JSX

JSX es una extensión de sintaxis para JavaScript que permite escribir marcas similares a HTML dentro de un archivo JavaScript. Aunque hay otras formas de escribir componentes, la mayoría de los desarrolladores de React prefieren la JSX y la mayoría de los repositorios de código lo usan.

1. JSX: Poniendo marcado dentro de JavaScript

La Web se ha construido sobre HTML, CSS, y JavaScript. Durante muchos años, los desarrolladores web mantuvieron el contenido en HTML, el diseño en CSS, y la lógica en JavaScript, generalmente en archivos separados. El contenido se codeo dentro del HTML mientras que la lógica de la página vivía por separado en JavaScript:

HTML

```
<div>
  <p></p>
  <form>
  </form>
</div>
```

JAVASCRIPT

```
isLoggedIn() {...}
onClick() {...}
onSubmit() {...}
```

Pero, a medida que la Web se volvió más interactiva, la lógica determinó cada vez más el contenido, resulta que JavaScript estaba a cargo del HTML. Esto es la razón por la que en React, la lógica de renderizado y el marcado viven juntos en el mismo lugar: componentes.

Sidebar

```
Sidebar() {
  if (isLoggedIn()) {
    <p>Welcome</p>
  } else {
    <Form />
  }
}
```

Form

```
Form() {
  onClick() {...}
  onSubmit() {...}
  <form onSubmit>
    <input onClick />
    <input onClick />
  </form>
}
```

Mantener juntas la lógica de renderizado y el marcado de un botón, garantiza que permanezcan sincronizados entre sí en cada edición. Por el contrario, los detalles que no están relacionados, como el marcado de un botón y el marcado de una barra lateral, están aislados entre sí, haciendo que sea más seguro cambiar cualquiera de ellos por su cuenta.

Cada componente de React es una función de JavaScript que puede contener algún marcado que React muestra en el navegador. Los componentes de React usan una extensión de sintaxis llamada JSX para representar el marcado. JSX se parece mucho a HTML, pero es un poco más estricto y puede mostrar información dinámica. La mejor manera de comprender esto es convertir algunas marcas HTML en marcas JSX.

2. Convirtiendo HTML a JSX

Supongamos que tienes algo de HTML (perfectamente válido):

```
<h1>Hedy Lamarr's Todos</h1>
<img
src="https://i.imgur.com/yXOvdOSs.jpg"
alt="Hedy Lamarr"    class="photo"
>
<ul>
    <li>Invent new traffic lights
    <li>Rehearse a movie scene
<li>Improve the spectrum technology
</ul>
```

Y quieres ponerlo en tu componente:

```
export default function TodoList() {
return (     // ???
  )
}
```

Si lo copiás y pegás tal como está, no funcionará:

```
export default function TodoList() {
return (
    // Este ejemplo no funciona!
<h1>Hedy Lamarr's Todos</h1>
    <img
src="https://i.imgur.com/yXOvdOSs.jpg"
alt="Hedy Lamarr"         class="photo"
    >
    <ul>
      <li>Invent new traffic lights
      <li>Rehearse a movie scene
      <li>Improve the spectrum technology
    </ul>
  );
}
```

Error

/App.js: Adjacent JSX elements must be wrapped in an enclosing tag. Did you want a JSX fragment <>...</>? (5:4)

3 | // This doesn't quite work!

4 | <h1>Hedy Lamarr's Todos</h1>

> 5 | <img

| ^

6 | src="https://i.imgur.com/yXOvdOSs.jpg"

7 | alt="Hedy Lamarr"

8 | class="photo"

Esto se debe a que JSX es más estricto y tiene algunas restricciones más que HTML.

3. Las Reglas de JSX 1. Devolver un solo elemento raíz

Para devolver múltiples elementos de un componente, envuélvelos con una sola etiqueta principal.

