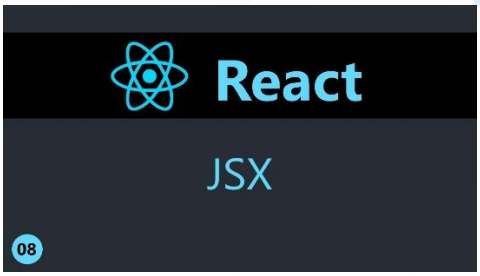


Ing. Luis Guillermo Molero Suárez

Presentando JSX

JSX, es una extensión de la sintaxis de JavaScript. Recomendamos usarlo con React para describir cómo debería ser la interfaz de usuario. JSX puede recordarte a un lenguaje de plantillas, pero viene con todo el poder de JavaScript. JSX produce “elementos” de React.

¿Por qué JSX?

React acepta el hecho de que la lógica de renderizado está intrínsecamente unida a la lógica de la interfaz de usuario: cómo se manejan los eventos, cómo cambia el estado con el tiempo y cómo se preparan los datos para su visualización.

En lugar de separar artificialmente tecnologías poniendo el maquetado y la lógica en archivos separados, React separa intereses con unidades ligeramente acopladas llamadas “componentes” que contienen ambas. Volveremos a los componentes en otra sección, pero si aún no te sientes cómodo maquetando en JS, esta charla podría convencerte de lo contrario.

React no requiere usar JSX, pero la mayoría de la gente lo encuentra útil como ayuda visual cuando trabajan con interfaz de usuario dentro del código Javascript. Esto también permite que React muestre mensajes de error o advertencia más útiles.

Insertando expresiones en JSX

En el ejemplo a continuación, declaramos una variable llamada name y luego la usamos dentro del JSX envolviéndola dentro de llaves:

```
const name = 'Josh Perez';const element = <h1>Hello, {name}</h1>;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Puedes poner cualquier expresión de JavaScript dentro de llaves en JSX. Por ejemplo, 2 + 2, user.firstName, o formatName(user) son todas expresiones válidas de Javascript.

En el ejemplo a continuación, insertamos el resultado de llamar la función de JavaScript,

```
function formatName(user) {
    return user.firstName + ' ' + user.lastName;
  }

  const user = {
    firstName: 'Harper',
    lastName: 'Perez'
  };

  const element = (
    <h1>
      Hello, {formatName(user)}!  </h1>
  );

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

Pruébalo en CodePen

Dividimos JSX en varias líneas para facilitar la lectura. Aunque no es necesario, cuando hagas esto también te recomendamos envolverlo entre paréntesis para evitar errores por la inserción automática del punto y coma.

JSX también es una expresión

Después de compilarse, las expresiones JSX se convierten en llamadas a funciones JavaScript regulares y se evalúan en objetos JavaScript.

Esto significa que puedes usar JSX dentro de declaraciones if y bucles for, asignarlo a variables, aceptarlo como argumento, y retornarlo desde dentro de funciones:

```
function getGreeting(user) {
    if (user) {
      return <h1>Hello, {formatName(user)}!</h1>;  }
    return <h1>Hello, Stranger.</h1>;}
```

Especificando atributos con JSX

Puedes utilizar comillas para especificar strings literales como atributos:

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

También puedes usar llaves para insertar una expresión JavaScript en un atributo:

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

No pongas comillas rodeando llaves cuando insertes una expresión JavaScript en un atributo. Debes utilizar comillas (para los valores de los strings) o llaves (para las expresiones), pero no ambas en el mismo atributo.

Advertencia:

Dado que JSX es más cercano a JavaScript que a HTML, React DOM usa la convención de nomenclatura camelCase en vez de nombres de atributos HTML.

Por ejemplo, class se vuelve className en JSX, y tabindex se vuelve tabIndex.

Especificando hijos con JSX

Si una etiqueta está vacía, puedes cerrarla inmediatamente con />, como en XML:

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

Las etiquetas de Javascript pueden contener hijos:

```
const element = (
  <div>
    <h1>Hello!</h1>
    <h2>Good to see you here.</h2>
  </div>
);
```

JSX previene ataques de inyección

Es seguro insertar datos ingresados por el usuario en JSX:

```
const title = response.potentiallyMaliciousInput;
// Esto es seguro:
const element = <h1>{title}</h1>;
```

Por defecto, React DOM escapa cualquier valor insertado en JSX antes de renderizarlo. De este modo, se asegura de que nunca se pueda insertar nada que no esté explícitamente escrito en tú aplicación. Todo es convertido en un string antes de ser renderizado. Esto ayuda a prevenir vulnerabilidades XSS (cross-site-scripting).

