Capitulo 5. Binding de una via en React

El uso de { }

5.1 Binding de contenido y atributos

JSX permite escribir marcas similares a HTML dentro de un archivo JavaScript, manteniendo la lógica de representación y el contenido en el mismo lugar. A veces se desea agregar un poco de lógica de JavaScript o hacer referencia a una propiedad dinámica dentro de ese marcado. En esta situación, se puede usar llaves ({ }) en su JSX para crear una conexión a JavaScript.

5.1 Binding de contenido y atributos

Se puede conectar como contenido de nuestro template usando la sintaxis de {}, tal como se muestra:

```
const name = 'Eduardo';

  return (

    <h1>{name}'s To Do List</h1>

  );
```

5.1 Binding de contenido y atributos

Se puede conectar como valor de los atributos del elemento HTML usando la sintaxis de {}, tal como se muestra:

```
const avatar = 'https://i.imgur.com/7vQD0fPs.jpg';
const name= 'Eduardo';
return (
  <img
    className="avatar"
    src={avatar}
    alt={name}
  />
);
```

5.2 Binding de datos de un objeto

Para pasar datos obtenidos de un objeto como contenido de un elemento HTML se sigue la sintaxis de objeto.atributo encerrados en llaves {}

```
const person = {
 name: 'Eduardo',
  theme: {
    backgroundColor: 'black',
    color: ‘green’
  }
};
```

```
<div style={person.theme}>
    <h1>{person.name}</h1>
    <img
     className="avatar"
     src="https://i.imgur.com/7vQD0fPs.jpg"
     alt="Eduardo"
    />
    <ul>
     <li>Programador Full Stack</li>
     <li>Asesor en Cadif1</li>
     <li>Creador AvilaPro Foundation</li>
    </ul>
  </div>
```

5.2 Binding de datos de un objeto

Pasar datos de un objetos como valores de atributos de un elemento HTML se realiza con la sintaxis de dobles llaves {{ }}.

```
<ul style={{
  backgroundColor: ‘blue’,
  color: ‘white’
}}>
  <li>Programador Full Stack</li>
  <li>Asesor en Cadif1</li>
  <li>Creador de AvilaPro Foundation</li>
</ul>
```

Capitulo 6. Pasar props a un componente.

El uso de ({ })

6.1 Paso de props de un componente a otro.

Los componentes de React utilizan *props* para comunicarse entre sí. Cada componente principal puede pasar cierta información a sus componentes secundarios dándoles props. Los props pueden parecerse a los atributos HTML, pero puede pasar cualquier valor de JavaScript a través de ellos, incluidos objetos, matrices y funciones.

6.1 Paso de props de un componente a otro.

Para pasar props al componente secundario se usa la misma sintaxis del binding de una vía, solo que en este caso el elemento que recibe la información es otro componente de React.

```
export default function Profile() {
return (
  <Avatar persona={{ name: 'Eduardo',
imageId: '1bX5QH6' }} size={100} />
);
}
```

6.2 Leer los props en el componente receptor.

Puede leer los props enumerando sus nombres separados por comas dentro de `({ })` directamente después *function NombreComponente*. Esto le permite usarlos dentro del código, como lo haría con una variable.

```
function Avatar({ person, size }) {

}
```

A esta sintaxis ({ }) se le llama desestructuración. La desestructuración es una expresión de JavaScript que permite desempacar valores de arreglos o propiedades de objetos en distintas variables.

6.3 Props con valor por defecto.

Si desea darle a un prop un valor predeterminado al que recurrir cuando no se especifica ningún valor, puede hacerlo con la desestructuración colocando = y el valor predeterminado justo después del parámetro:

```
function Avatar({ person, size = 100 })
{
// ...
}
```

El valor predeterminado solo se usa se falta el prop o si se pasa un valor undefined.

Capitulo 7. Renderizado Condicional.