Por ejemplo, podés usar un :

```
<div>
  <h1>Hedy Lamarr's Todos</h1>
  <img
src="https://i.imgur.com/yXOvdOSs.jpg"
alt="Hedy Lamarr"      class="photo"
  >
<ul>
    ...
  </ul>
</div>
```

Si no deseás agregar un adicional a tu marcado, puedes escribir <> y </> en su lugar:

```
<>
  <h1>Hedy Lamarr's Todos</h1>
  <img
src="https://i.imgur.com/yXOvdOSs.jpg"
alt="Hedy Lamarr"      class="photo"
  >
<ul>
    ...
  </ul>
</>
```

Esta etiqueta vacía se llama un Fragmento. Los Fragmentos te permiten agrupar cosas sin dejar ningún rastro en el árbol HTML del navegador.

2. Cierra todas las etiquetas

JSX requiere que las etiquetas se cierren explícitamente: las etiquetas de cierre automático como <img> deben convertirse en <img />, y etiquetas envolventes como oranges deben convertirse como oranges.

Así es como la imagen y los elementos de la lista de Hedy Lamarr se ven cerrados:

```
<>   <img
src="https://i.imgur.com/yXOvdOSs.jpg"
alt="Hedy Lamarr"      class="photo"
   />
  <ul>
    <li>Invent new traffic lights</li>
    <li>Rehearse a movie scene</li>
    <li>Improve the spectrum technology</li>
  </ul>
</>
```

3. camelCase

JSX se convierte en JavaScript y los atributos escritos en JSX se convierten en keys de objetos JavaScript. En tus propios componentes, a menudo vas a querer leer esos atributos en variables. Pero JavaScript tiene limitaciones en los nombres de variables. Por ejemplo, sus nombres no pueden contener guiones ni ser palabras reservadas como class.

Por eso, en React, muchos atributos HTML y SVG están escritos en camelCase. Por ejemplo, en lugar de stroke-width usa strokeWidth. Dado que class es una palabra reservada, en React escribes className en su lugar, con el nombre de la propiedad DOM correspondiente:

```
<img
src="https://i.imgur.com/yXOvdOSs.jpg"
alt="Hedy Lamarr"    className="photo"
/>
```

Podés encontrar todos estos atributos en la lista de props de los componentes DOM. Si te equivocas en uno, no te preocupes, React imprimirá un mensaje con una posible corrección en la consola del navegador.

RENDERIZADO CONDICIONAL Y DE LISTAS

1. Renderizado condicional

Usando componentes es común necesitar mostrar diferentes cosas dependiendo de diferentes condiciones. En React, puedes renderizar JSX de forma condicional utilizando la sintaxis de JavaScript como las declaraciones if, && y los operadores?:.

2. Devolución condicional de JSX

Supongamos que tenemos un componente PackingList que muestra varios Items, que pueden ser marcados como empaquetados o no:

```
function Item({ name, isPacked }) {
  return <li className="item">{name}</li>;
}

export default function PackingList() {
  return (
    <section>
      <h1>Sally Ride's Packing List</h1>
      <ul>
        <Item
          isPacked={true}
          name="Space suit"
        />
        <Item
          isPacked={true}
          name="Helmet with a golden leaf"
        />
        <Item
          isPacked={false}
```

```
            name="Photo of Tam"
          />
        </ul>
      </section>
    );
}
```

Podemos observar que algunos de los componentes Item tienen su prop isPacked asignada a true en lugar de false. Se deseamos añadir una marca de verificación (✓) a los elementos empaquetados si isPacked={true}.

Podemos escribir esto como una declaración if/else así:

```
if (isPacked) {
  return <li className="item">{name} ✓</li>;
}
return <li className="item">{name}</li>;
```

Si la prop isPacked es true, este código devuelve un árbol JSX diferente. Con este cambio, algunos de los elementos obtienen una marca de verificación al final:

```
function Item({ name, isPacked }) {
  if (isPacked) {
    return <li className="item">{name} ✓</li>;
  }
  return <li className="item">{name}</li>;
}

export default function PackingList() {
  return (
    <section>
      <h1>Sally Ride's Packing List</h1>
      <ul>
        <Item
          isPacked={true}
```

```
          name="Space suit"
        />
        <Item
          isPacked={true}
          name="Helmet with a golden leaf"
        />
        <Item
          isPacked={false}
          name="Photo of Tam"
        />
      </ul>
    </section>
  );
}
```

Podemos probar editar lo que se devuelve en cualquiera de los dos casos y observar cómo cambia el resultado.