JSX representa objetos

Babel compila JSX a llamadas de React.createElement().

Estos dos ejemplos son idénticos:

```
const element = (
    <h1 className="greeting">
      Hello, world!
    </h1>
  );
  const element = React.createElement(
    'h1',
    {className: 'greeting'},
    'Hello, world!'
  );
```

React.createElement() realiza algunas comprobaciones para ayudarte a escribir código libre de errores, pero, en esencia crea un objeto como este:

```
// Nota: Esta estructura está simplificada
const element = {
    type: 'h1',
    props: {
      className: 'greeting',
      children: 'Hello, world!'
    }
  };
```

Estos objetos son llamados “Elementos de React”. Puedes pensar en ellos como descripciones de lo que quieres ver en pantalla. React lee estos objetos y los usa para construir el DOM y mantenerlo actualizado.

Vamos a explorar el renderizado de los elementos de React al DOM en la siguiente sección.

Renderizando elementos

Los elementos son los bloques más pequeños de las aplicaciones de React.
Un elemento describe lo que quieres ver en la pantalla:

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

A diferencia de los elementos del DOM de los navegadores, los elementos de React son objetos planos, y su creación es de bajo costo. React DOM se encarga de actualizar el DOM para igualar los elementos de React.

Nota:

Uno podría confundir los elementos con el muy conocido concepto de “componentes”. En la siguiente sección hablaremos de componentes. Los elementos son los que “constituyen” los componentes, y recomendamos leer esta sección antes de continuar.

Renderizando un elemento en el DOM

Digamos que hay un en alguna parte de tu archivo HTML:

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

Lo llamamos un nodo “raíz” porque todo lo que esté dentro de él será manejado por React DOM.

Las aplicaciones construidas solamente con React usualmente tienen un único nodo raíz en el DOM. Dado el caso que estés integrando React en una aplicación existente, puedes tener tantos nodos raíz del DOM aislados como quieras.

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

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

Actualizando el elemento renderizado

Los elementos de React son inmutables. Una vez creas un elemento, no puedes cambiar sus hijos o atributos. Un elemento es como un fotograma solitario en una película: este representa la interfaz de usuario en cierto punto en el tiempo.

Con nuestro conocimiento hasta este punto, la única manera de actualizar la interfaz de usuario es creando un nuevo elemento, y pasarlo a ReactDOM.render().

Considera este ejemplo de un reloj en marcha:

```
function tick() {
    const element = (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {new Date().toLocaleTimeString()}.</h2>
      </div>
    );
    ReactDOM.render(element, document.getElementById('root'));}

  setInterval(tick, 1000);
```

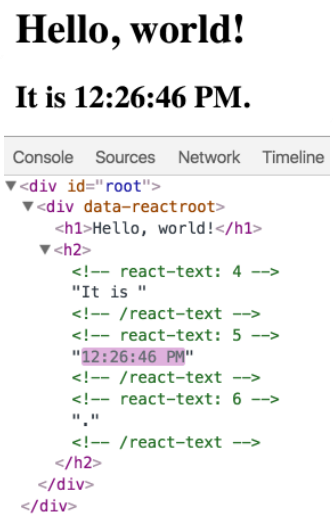
Nota:

En la práctica, la mayoría de las aplicaciones de React solo llaman a ReactDOM.render() una vez. En las siguientes secciones aprenderemos cómo el código se puede encapsular en componentes con estado.

React solo actualiza lo que es necesario

React DOM 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.

Puedes verificar esto inspeccionando el último ejemplo con las herramientas del navegador:



Aunque creamos un elemento que describe el árbol de la interfaz de usuario en su totalidad en cada instante, React DOM solo actualiza el texto del nodo cuyo contenido cambió.

En nuestra experiencia, pensar en cómo la interfaz de usuario debería verse en un momento dado y no en cómo cambiarla en el tiempo, elimina toda una clase de errores.

Componentes y propiedades

Los componentes permiten separar la interfaz de usuario en piezas independientes, reutilizables y pensar en cada pieza de forma aislada. Esta página proporciona una introducción a la idea de los componentes. Puedes encontrar una API detallada sobre componentes aquí.

Conceptualmente, los componentes son como las funciones de JavaScript. Aceptan entradas arbitrarias (llamadas “props”) y devuelven a React elementos que describen lo que debe aparecer en la pantalla.

Componentes funcionales y de clase

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

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

Esta función es un componente de React válido porque acepta un solo argumento de objeto “props” (que proviene de propiedades) con datos y devuelve un elemento de React. Llamamos a dichos componentes “funcionales” porque literalmente son funciones JavaScript.

