componentDidUpdate` seguirá siendo llamado.



Listas

RENDERIZADO DE MÚLTIPLES COMPONENTES.

Puedes hacer colecciones de elementos e incluirlos en JSX usando llaves {}

```
const numbers = [1,2,3,4,5];
const listItems = numbers.map((number) =>
  <li>{number}</li>
);
```

recorremos el array numbers usando la Funcion map() de JavaScript. Devolvemos un elemento <li> por cada item. Finalmente, asignamos el array de elementos resultante a listItems.

```
ReactDOM.render(
 <ul> {listItems} </ul>,
 document.getElementById('root')
);
```

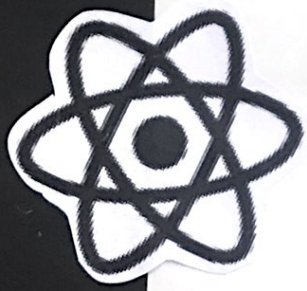
Incluimos entero el array listItems dentro de un elemento <ul>, y lo renderizamos al DOM.

*Este código muestra una lista de números entre 1 y 5.

Refactorizamos el ejemplo anterior en un componente que acepte un array de numbers e imprima una lista de elementos:

```
function NumberList(props) {
const numbers = props.numbers;
const listItems = numbers.map ((number) =>
 <li> {number} </li>
);
return (
 <ul> {listItems} </ul>
);
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
<NumberList numbers={numbers} />,
 document.getElementById('root')
);
```



Al ejecutar este código, serás advertido que una key debería ser proporcionada para ítems de lista.

```
<li key={number.toString()}>
     {number}
 </li>
```

Key?

Ayudan a React a identificar que ítems han cambiado, son agregados, o son eliminados.
Las keys deben ser dadas a los elementos dentro del array.
Habitualmente vas a usar IDs de tus datos como key.

```
<li key={todo.id}>
  {todo.text} </li>
```

F·O·R·M·U·L·A·R·I·O·S

En HTML, los elementos de formularios como los `<input>`, `<textarea>` y el `<select>` mantienen sus propios estados y los actualizan de acuerdo a la interacción del usuario.
En React, el estado mutable es mantenido normalmente en la propiedad del estado de los componentes, y solo se actualiza con `setState()`.

textarea

En React, un `<textarea>` utiliza un atributo `value`, y es inicializado en el constructor.

```
class myForm extends React.component {
  constructor(props) {
   super(props);
   this.state = {
   description: 'The content of a textarea goes in the
   value attribute' };
 }

  render () {
   return (
    <Form>
    <textarea value = {this.state.description} />
   </Form> );
  }
}
```

select

React utiliza el atributo `value` en la raíz de la etiqueta `select`.

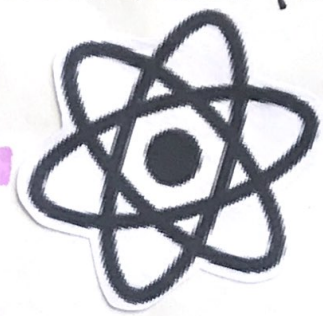
```
class MyForm extends React.Component{
    constructor (props) {
     super (props);
     this.state = {
        mycar : 'Cybertruck'
     };
  }
  render() {
   return (
   <form>
   <select value = {this.state.mycar} >
    <option value = "Twingo"> Twingo </option>
    <option value = "Cybertruck"> Cybertruck </option>
   </select>
   </form> );
  }
}
```

Puedes pasar un array al atributo value, permitiendo que selecciones múltiples opciones en una etiqueta `Select`:

```
<select multiple={true} value={['B', 'C']}>
```

En resumen, esto hace que <input type="text">, <textarea> y <select> trabajen de manera similar, todos aceptan un atributo value el cual puedes usar para implementar un componente controlado.

Un campo de un formulario cuyos valores son controlados por React es denominado 'componente controlado'.

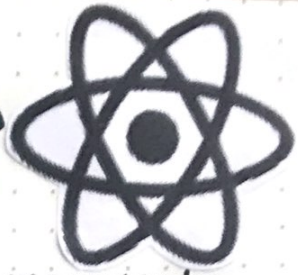


levantando el estado

En React, la comparticion del estado puede lograrse moviendo el estado hacia arriba al ancestro común más cercano de los componentes que lo necesitan. A esto se le llama "levantar el estado".

El componente que posee el estado compartido se convierte en la "fuente de verdad"

composición



{React tiene un potente modelo de composición, y se recomienda usar composición en lugar de herencia}

~ CONTENCIÓN ~

Algunos componentes no conocen sus hijos de antemano. Es común para componentes como Sidebar o Dialog que representan "cajas" genéricas.

Se recomienda que estos componentes usen la prop especial children para pasar elementos hijos directamente en su resultado:

```
Function FancyBorder (props) {
  return (
  <div className={'FancyBorder FancyBorder-' + props.color}>
    {props.children}
  </div>
 );
}
```

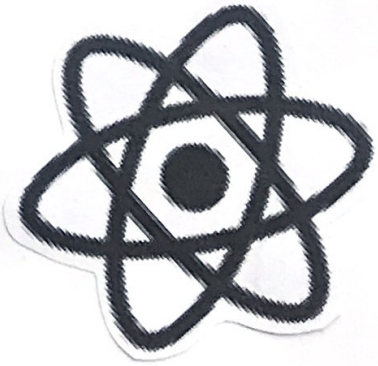
Esto permite que otros componentes les pasen hijos arbitrarios anidando el JSX.

```
Function WelcomeDialog () {
  return (
<FancyBorder color="blue">
 <h1 className="Dialog-title">
   Welcome
 </h1>
 <p className="Dialog-message">
   Thank you for visiting
 </p>
</FancyBorder> );
}
```

.: ESPECIALIZACIÓN :.

A veces pensamos en componentes como "casos" concretos de otros componentes.

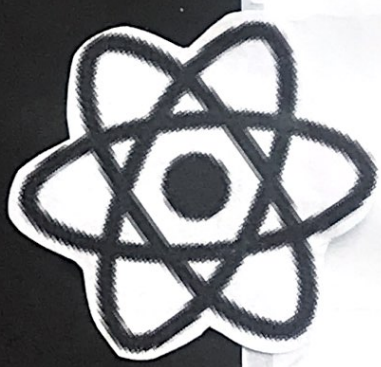
Por ejemplo, podriamos decir que un WelcomeDialog es un caso concreto de Dialog.



En react, esto también se consigue por composición, en la que un componente más "específico" renderiza uno más "genérico" y lo configura con props:

```
Function Dialog(props) {
render(
 <FancyBorder color="blue">
 <h1 className="Dialog-title">
  {props.title}
 </h1>
 <p className="Dialog-message">
    {props.message}
 </p>
 </FancyBorder>
);
}

Function WelcomeDialog() {
 return(
 <Dialog
    title="Welcome"
   message="Thank you for visiting" />
 );
 }
```



Fuente

- Documentación oficial de ReactJs:
  "conceptos principales"

https://es.reactjs.org/docs

- "Eventos":

https://desarrolloweb.com

Hecho con ❤ por Majo

Tw: @Majoledes

Enero, 2020.-