De esta forma estamos creando una lógica de ramificación con las sentencias if y return de JavaScript. En React, el flujo de control (como las condiciones) es manejado por JavaScript.

3. Devolución de nada con null

En algunas situaciones, no queremos mostrar nada en absoluto. Por ejemplo, digamos que no queremos mostrar elementos empaquetados en absoluto. Un componente debe devolver algo. En este caso, puedes devolver null:

```
if (isPacked) {
  return null;
}
return <li className="item">{name}</li>;
```

Si isPacked es verdadero, el componente no devolverá nada, null. En caso contrario, devolverá JSX para ser renderizado.

```
function Item({ name, isPacked }) {
  if (isPacked) {
    return null;
  }
  return <li className="item">{name}</li>;
}

export default function PackingList() {
  return (
    <section>
      <h1>Sally Ride's Packing List</h1>
      <ul>
        <Item
          isPacked={true}
          name="Space suit"
        />
        <Item
          isPacked={true}
          name="Helmet with a golden leaf"
        />
        <Item
          isPacked={false}
          name="Photo of Tam"
        />
      </ul>
    </section>
  );
}
```

En la práctica, devolver null en un componente no es común porque podría sorprender a un desarrollador que intente renderizarlo. Lo más frecuente

es incluir o excluir condicionalmente el componente en el JSX del componente padre.

4. Exclusión condicional de JSX

En el ejemplo anterior, controlábamos qué árbol JSX (si es que había alguno) era devuelto por el componente. Es posible que hayamos notado alguna duplicación en la salida de la renderización:

```
<li className="item">{name} ✓</li>
```

es muy similar a

```
<li className="item">{name}</li>
```

Ambas ramas condicionales devuelven <li className="item">...:

```
if (isPacked) {
  return <li className="item">{name} ✓</li>;
}
return <li className="item">{name}</li>;
```

Aunque esta duplicación no es perjudicial, podría hacer que tu código sea más difícil de mantener. Ejemplo si queremos cambiar el className tenemos que hacerlo en dos lugares en el código. En esta situación, podríamos incluir condicionalmente un poco de JSX para hacer tu código más legible.

5. Operador condicional (ternario) (?:)

JavaScript tiene una sintaxis compacta para escribir una expresión condicional, el operador condicional u “operador ternario”.

En lugar de esto:

```
if (isPacked) {
  return <li className="item">{name} ✓</li>;
}
return <li className="item">{name}</li>;
```

Podés escribir esto:

```
return (
  <li className="item">
    {isPacked ? name + ' ✓' : name}
```

```
  </li>
);
```

Podés leerlo como “si isPacked es verdadero, entonces (?) renderiza name + ' ✓', de lo contrario (:) renderiza name”)

Ahora digamos que queremos envolver el texto del elemento completado en otra etiqueta HTML, como <del> para tacharlo. Podemos añadir aún más líneas nuevas y paréntesis para que sea más fácil anidar más JSX en cada uno de los casos:

```
function Item({ name, isPacked }) {
  return (
    <li className="item">
      {isPacked ? (
        <del>
          {name + ' ✓'}
        </del>
      ) : (
        name
      )}
    </li>
  );
}

export default function PackingList() {
  return (
    <section>
      <h1>Sally Ride's Packing List</h1>
      <ul>
        <Item
          isPacked={true}
```

```
          name="Space suit"
        />
        <Item
          isPacked={true}
          name="Helmet with a golden leaf"
        />
        <Item
          isPacked={false}
          name="Photo of Tam"
        />
      </ul>
    </section>
  );
}
```

Este estilo funciona bien para condiciones simples, pero hay que utilízalo con moderación. Si los componentes se desordenan con demasiado marcado condicional anidado, consideremos la posibilidad de extraer componentes hijos para limpiar las cosas. En React, el marcado es una parte del código, por lo que podemos utilizar herramientas como variables y funciones para ordenar las expresiones complejas.