También puedes utilizar una clase de ES6 para definir un componente:

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

Los dos componentes anteriores son equivalentes desde el punto de vista de React. Tanto los componentes de función como de clase tienen algunas características adicionales que veremos en las próximas secciones.

Renderizando un componente

Anteriormente, sólo encontramos elementos de React que representan las etiquetas del DOM:

```
const element = <div />;
```

Sin embargo, los elementos también pueden representar componentes definidos por el usuario:

```
const element = <Welcome name="Sara" />;
```

Cuando React ve un elemento representando un componente definido por el usuario, pasa atributos JSX e hijos a este componente como un solo objeto. Llamamos a este objeto “props”.

Por ejemplo, este código muestra “Hello, Sara” en la página:

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

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

Nota:

Comienza siempre los nombres de componentes con una letra mayúscula.
React trata los componentes que empiezan con letras minúsculas como etiquetas del DOM. Por ejemplo, representa una etiqueta div HTML pero <Welcome /> representa un componente y requiere que Welcome esté definido.

Composición de componentes

Los componentes pueden referirse a otros componentes en su salida. Esto nos permite utilizar la misma abstracción de componente para cualquier nivel de detalle. Un botón, un cuadro de diálogo, un formulario, una pantalla: en aplicaciones de React, todos son expresados comúnmente como componentes.

Por ejemplo, podemos crear un componente App que renderiza Welcome muchas veces:

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

  function App() {
    return (
      <div>
        <Welcome name="Sara" />      <Welcome name="Cahal" />
 <Welcome name="Edite" />    </div>
    );
  }

  ReactDOM.render(
    <App />,
    document.getElementById('root')
  );
```

Extracción de componentes

No tengas miedo de dividir los componentes en otros más pequeños.

Por ejemplo, considera este componente Comment:

```
function Comment(props) {
    return (
      <div className="Comment">
        <div className="UserInfo">
          <img className="Avatar"
            src={props.author.avatarUrl}
            alt={props.author.name}
          />
          <div className="UserInfo-name">
            {props.author.name}
          </div>
        </div>
        <div className="Comment-text">
          {props.text}
        </div>
        <div className="Comment-date">
          {formatDate(props.date)}
        </div>
      </div>
    );
  }
```

Pruébalo en CodePen

Acepta author (un objeto), text (un string), y date (una fecha) como props, y describe un comentario en una web de redes sociales.

Este componente puede ser difícil de cambiar debido a todo el anidamiento, y también es difícil reutilizar partes individuales de él. Vamos a extraer algunos componentes de este.

Primero, vamos a extraer Avatar:

```
function Avatar(props) {
    return (
      <img className="Avatar"       src={props.user.avatarUrl}
 alt={props.user.name}    />  );
  }
```

El Avatar no necesita saber que está siendo renderizado dentro de un Comment. Esto es por lo que le dimos a su propiedad un nombre más genérico: user en vez de author.

Recomendamos nombrar las props desde el punto de vista del componente, en vez de la del contexto en el que se va a utilizar.

Ahora podemos simplificar Comment un poquito:

```
function Comment(props) {
    return (
      <div className="Comment">
        <div className="UserInfo">
          <Avatar user={props.author} />        <div className="Us
erInfo-name">
            {props.author.name}
          </div>
        </div>
        <div className="Comment-text">
          {props.text}
        </div>
        <div className="Comment-date">
          {formatDate(props.date)}
        </div>
      </div>
    );
  }
```

A continuación, vamos a extraer un componente UserInfo que renderiza un Avatar al lado del nombre del usuario:

```
function UserInfo(props) {
    return (
      <div className="UserInfo">      <Avatar user={props.user} />
      <div className="UserInfo-
name">        {props.user.name}      </div>    </div>  );
  }
```

Esto nos permite simplificar Comment aún más:

```
function Comment(props) {
    return (
      <div className="Comment">
        <UserInfo user={props.author} />      <div className="Comm
ent-text">
          {props.text}
        </div>
        <div className="Comment-date">
          {formatDate(props.date)}
        </div>
      </div>
    );
  }
```

Las props son de solo lectura

Ya sea que declares un componente como una función o como una clase, este nunca debe modificar sus props. Considera esta función sum :

```
function sum(a, b) {
    return a + b;
  }
```

Tales funciones son llamadas “puras” porque no tratan de cambiar sus entradas, y siempre devuelven el mismo resultado para las mismas entradas.

En contraste, esta función es impura porque cambia su propia entrada:

```
function withdraw(account, amount) {
    account.total -= amount;
  }
```

React es bastante flexible pero tiene una sola regla estricta:

Todos los componentes de React deben actuar como funciones puras con respecto a sus props.

Por supuesto, las interfaces de usuario de las aplicaciones son dinámicas y cambian con el tiempo. En la siguiente sección, introduciremos un nuevo concepto de “estado”. El estado le permite a los componentes de React cambiar su salida a lo largo del tiempo en respuesta a acciones del usuario, respuestas de red y cualquier otra cosa, sin violar esta regla.

Estado y ciclo de vida

Esta página introduce el concepto de estado y ciclo de vida en un componente de React.

Consideremos el ejemplo del reloj de una de las secciones anteriores. En Renderizando elementos, aprendimos solo una forma de actualizar la interfaz de usuario. Invocamos a ReactDOM.render() para que cambie el resultado renderizado.