El uso de if, && y ? :

7.1 JSX condicionados por if

A menudo, sus componentes necesitarán mostrar cosas diferentes dependiendo de las diferentes condiciones. En React, puede renderizar JSX de forma condicional utilizando la sintaxis de JavaScript, como declaraciones if, operadores lógicos && y operadores ternarios ? :

7.1 JSX condicionados por if

Es posible determinar que plantillas JSX se va a renderizar usando la sentencia if:

```
if (esEspecial) {
   return <li className="item">{name}-Especial</li>;
}else{
   return <li className="item">{name}</li>;
}
```

7.1 JSX condicionados por if

En caso de no querer renderizar nada, se puede devolver null:

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

7.2 Condicionado con el operador ternario ? :

Podemos usar el operador ternario para determinar que parte del contenido JSX deseamos mostrar dada una condicion.

```
return (
  <li className="item">
    {esEspecial ? 'Super' + name : name}
  </li>
);
```

7.2 Condicionado con el operador ternario ? :

Con el operador ternario podemos incluso determinar JSX completo como respuesta.

```
return (
 <li className="item">
  {esEspecial ?
     (<p class=”especial”> name </p> ) :
     (<span>Nada especial</span>)
   }
</li>
);
```

7.3 Uso del operador lógico && :

Otra forma de condicionar el contenido de un elemento es con el uso de &&.

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

Capitulo 8. Renderizado de Arreglos.

El uso de map() y filter() :

8.1 Representar datos de una matriz

A menudo querrás mostrar varios componentes similares de un arreglo de datos. Puede utilizar los métodos de arreglos de JavaScript para manipular un arreglo de datos. Se usará filter() y map() con React para filtrar y transformar su conjunto de datos en un conjunto de componentes.

8.1 Representar datos de una matriz

Para la representación se necesita de una matriz, la asignación de una variable el resultado de “mapear” el arreglo y agregar dicha variable al binding { } en alguna parte de la plantilla.

8.1 Representar datos de una matriz

Ejemplo:

```
var arreglo = [“a”, “b”, “c”];

export default function List() {
  const listaDeArreglo= arreglo.map(el =>
    <li>{el}</li>
  );
  return <ul>{ listaDeArreglo }</ul>;
```

8.1 Representar datos de una matriz

Esto generará como resultado una lista como la siguiente:

- a
- b
- c

Pero en la consola se generará un warning como el siguiente:

8.1 Representar datos de una matriz

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

Aprenderemos a solventar esta advertencia a continuación.

8.2 Orden del array con la prop “key”

Los elementos JSX directamente dentro de una map() siempre necesitan key!

8.2 Orden del array con la prop “key”

Las “keys” le indican a React a qué elemento del arreglo corresponde cada componente, para que pueda compararlos más adelante. Esto se vuelve importante si los elementos de su arreglo se pueden mover (por ejemplo, debido a la clasificación), insertarse o eliminarse. Una buena elección de “key” ayuda a React a inferir qué sucedió exactamente y realizar las actualizaciones correctas en el árbol DOM.

8.2 Orden del array con la prop “key”

Por lo que se recomienda que si los datos son generados manualmente proporcionar una propiedad única a cada elemento que contenga dicha “key” en otras palabras, convertir el arreglo de valores en un arreglo de objetos que tienen entre otras propiedades, el atributo que se usará como “key” en el renderizado. Esto es más intuitivo en datos que se obtienen de una base de datos porque estos suelen incluir el atributo “id” que sirve para este fin perfectamente.

8.2 Orden del array con la prop “key”

Es así que nuestro arreglo quedaría:

```
var arreglo = [

{ id: 1, valor: “a”},

{ id: 2, valor: “b”},

{ id: 3, valor: “c”},

]
```

8.2 Orden del array con la prop “key”

Y su renderizado corregido:

```
export default function List() {
  const listaDeArreglo= arreglo.map(el =>
    <li key= {el.id}>{ el.valor }</li>
  );
  return <ul>{ listaDeArreglo }</ul>;
```

8.3 Filtrar elementos de un arreglo.

Supongamos que nuestros datos tienen alguna propiedad en particular que los puede catalogar entre ellos. Se puede usar el método filter() para “filtrar” un grupo determinado de elementos que comparten un valor de dicha propiedad.