6. Operador lógico AND (&&)

Otro atajo común que encontraremo es el operador lógico AND (&&) de JavaScript. Dentro de los componentes de React, a menudo surge cuando queremos renderizar algún JSX cuando la condición es verdadera, o no renderizar nada en caso contrario. Con &&, podrías renderizar condicionalmente la marca de verificación sólo si isPacked es true:

```
return (
  <li className="item">
    {name} {isPacked && '✓'}
  </li>
);
```

Podés leer esto como “si isPacked, entonces (&&) renderiza la marca de verificación, si no, no renderiza nada.”

```
function Item({ name, isPacked }) {
  return (
    <li className="item">
      {name} {isPacked && '✓'}
    </li>
  );
}

export default function PackingList() {
  return (
    <section>
      <h1>Sally Ride's Packing List</h1>
      <ul>
        <Item
          isPacked={true}
          name="Space suit"
        />
        <Item
          isPacked={true}
          name="Helmet with a golden leaf"
        />
        <Item
          isPacked={false}
          name="Photo of Tam"
        />
      </ul>
    </section>
  );
}
```

Una expresión JavaScript && devuelve el valor de su lado derecho (en nuestro caso, la marca de verificación) si el lado izquierdo (nuestra

condición) es true. Pero si la condición es false, toda la expresión se convierte en false. React considera false como un "agujero" en el árbol JSX, al igual que null o undefined, y no renderiza nada en su lugar.

No hay que poner números a la izquierda de &&. Porque para comprobar la condición, JavaScript convierte el lado izquierdo en un booleano automáticamente. Sin embargo, si el lado izquierdo es 0, entonces toda la expresión obtiene ese valor (0), y React representará felizmente 0 en lugar de nada. Por ejemplo, un error común es escribir código como messageCount && New messages . Es fácil suponer que no renderiza nada cuando messageCount es 0, pero en realidad renderiza el propio 0. Para arreglarlo, tenemos que hacer que el lado izquierdo sea un booleano: messageCount > 0 && New messages .

7. Asignación condicional de JSX a una variable

Para no complicar el código, podemos usar sentencia if y una variable. POdemos reasignar las variables definidas con let, así que empezamos proporcionando el contenido por defecto que queremos mostrar, el nombre:

```
let itemContent = name;
```

Utilizamos una sentencia if para reasignar una expresión JSX a itemContent si isPacked es true:

```
if (isPacked) {
  itemContent = name + " ✓";
}
```

Las llaves abren la "ventana a JavaScript". Inserta la variable con llaves en el árbol JSX devuelto, anidando la expresión previamente calculada dentro de JSX:

```
<li className="item">
  {itemContent}
</li>
```

8. Renderizado de listas

A menudo necesitamos mostrar muchos componentes similares de una colección de datos. Podemos usar los métodos de array de JavaScript para

manipular un array de datos. En este caso, usaremos filter() y map() con React para filtrar y transformar tu array de datos en un array de componentes.

Supongamos que tenemos una lista de contenido.

```
<ul>
  <li>Creola Katherine Johnson: mathematician</li>
  <li>Mario José Molina-Pasquel Henríquez: chemist</li>
  <li>Mohammad Abdus Salam: physicist</li>
  <li>Percy Lavon Julian: chemist</li>
  <li>Subrahmanyan Chandrasekhar: astrophysicist</li>
</ul>
```

La única diferencia entre esos elementos de la lista es su contenido, sus datos. Ejemplo de cómo generar una lista de elementos de un array:

1. Mové los datos en un array:

```
const people = [
  'Creola Katherine Johnson: mathematician',
  'Mario José Molina-Pasquel Henríquez: chemist',
  'Mohammad Abdus Salam: physicist',
  'Percy Lavon Julian: chemist',
  'Subrahmanyan Chandrasekhar: astrophysicist'
];
```