```
function tick() {
    const element = (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {new Date().toLocaleTimeString()}.</h2>
      </div>
    );
    ReactDOM.render(    element,    document.getElementById('root'
)  );}

  setInterval(tick, 1000);
```

En esta sección, aprenderemos como hacer al componente Clock verdaderamente reutilizable y encapsulado. Configurarás tu propio temporizador y se actualizará cada segundo.

Podemos comenzar por encapsular cómo se ve el reloj:

```
function Clock(props) {
    return (
      <div>      <h1>Hello, world!</h1>      <h2>It is {props.date
.toLocaleTimeString()}.</h2>    </div>  );
  }

  function tick() {
    ReactDOM.render(
      <Clock date={new Date()} />,    document.getElementById('roo
t')
    );
  }

  setInterval(tick, 1000);
```

Sin embargo, se pierde un requisito crucial: el hecho de que Clock configure un temporizador y actualice la interfaz de usuario cada segundo debe ser un detalle de implementación de Clock.

Idealmente, queremos escribir esto una vez y que Clock se actualice a sí mismo:

```
ReactDOM.render(
    <Clock />,  document.getElementById('root')
  );
```

Para implementar esto, necesitamos agregar «estado» al componente Clock.
El estado es similar a las props, pero es privado y está completamente controlado por el componente.

Convertir una función en una clase

Se puede convertir un componente de función como Clock en una clase en cinco pasos:

1. Crear una clase ES6 con el mismo nombre que herede de React.Component.
2. Agregar un único método vacío llamado render().
3. Mover el cuerpo de la función al método render().
4. Reemplazar props con this.props en el cuerpo de render().
5. Borrar el resto de la declaración de la función ya vacía.

```
class Clock extends React.Component {
    render() {
      return (
        <div>
          <h1>Hello, world!</h1>
          <h2>It is {this.props.date.toLocaleTimeString()}.</h2>
        </div>
      );
    }
  }
```

Clock ahora se define como una clase en lugar de una función.

El método render se invocará cada vez que ocurre una actualización; pero, siempre y cuando rendericemos <Clock /> en el mismo nodo del DOM, se usará solo una única instancia de la clase Clock. Esto nos permite utilizar características adicionales como el estado local y los métodos de ciclo de vida.

Agregar estado local a una clase

Moveremos date de las props hacia el estado en tres pasos:

1. Reemplazar this.props.date con this.state.date en el método render():

```
class Clock extends React.Component {
    render() {
      return (
        <div>
          <h1>Hello, world!</h1>
          <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
    </div>
      );
    }
  }
```

2. Añadir un constructor de clase que asigne el this.state inicial:

```
class Clock extends React.Component {
    constructor(props) {
      super(props);
      this.state = {date: new Date()};  }

    render() {
      return (
        <div>
          <h1>Hello, world!</h1>
          <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
        </div>
      );
    }
 }
```

Nota cómo pasamos props al constructor base:

```
 constructor(props) {
    super(props);     this.state = {date: new Date()};
  }
```

Los componentes de clase siempre deben invocar al constructor base con props.

3. Eliminar la prop date del elemento <Clock />:

```
ReactDOM.render(
    <Clock />,  document.getElementById('root')
  );
```

Posteriormente regresaremos el código del temporizador al propio componente.

El resultado es el siguiente:

```
class Clock extends React.Component {
    constructor(props) {    super(props);    this.state = {date: n
ew Date()};  }
    render() {
      return (
        <div>
          <h1>Hello, world!</h1>
          <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
    </div>
      );
    }
  }

  ReactDOM.render(
    <Clock />,  document.getElementById('root')
  );
```

A continuación, haremos que Clock configure su propio temporizador y se actualice cada segundo.

Agregar métodos de ciclo de vida a una clase

En aplicaciones con muchos componentes, es muy importante liberar recursos tomados por los componentes cuando se destruyen.

Queremos configurar un temporizador cada vez que Clock se renderice en el DOM por primera vez. Esto se llama «montaje» en React.

También queremos borrar ese temporizador cada vez que el DOM producido por Clock se elimine. Esto se llama «desmontaje» en React.

Podemos declarar métodos especiales en la clase del componente para ejecutar algún código cuando un componente se monta y desmonta:

```
class Clock extends React.Component {
    constructor(props) {
      super(props);
      this.state = {date: new Date()};
    }

    componentDidMount() {  }
    componentWillUnmount() {  }
    render() {
      return (
        <div>
          <h1>Hello, world!</h1>
          <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
        </div>
      );
    }
  }
```

Estos métodos son llamados «métodos de ciclo de vida».

El método componentDidMount() se ejecuta después que la salida del componente ha sido renderizada en el DOM. Este es un buen lugar para configurar un temporizador:

```
componentDidMount() {
    this.timerID = setInterval(      () => this.tick(),      1000
  );  }
```

Nota como guardamos el ID del temporizador en this (this.timerID).

Si bien this.props es configurado por el mismo React y this.state tiene un significado especial, eres libre de añadir campos adicionales a la clase manualmente si necesitas almacenar algo que no participa en el flujo de datos (como el ID de un temporizador). Eliminaremos el temporizador en el método de ciclo de vida componentWillUnmount():

```
componentWillUnmount() {
    clearInterval(this.timerID);  }
```

Finalmente, implementaremos un método llamado tick() que el componente Clock ejecutará cada segundo.

Utilizará this.setState() para programar actualizaciones al estado local del componente.

```
class Clock extends React.Component {
    constructor(props) {
      super(props);
      this.state = {date: new Date()};
    }

    componentDidMount() {
      this.timerID = setInterval(
        () => this.tick(),
        1000
      );
    }

    componentWillUnmount() {
      clearInterval(this.timerID);
    }

    tick() {    this.setState({      date: new Date()    });  }
    render() {
      return (
        <div>
          <h1>Hello, world!</h1>
          <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
        </div>
      );
    }
  }

  ReactDOM.render(
    <Clock />,
    document.getElementById('root')
  );
```

Ahora el reloj cambia cada segundo.
Repasemos rápidamente lo que está sucediendo y el orden en que se invocan los métodos:

1. Cuando se pasa <Clock /> a ReactDOM.render(), React invoca al constructor del componente Clock. Ya que Clock necesita mostrar la hora actual, inicializa this.state con un objeto que incluye la hora actual. Luego actualizaremos este estado.
2. React invoca entonces al método render() del componente Clock. Así es como React sabe lo que se debe mostrar en pantalla. React entonces actualiza el DOM para que coincida con la salida del renderizado de Clock.

3. Cuando la salida de Clock se inserta en el DOM, React invoca al método de ciclo de vida componentDidMount(). Dentro de él, el componente Clock le pide al navegador que configure un temporizador para invocar al método tick() del componente una vez por segundo.
4. Cada segundo el navegador invoca al método tick(). Dentro de él, el componente Clock planifica una actualización de la interfaz de usuario al invocar a setState() con un objeto que contiene la hora actual. Gracias a la invocación a setState(), React sabe que el estado cambió e invoca de nuevo al método render() para saber qué debe estar en la pantalla. Esta vez, this.state.date en el método render() será diferente, por lo que el resultado del renderizado incluirá la hora actualizada. Conforme a eso React actualiza el DOM.
5. Si el componente Clock se elimina en algún momento del DOM, React invoca al método de ciclo de vida componentWillUnmount(), por lo que el temporizador se detiene.

Usar el estado correctamente

Hay tres cosas que debes saber sobre setState().

No modifiques el estado directamente

Por ejemplo, esto no volverá a renderizar un componente:

```
// Incorrecto
this.state.comment = 'Hello';
```

En su lugar utiliza setState():

```
// Correcto
this.setState({comment: 'Hello'});
```

El único lugar donde puedes asignar this.state es el constructor.

Las actualizaciones del estado pueden ser asíncronas

React puede agrupar varias invocaciones a setState() en una sola actualización para mejorar el rendimiento.

Debido a que this.props y this.state pueden actualizarse de forma asincrónica, no debes confiar en sus valores para calcular el siguiente estado.

Por ejemplo, este código puede fallar en actualizar el contador:

```
// Incorrecto
this.setState({
    counter: this.state.counter + this.props.increment,
  });
```

Para arreglarlo, usa una segunda forma de setState() que acepta una función en lugar de un objeto. Esa función recibirá el estado previo como primer argumento, y las props en el momento en que se aplica la actualización como segundo argumento:

```
// Correcto
this.setState((state, props) => ({
    counter: state.counter + props.increment
  }));
```

Anteriormente usamos una función flecha, pero se podría haber hecho igualmente con funciones comunes:

```
// Correcto
this.setState(function(state, props) {
    return {
      counter: state.counter + props.increment
    };
  });
```

Las actualizaciones de estado se fusionan

Cuando invocas a setState(), React combina el objeto que proporcionaste con el estado actual.

Por ejemplo, tu estado puede contener varias variables independientes:

```
constructor(props) {
    super(props);
    this.state = {
      posts: [],      comments: []    };
  }
```

Luego puedes actualizarlas independientemente con invocaciones separadas a setState():

```
componentDidMount() {
    fetchPosts().then(response => {
      this.setState({
        posts: response.posts      });
    });

    fetchComments().then(response => {
      this.setState({
        comments: response.comments      });
    });
  }
```

La fusión es superficial, asi que this.setState({comments}) deja intacto a this.state.posts, pero reemplaza completamente this.state.comments.

Los datos fluyen hacia abajo

Ni los componentes padres o hijos pueden saber si un determinado componente tiene o no tiene estado y no les debería importar si se define como una función o una clase. Por eso es que el estado a menudo se le denomina local o encapsulado. No es accesible desde otro componente excepto de aquel que lo posee y lo asigna.

Un componente puede elegir pasar su estado como props a sus componentes hijos:

```
<FormattedDate date={this.state.date} />
```

El componente FormattedDate recibiría date en sus props y no sabría si vino del estado de Clock, de los props de Clock, o si se escribió manualmente:

```
function FormattedDate(props) {
    return <h2>It is {props.date.toLocaleTimeString()}.</h2>;
  }
```

A esto comúnmente se le llama flujo de datos «descendente» o «unidireccional». Cualquier estado siempre es propiedad de algún componente específico, y cualquier dato o interfaz de usuario derivados de ese estado solo pueden afectar los componentes «debajo» de ellos en el árbol.

Si imaginas un árbol de componentes como una cascada de props, el estado de cada componente es como una fuente de agua adicional que se le une en un punto arbitrario, pero también fluye hacia abajo.

Para mostrar que todos los componentes están verdaderamente aislados, podemos crear un componente App que represente tres componentes <Clock>:

```
function App() {
    return (
      <div>
        <Clock />      <Clock />      <Clock />    </div>
    );
  }

  ReactDOM.render(
    <App />,
    document.getElementById('root')
  );
```

Cada Clock configura su propio temporizador y se actualiza de forma independiente.

En las aplicaciones de React, si un componente tiene o no estado se considera un detalle de implementación del componente que puede cambiar con el tiempo. Puedes usar componentes sin estado dentro de componentes con estado y viceversa.

Manejando eventos

Manejar eventos en elementos de React es muy similar a manejar eventos con elementos del DOM. Hay algunas diferencias de sintaxis:

- 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.

Por ejemplo, el HTML:

```
<button onclick="activateLasers()">
Activate Lasers
</button>
En React es algo diferente:
<button onClick={activateLasers}>  Activate Lasers
</button>
```

Otra diferencia es que en React no puedes retornar false para prevenir el comportamiento por defecto. Debes, explícitamente, llamar preventDefault. Por ejemplo, en un HTML plano, para prevenir el comportamiento por defecto de enviar un formulario, puedes escribir:

```
<form onsubmit="console.log('You clicked submit.'); return false">
  <button type="submit">Submit</button>
</form>
En cambio en React, esto podría ser:
function Form() {
  function handleSubmit(e) {
    e.preventDefault();    console.log('You clicked submit.');
  }

  return (
    <form onSubmit={handleSubmit}>
      <button type="submit">Submit</button>
    </form>
  );
}
```

Aquí, e es un evento sintético. React define estos eventos sintéticos acorde a las especificaciones W3C, para que no tengas que preocuparte por la compatibilidad entre distintos navegadores. Los eventos de React no funcionan exactamente igual que los eventos nativos. Mira la guía de referencia SyntheticEvent para aprender más.
Cuando estás utilizando React, generalmente no necesitas llamar addEventListener para agregar escuchadores de eventos a un elemento del DOM después de que este es creado.

En cambio, solo debes proveer un manejador de eventos cuando el elemento se renderiza inicialmente.

Cuando defines un componente usando una clase de ES6, un patrón muy común es que los manejadores de eventos sean un método de la clase. Por ejemplo, este componente Toggle renderiza un botón que permite al usuario cambiar el estado entre “ENCENDIDO” y “APAGADO”:

```
class Toggle extends React.Component {
    constructor(props) {
      super(props);
      this.state = {isToggleOn: true};