8.3 Filtrar elementos de un arreglo.

Supongamos nuestros datos:

```
var arreglo = [

{ id: 1, valor: “a”, important: true},

{ id: 2, valor: “b”, important: false},

{ id: 3, valor: “c”, important: true},

]
```

8.3 Filtrar elementos de un arreglo.

Y su renderizado corregido:

```
export default function List() {
  const arregloFiltrado= arreglo.filter(el =>
    el.important === true
  );
```

Esto devolverá un nuevo arreglo cuyos ítems solo serán los que cumplen la condición.

8.3 Filtrar elementos de un arreglo.

Seguido de esto se puede “mapear” dicho arreglo y renderizar luego.

```
const listaDeArrreglo= arregloFiltrado.map(el =>
  <li key= { el.id } >
    { el.valor }
  </li>
);
return <ul> { listaDeArreglo } </ul>
```

8.4 Uso de Fragment

Si se necesita representar no uno, sino varios nodos DOM; la sintaxis corta <>...</> de Fragmento no le permitirá pasar una clave, por lo que debe agruparlos en un solo elemento o usar la sintaxis un poco más larga y explícita : <Fragment>.

8.4 Uso de Fragment

```
export default function List() {

  const listaDeArreglo= arreglo.map(el =>

    <Fragment key= {el.id}>

        <span> Numero: { el.id }</span>

        <h1> Nombre: { el.valor } </h1>

    </Fragment>

  );
```

Capitulo 9. Manejadores de Eventos en React.

Funciones “handle”

9.1 Agregar controlador de evento

React permite agregar controladores de eventos a JSX. Los controladores de eventos son funciones propias que se activarán en respuesta a interacciones como hacer clic, desplazarse, centrarse en las entradas del formulario, etc.

9.1 Agregar controlador de evento

Para agregar un controlador de eventos, primero se define una función y luego se pasa como “prop” a la etiqueta JSX adecuada.

Ahora bien las funciones del controlador de eventos deberían:

- Generalmente se definen *dentro* de sus componentes.
- Tener nombres que comiencen con handle seguido del nombre del evento.

9.1 Agregar controlador de evento

Por convención, es común nombrar los controladores de eventos como handle seguidos del nombre del evento. A menudo se puede ver:

onClick={handleClick},

onMouseEnter={handleMouseEnter}, etc.

9.1 Agregar controlador de evento

Alternativamente, puede definir un controlador de eventos en línea en JSX:

```
<button onClick={function handleClick() {
  alert('You clicked me!');
}}>
```

9.1 Agregar controlador de evento

O, más concisamente, usando una arrow function:

```
<button onClick={() => {
  alert('You clicked me!');
}}>
```

9.2 Lectura de props en controlador de eventos

Debido a que los controladores de eventos se declaran dentro de un componente, tienen acceso a los props del componente.

9.2 Lectura de props en controlador de eventos

```
function AlertButton({ message, children }) {
  return (
    <button onClick={() => alert(message)}>
      {children}
    </button>
  );
}
```

9.2 Lectura de props en controlador de eventos

```
<AlertButton message="Playing!">
  Play Movie
</AlertButton>
<AlertButton message="Uploading!">
  Upload Image
</AlertButton>
```

9.3 Pasar controladores de eventos como props

A menudo puede requerir que el componente principal especifique un controlador de eventos secundario.

Considere los botones: dependiendo de dónde esté usando un componente “Button”, es posible que desee ejecutar una función diferente.

Para hacer esto, pase un prop que el componente recibe de su padre como controlador de eventos.