2. Mapeá los miembros de people en un nuevo array de nodos JSX, listItems:

```
const listItems = people.map(person => <li>{person}</li>);
```

3. Devolvé listItems desde tu componente envuelto en un <ul>:

```
return <ul>{listItems}</ul>;
```

Acá está el resultado:

```
const people = [
  'Creola Katherine Johnson: mathematician',
  'Mario José Molina-Pasquel Henríquez: chemist',
```

```
  'Mohammad Abdus Salam: physicist',
  'Percy Lavon Julian: chemist',
  'Subrahmanyan Chandrasekhar: astrophysicist'
];

export default function List() {
  const listItems = people.map(person =>
    <li>{person}</li>
  );
  return <ul>{listItems}</ul>;
}
```

Obtendremos un mensaje de error por consola:

```
Warning: Each child in a list should have a unique "key" prop.

heck the render method of `List`. See https://reactjs.org/link/warning-keys for more information.
    at li
    at List
```

Esto se debe a que no enviamos ningún key/id al contenido de la lista.

9. Filtrar arrays de objetos

Estos datos pueden ser estructurados incluso más.

```
const people = [{
  id: 0,
  name: 'Creola Katherine Johnson',
  profession: 'mathematician',
}, {
  id: 1,
  name: 'Mario José Molina-Pasquel Henríquez',
  profession: 'chemist',
}, {
  id: 2,
```

```
  name: 'Mohammad Abdus Salam',
  profession: 'physicist',
}, {
  name: 'Percy Lavon Julian',
  profession: 'chemist',
}, {
  name: 'Subrahmanyan Chandrasekhar',
  profession: 'astrophysicist',
}];
```

Supongamos que queremos mostrar solo las personas cuya profesión sea 'chemist'. Podemos usar el método filter() de JavaScript para devolver solo esas personas. Este método toma un array de objetos, los pasa por un «test» (una función que devuelve true o false), y devuelve un nuevo array de solo esos objetos que han pasado el test (que han devuelto true).

Solo queremos los objetos donde profession es 'chemist'. La función «test» para esto se ve como (person) => person.profession === 'chemist'. Aquí está cómo juntarlo:

1. Creamos un nuevo array solo de personas que sean «químicos», chemists, llamando al método filter() en people filtrando por person.profession === 'chemist':

```
const chemists = people.filter(person =>
  person.profession === 'chemist'
);
```

2. Ahora mapeá sobre chemists:

```
const listItems = chemists.map(person =>
  <li>
     <img
       src={getImageUrl(person)}
       alt={person.name}
     />
     <p>
       <b>{person.name}:</b>
```

```
        {' ' + person.profession + ' '}
        known for {person.accomplishment}
      </p>
    </li>
);
```

3. Por último, devolvé el listItems de tu componente:

```
return <ul>{listItems}</ul>;
```

Nuevamente temenos un error de consola igual al anterior

```
Warning: Each child in a list should have a unique "key" prop.

Check the render method of `List`. See https://reactjs.org/link/warning-keys for more information.
    at li
    at List
```

Para tener en cuenta, las funciones de flecha implícitamente devuelven la expresión justo después del =>, así que no necesitas declarar un return:

```
const listItems = chemists.map(person =>
  <li>...</li> // Implicit return!
);
```

Sin embargo, debemos escribir el return explícitamente si nuestra => está seguida por una llave{

```
const listItems = chemists.map(person => { // Curly brace
  return <li>...</li>;
});
```

Las funciones de flecha que tienen => { se dice que tienen un «cuerpo de bloque». Nos permiten escribir más de una sola línea de código, pero tenemos que declarar un return por nuestra cuenta.

10.Mantener los elementos de una lista en orden con key

Vimos que anteriormente teníamos una advertencia en la consola (Warning)

Esto se debe a que tenemos que darle a cada elemento del array una key, una cadena de texto o un número que lo identifique de manera única entre otros elementos del array:

```
<li key={person.id}>...</li>
```

Los elementos JSX directamente dentro de una llamada a un map() siempre necesitan keys.