9.3 Pasar controladores de eventos como props

```
function Button({ onClick, children }) {
  return (
    <button onClick={onClick}>
      {children}
    </button>
  );
}
```

9.3 Pasar controladores de eventos como props

```
function PlayButton({ movieName }) {
  function handlePlayClick() {
    alert(`Playing ${movieName}!`);
  }
  return (
    <Button onClick={handlePlayClick}>
      Play "{movieName}"
    </Button>
  );}
```

9.3 Pasar controladores de eventos como props

```
function UploadButton() {
  return (
    <Button onClick={() => alert('Uploading!')}>
      Upload Image
    </Button>
  );
}
```

9.3 Pasar controladores de eventos como props

```
export default function Toolbar() {
  return (
    <div>
      <PlayButton movieName="Kiki's Delivery Service" />
      <UploadButton />
    </div>
  );}
```

Capitulo 10. State: La memoria de un componente

Uso del hook: useState

10.1 Hooks en React

Los hooks te permiten usar diferentes funciones de React de tus componentes. Puedes usar los hooks incorporados o combinarlos para construir los tuyos propios. Vamos a enumerar todos los Hooks integrados en React.

10.1 Hooks en React

State Hook

El estado permite que un componente "recuerde" información como la entrada del usuario. Por ejemplo, un componente de formulario puede usar el estado para almacenar el valor de entrada, mientras que un componente de galería de imágenes puede usar el estado para almacenar el índice de la imagen seleccionada.

Para agregar estado a un componente, use uno de estos Hooks:

- useState declara una variable de estado que puede actualizar directamente.
- useReducer declara una variable de estado con la lógica de actualización dentro de una “función reductora”.

10.1 Hooks en React

Context Hook

El contexto permite que un componente reciba información de padres lejanos sin pasarla como accesorios. Por ejemplo, el componente de nivel superior de su aplicación puede pasar el tema de la interfaz de usuario actual a todos los componentes siguientes, sin importar cuán profundo sea.

- useContext lee y se suscribe a un contexto.

10.1 Hooks en React

Ref Hook

Las referencias permiten que un componente contenga información que no se utiliza para la renderización, como un nodo DOM o un ID de tiempo de espera. A diferencia del estado, actualizar una referencia no vuelve a representar su componente. Las referencias son una "escotilla de escape" del paradigma de React. Son útiles cuando necesita trabajar con sistemas que no son React, como las API del navegador integradas.

- useRef declara un árbitro. Puede contener cualquier valor, pero la mayoría de las veces se usa para contener un nodo DOM.
- useImperativeHandle le permite personalizar la referencia expuesta por su componente. Esto rara vez se usa.

10.1 Hooks en React

Effects Hook

Los efectos permiten que un componente se conecte y sincronice con sistemas externos. Esto incluye lidiar con la red, el DOM del navegador, animaciones, widgets escritos usando una biblioteca de UI diferente y otro código que no sea de React.

- useEffect conecta un componente a un sistema externo.

10.1 Hooks en React

Performance Hook

Una forma común de optimizar el rendimiento de la renderización es omitir el trabajo innecesario. Por ejemplo, puede decirle a React que reutilice un cálculo almacenado en caché o que omita una nueva renderización si los datos no han cambiado desde la renderización anterior.

Para omitir cálculos y renderizaciones innecesarias, utilice uno de estos hooks:

- useMemo le permite almacenar en caché el resultado de un cálculo costoso.
- useCallback le permite almacenar en caché una definición de función antes de pasarla a un componente optimizado.

10.1 Hooks en React

Otros Hook

Estos hooks son principalmente útiles para los autores de bibliotecas y no se usan comúnmente en el código de la aplicación.

- useDebugValue le permite personalizar la etiqueta que muestra React DevTools para su Hook personalizado.
- useId permite que un componente asocie una identificación única consigo mismo. Normalmente se utiliza con API de accesibilidad.
- useSyncExternalStore permite que un componente se suscriba a una store externa.

10.2 Agregar una variable de estado.

Los componentes a menudo necesitan cambiar lo que aparece en la pantalla como resultado de una interacción. Escribir en el formulario debería actualizar el campo de entrada, hacer clic en "siguiente" en un carrusel de imágenes debería cambiar la imagen que se muestra, hacer clic en "comprar" debería colocar un producto en el carrito de compras. Los componentes necesitan "recordar" cosas: el valor de entrada actual, la imagen actual, el carrito de compras. En React, este tipo de memoria específica de componente se llama *estado* .

10.2 Agregar una variable de estado.

Una variable regular no es suficiente por las siguientes razones:

1. **Las variables regulares no persisten entre renderizaciones.** Cuando React renderiza este componente por segunda vez, lo renderiza desde cero; no considera ningún cambio en las variables locales.
2. **Los cambios en las variables regulares no activarán las renderizaciones.** React no se da cuenta de que necesita renderizar el componente nuevamente con los nuevos datos.

10.2 Agregar una variable de estado.

Para actualizar un componente con nuevos datos, deben suceder dos cosas:

1. **Conservar** los datos entre renderizados.
2. **Activar** React para renderizar el componente con nuevos datos (volver a renderizar).

10.2 Agregar una variable de estado.

El Hook useState proporciona esas dos cosas:

1. Una **variable de estado** para retener los datos entre renderizaciones.
2. Una **función de establecimiento de estado** para actualizar la variable y activar React para renderizar el componente nuevamente.

10.2 Agregar una variable de estado.

Para agregar una variable de estado, importe *useState* desde React en la parte superior del archivo:

import { useState } from 'react';

Luego, reemplace esta línea:

let index = 0;

con

const [index, setIndex] = useState(0);

10.3 Conociendo useState

La convención es nombrar este par como:

const [varEstado, setVarEstado].

Puedes nombrarlo como quieras, pero las convenciones hacen que las cosas sean más fáciles de entender en todos los proyectos.

10.3 Conociendo useState

El único argumento de useState es el **valor inicial** de su variable de estado. En este ejemplo, el valor inicial de *index* se establece en 0 con useState(0).

```
const [index, setIndex] = useState(0);
```

Cada vez que su componente se procesa, useState le proporciona una matriz que contiene dos valores:

1. La **variable de estado** (index) con el valor que almacenó.
2. La **función de establecimiento de estado** (setIndex) que puede actualizar la variable de estado y activar React para renderizar el componente nuevamente.

10.3 Conociendo useState

La sintaxis [y] usada en useState se llama desestructuración de matrices y le permite leer valores de una matriz. La matriz devuelta por useState siempre tiene exactamente dos elementos. Ejemplo del uso del set:

```
function handleClick() {
 setIndex(index + 1);
}
```

Capitulo 11. Actualizacion de Objetos

Object Update

11.1 Mutabilidad

El estado puede contener cualquier tipo de valor de JavaScript, incluidos los objetos. Pero no deberías cambiar los objetos que tienes en el estado React directamente.

En cambio, cuando desee actualizar un objeto, deberá crear uno nuevo (o hacer una copia de uno existente) y luego configurar el estado para usar esa copia.

11.1 Mutabilidad

Consideremos ahora un objeto en estado:

const [position, setPosition] = useState({ x: 0, y: 0 });

Técnicamente, es posible cambiar el contenido del propio objeto . Esto se llama mutación:

position.x = 5;

11.1 Mutabilidad

Sin embargo, aunque los objetos en estado React son técnicamente mutables, debes tratarlos como si fueran inmutables, como números, booleanos y cadenas. En lugar de “mutarlos”, siempre debes reemplazarlos.

En otras palabras, debes tratar cualquier objeto JavaScript que pongas en estado como de solo lectura.

11.1 Mutabilidad

Pongamos este ejemplo:

```
onPointerMove={e => {
  position.x = e.clientX;
  position.y = e.clientY;
}}
```

Este código modifica el objeto asignado *position* desde el render anterior. Pero sin utilizar la función de configuración de estado, React no tiene idea de que el objeto ha cambiado. Entonces React no hace nada en respuesta. Es como intentar cambiar el orden después de haber comido. Si bien el estado mutante puede funcionar en algunos casos, no lo recomendamos. Debes tratar el valor de estado al que tienes acceso en un renderizado como de solo lectura.

11.1 Mutabilidad

Para activar realmente una nueva renderización en este caso, cree un *nuevo* objeto y páselo a la función de configuración de estado:

```
onPointerMove={e => {
 setPosition({
   x: e.clientX,
   y: e.clientY
 });
}}
```

11.1 Mutabilidad

Con setPosition, le estás diciendo a React:

- Reemplazar position con este nuevo objeto.
- Y renderice este componente nuevamente.