Las keys le indican a React que objeto del array corresponde a cada componente, para así poder emparejarlo más tarde. Esto se vuelve más importante si los objetos de nuestros arrays se pueden mover (p. ej. debido a un ordenamiento), insertar, o eliminar. Una key bien escogida ayuda a React a entender lo que ha sucedido exactamente, y hacer las correctas actualizaciones en el árbol del DOM.

En vez de generar keys sobre la marcha, deberíamos incluirlas en nuestros datos:

```
export const people = [{
  id: 0, // Used in JSX as a key
  name: 'Creola Katherine Johnson',
  profession: 'mathematician',
  accomplishment: 'spaceflight calculations',
  imageId: 'MK3eW3A'
}, {
  id: 1, // Used in JSX as a key
  name: 'Mario José Molina-Pasquel Henríquez',
  profession: 'chemist',
  accomplishment: 'discovery of Arctic ozone hole',
  imageId: 'mynHUSa'
}, {
  id: 2, // Used in JSX as a key
  name: 'Mohammad Abdus Salam',
  profession: 'physicist',
  accomplishment: 'electromagnetism theory',
  imageId: 'bE7W1ji'
}, {
```

```
  id: 3, // Used in JSX as a key
  name: 'Percy Lavon Julian',
  profession: 'chemist',
  accomplishment: 'pioneering cortisone drugs, steroids and
birth control pills',
  imageId: 'IOjWm71'
}, {
  id: 4, // Used in JSX as a key
  name: 'Subrahmanyan Chandrasekhar',
  profession: 'astrophysicist',
  accomplishment: 'white dwarf star mass calculations',
  imageId: 'lrWQx8l'
}];
```

11.Keys

Distintas fuentes de datos dan diferentes fuentes de keys:

- Datos de una base de datos: Si los datos vienen de una base de datos, puedes usar las keys/IDs de la base de datos, que son únicas por naturaleza.
- Datos generados localmente: Si los datos son generados y persistidos localmente (p. ej. notas en una app de tomar notas), usa un contador incremental, crypto.randomUUID() o un paquete como uuid cuando este creando objetos.

12.Reglas de las keys

Las keys tienen que ser únicas entre elementos hermanos. Sin embargo, está bien usar las mismas keys para nodos JSX en arrays diferentes.

Las keys no tienen que cambiar nunca. No las generes mientras renderizas.

¿Por qué React necesita keys?

Imagina que los archivos de tu escritorio no tuvieran nombres. En vez de eso, tu te referirías a ellos por su orden — el primer archivo, el segundo, y así. Podrías acostumbrarte a ello, pero una vez borres un archivo, se volvería algo confuso. El segundo archivo se convertiría en el primero, el tercer archivo se convertiría en el segundo, y así.

Los nombres de archivos en una carpeta y las keys JSX en un array tienen un propósito similar. Nos permiten identificar un objeto de manera única entre sus hermanos. Una key bien escogida da más información aparte de la posición en el array. incluso si la posición cambia debido a un reordenamiento, la key permite a React identificar al elemento a lo largo de su ciclo de vida.

Podrías estar tentado a usar el índice del elemento en el array como su key. De hecho, eso es lo que React usará si tu no específicas un key en absoluto. Pero el orden en el que renderizas elementos cambiará con el tiempo si un elemento es insertado, borrado, o si se reordena su array. El índice como key lleva a menudo a sutiles y confusos errores.

Igualmente, no generes keys sobre la marcha, p. ej. con key={Math.random()}. Esto hará que las keys nunca coincidan entre renderizados, llevando a todos tus componentes y al DOM a recrearse cada vez. No solo es una manera lenta, sino que también pierde cualquier input del usuario dentro de los elementos listados. En vez de eso, usa unas IDs basadas en datos.

Date cuenta de que tus componentes no reciben la key como un prop. Solo es usado como pista para React. Si tus componentes necesitan un ID, se lo tienes que pasar como una prop separada: <Profile key={id} userId={id} />.