11.2 Actualización de Objeto

La forma confiable de obtener el comportamiento que busca es crear un nuevo objeto y pasárselo. Por ejemplo:

```
setPerson({
    firstName: e.target.value,
    lastName: person.lastName,
    email: person.email
});
```

11.2 Actualización de Objeto

Puede utilizar la Spread syntax (…) de objetos para no tener que copiar cada propiedad por separado.

```
setPerson({
    ...person, // Copia los campos del objeto
    firstName: e.target.value // Sobreescribe el valor que desea editar
});
```

11.2 Actualización de Objeto

Ejemplo de formulario que actualiza el objeto:

```
<label>
    First name:
    <input
      value={person.firstName}
      onChange={handleFirstNameChange}
    />
</label>
```

11.2 Actualización de Objeto

Tenga en cuenta que la ...sintaxis extendida es "superficial": solo copia cosas en un nivel de profundidad. Esto lo hace rápido, pero también significa que si desea actualizar una propiedad anidada, tendrá que usarla más de una vez.

11.3 Actualización de Objeto Anidado

Considere una estructura de objeto anidada como esta:

```
const [person, setPerson] = useState({
 name: 'Niki de Saint Phalle',
 artwork: {
   title: 'Blue Nana',
   city: 'Hamburg',
   image: 'https://i.imgur.com/Sd1AgUOm.jpg',
 }
});
```

11.3 Actualización de Objeto Anidado

Considere una estructura de objeto anidada como esta:

```
const [person, setPerson] = useState({
 name: 'Niki de Saint Phalle',
 artwork: {
   title: 'Blue Nana',
   city: 'Hamburg',
   image: 'https://i.imgur.com/Sd1AgUOm.jpg',
 }
});
```

Cómo actualizar la propiedad city?

11.3 Actualización de Objeto Anidado

La siguiente es una forma:

```
setPerson({
    ...person, //Copia los campos anteriores
    artwork: { // Reemplaza este objeto
        ...person.artwork, // con los mismos valores
    anteriores
        city: 'New Delhi' // pero nuevo valor de city.
    }
});
```

11.4 Actualización de Objeto Anidado con Immer

Si su estado está profundamente anidado, es posible que desee considerar “aplanarlo”. Pero, si no desea cambiar la estructura de su estado, es posible que prefiera un atajo a los diferenciales anidados. *Immer* es una biblioteca popular que te permite escribir usando la sintaxis conveniente pero cambiante y se encarga de producir las copias por ti. Con *Immer*, el código que escribes parece como si estuvieras "rompiendo las reglas" y mutando un objeto.

11.4 Actualización de Objeto Anidado con Immer

Esto es un cambio con Immer:

```
updatePerson(draft => {
    draft.artwork.city = 'Lagos';
});
```

11.4 Actualización de Objeto Anidado con Immer

Para instalar y usar *Immer*:

1. Ejecute npm install use-immer para agregar Immer como dependencia
2. Luego reemplace import { useState } from 'react' con import { useImmer } from 'use-immer'

11.4 Actualización de Objeto Anidado con Immer

Un ejemplo del uso de Immer:

```
const [person, updatePerson] = useImmer({
    name: 'Niki de Saint Phalle',
    artwork: {
      title: 'Blue Nana',
      city: 'Hamburg',
      image: 'https://i.imgur.com/Sd1AgUOm.jpg',
    }
  });
```

11.4 Actualización de Objeto Anidado con Immer

Ejemplo de manejadores de eventos con Immer:

```
function handleNameChange(e) {
    updatePerson(draft => {
      draft.name = e.target.value;
    });
  }
```

11.4 Actualización de Objeto Anidado con Immer

Ejemplo de manejadores de eventos con Immer:

```
function handleTitleChange(e) {
    updatePerson(draft => {
      draft.artwork.title = e.target.value;
    });
  }
```