      // Este enlace es necesario para hacer que `this` funcione e
n el callback    this.handleClick = this.handleClick.bind(this);
}

    handleClick() {    this.setState(prevState => ({      isToggle
On: !prevState.isToggleOn    }));  }
    render() {
      return (
        <button onClick={this.handleClick}>        {this.state.isT
oggleOn ? 'ON' : 'OFF'}
        </button>
      );
    }
  }

  ReactDOM.render(
    <Toggle />,
    document.getElementById('root')
  );
```

Debes tener mucho cuidado en cuanto al significado de this en los callbacks de JSX. En JavaScript, los métodos de clase no están ligados por defecto. Si olvidas ligar this.handleClick y lo pasas a onClick, this será undefined cuando se llame la función.

Esto no es un comportamiento especifico de React; esto hace parte de como operan las funciones JavaScript. Generalmente, si refieres un método sin usar () después de este, tal como onClick={this.handleClick}, deberías ligar ese método.

Si te molesta llamar bind, existen dos maneras de evitarlo. Si usas la sintaxis experimental campos públicos de clases, puedes usar los campos de clases para ligar los callbacks correctamente:

```
class LoggingButton extends React.Component {
    // Esta sintaxis nos asegura que `this` está ligado dentro de
handleClick  // Peligro: esto es una sintaxis *experimental*  hand
leClick = () => {    console.log('this is:', this);  }
    render() {
      return (
        <button onClick={this.handleClick}>
          Click me
        </button>
      );
    }
  }
```

Esta sintaxis está habilitada por defecto en Create React App.
Si no estas usando la sintaxis de campos públicos de clases, puedes usar una función flecha en el callback:

```
class LoggingButton extends React.Component {
    handleClick() {
      console.log('this is:', this);
    }

    render() {
      // Esta sintaxis nos asegura que `this` esta ligado dentro d
e handleClick    return (      <button onClick={() => this.handleC
lick()}>        Click me
        </button>
      );
    }
```

El problema con esta sintaxis es que un callback diferente es creado cada vez que LogginButton es renderizado. En la mayoría de los casos, esto está bien. Sin embargo, si este callback se pasa como una propiedad a componentes más bajos, estos componentes podrían renderizarse nuevamente. Generalmente, recomendamos ligar en el constructor o usar la sintaxis de campos de clases, para evitar esta clase de problemas de rendimiento.
Pasando argumentos a escuchadores de eventos.

Dentro de un bucle es muy común querer pasar un parámetro extra a un manejador de eventos. Por ejemplo, si id es el ID de una fila, cualquiera de los códigos a continuación podría funcionar:

```
<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>
<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button
>
```

Las dos líneas anteriores son equivalentes, y utilizan funciones flecha y Function.prototype.bind respectivamente.

En ambos casos, el argumento e que representa el evento de React va a ser pasado como un segundo argumento después del ID. Con una función flecha, tenemos que pasarlo explícitamente, pero con bind cualquier argumento adicional es pasado automáticamente

Renderizado 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 en React funciona de la misma forma que lo hacen las condiciones en JavaScript. Usa operadores de JavaScript como if o el operador condicional para crear elementos representando el estado actual, y deja que React actualice la interfaz de usuario para emparejarlos.

Considera estos dos componentes:

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

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

Vamos a crear un componente Greeting que muestra cualquiera de estos componentes dependiendo si el usuario ha iniciado sesión:

```
function Greeting(props) {
    const isLoggedIn = props.isLoggedIn;
    if (isLoggedIn) {    return <UserGreeting />;  }  return <Gues
tGreeting />;}
  ReactDOM.render(
    // Intentar cambiando isLoggedIn={true}:
    <Greeting isLoggedIn={false} />,  document.getElementById('roo
t'));
```

Este ejemplo renderiza un saludo diferente según el valor del prop isLoggedIn.

Variables de elementos

Puedes usar variables para almacenar elementos. Esto puede ayudarte para renderizar condicionalmente una parte del componente mientras el resto del resultado no cambia. Considera estos dos componentes nuevos que representan botones de cierre e inicio de sesión:

```
function LoginButton(props) {
    return (
      <button onClick={props.onClick}>
        Login
      </button>
    );
  }

  function LogoutButton(props) {
    return (
      <button onClick={props.onClick}>
        Logout
      </button>
    );
  }
```

En el siguiente ejemplo, crearemos un componente con estado llamado LoginControl.
El componente va a renderizar <LoginButton /> o <LogoutButton /> dependiendo de su estado actual. También va a renderizar un <Greeting /> del ejemplo anterior:

```
class LoginControl extends React.Component {
    constructor(props) {
      super(props);
      this.handleLoginClick = this.handleLoginClick.bind(this);
      this.handleLogoutClick = this.handleLogoutClick.bind(this);
      this.state = {isLoggedIn: false};
    }
    handleLoginClick() {
      this.setState({isLoggedIn: true});
    }
    handleLogoutClick() {
      this.setState({isLoggedIn: false});
    }
    render() {
      const isLoggedIn = this.state.isLoggedIn;
      let button;
      if (isLoggedIn) {
        button = <LogoutButton onClick={this.handleLogoutClick} />
;
      } else {
        button = <LoginButton onClick={this.handleLoginClick} />;
   }     return (       <div>         <Greeting isLoggedIn={isLoggedI
n} />         {button}       </div>
      );
    }
  }ReactDOM.render(
    <LoginControl />,
    document.getElementById('root')
  );
```

Si bien declarar una variable y usar una sentencia if es una buena forma de renderizar condicionalmente un componente, a veces podrías querer usar una sintaxis más corta. Hay algunas formas de hacer condiciones en una línea en JSX, explicadas a continuación.

If en una línea con operador lógico &&.

Puedes incluir expresiones en JSX envolviéndolas en llaves. Esto incluye el operador lógico && de JavaScript. Puede ser útil para incluir condicionalmente un elemento:

```
function Mailbox(props) {
    const unreadMessages = props.unreadMessages;
    return (
      <div>
        <h1>Hello!</h1>
        {unreadMessages.length > 0 &&        <h2>           You hav
e {unreadMessages.length} unread messages.        </h2>      }
</div>
    );
  }

  const messages = ['React', 'Re: React', 'Re:Re: React'];
  ReactDOM.render(
    <Mailbox unreadMessages={messages} />,
    document.getElementById('root')
  );
```

Esto funciona porque en JavaScript, true && expresión siempre evalúa a expresión, y false && expresión siempre evalúa a false.

Por eso, si la condición es true, el elemento justo después de && aparecerá en el resultado. Si es false, React lo ignorará.

Ten en cuenta que retornar expresiones falsas hará que el elemento después de '&&' sea omitido pero retornará el valor falso. En el ejemplo de abajo, '
0
' será retornado por el método de renderizado.

```
render() {
    const count = 0;  return (
      <div>
        { count && <h1>Messages: {count}</h1>}    </div>
    );
  }
```

If-Else en una línea con operador condicional

Otro método para el renderizado condicional de elementos en una línea es usar el operador condicional condición? true : false de JavaScript.
En el siguiente ejemplo, lo usaremos para renderizar de forma condicional un pequeño bloque de texto.

```
render() {
    const isLoggedIn = this.state.isLoggedIn;
    return (
      <div>
        The user is <b>{isLoggedIn ? 'currently' : 'not'}</b> logg
ed in.    </div>
    );
  }
```

También puede usarse para expresiones más grandes, aunque es menos obvio lo que está pasando:

```
render() {
    const isLoggedIn = this.state.isLoggedIn;
    return (
      <div>
        {isLoggedIn        ? <LogoutButton onClick={this.handleLog
outClick} />
          : <LoginButton onClick={this.handleLoginClick} />      }
      </div>  );
  }
```

Al igual que en JavaScript, depende de ti elegir un estilo apropiado según lo que tú y tu equipo consideren más legible. Recuerda también que cuando las condiciones se vuelven demasiado complejas, puede ser un buen momento para extraer un componente.

Evitar que el componente se renderice

En casos excepcionales, es posible que desees que un componente se oculte a sí mismo aunque haya sido renderizado por otro componente. Para hacer esto, devuelve null en lugar del resultado de renderizado.

En el siguiente ejemplo, el <WarningBanner /> se renderiza dependiendo del valor del prop llamado warn. Si el valor del prop es false, entonces el componente no se renderiza:

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

  class Page extends React.Component {
    constructor(props) {
      super(props);
      this.state = {showWarning: true};
      this.handleToggleClick = this.handleToggleClick.bind(this);
    }
    handleToggleClick() {
      this.setState(state => ({
        showWarning: !state.showWarning
      }));
    }
    render() {
      return (
        <div>
          <WarningBanner warn={this.state.showWarning} />
          <button onClick={this.handleToggleClick}>
            {this.state.showWarning ? 'Hide' : 'Show'}
          </button>      </div>
      );
    }
  }

  ReactDOM.render(
    <Page />,
    document.getElementById('root')
  );
```

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 y keys

Primero, vamos a revisar como transformas listas en Javascript.
Dado el código de abajo, usamos la función map() para tomar un array de numbers y duplicar sus valores. Asignamos el nuevo array devuelto por map() a la variable doubled y la mostramos:

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((number) => number * 2);console.log(do
ubled);
```

Este código muestra [2, 4, 6, 8, 10] a la consola.

En React, transformar arrays en listas de elementos es casi idéntico.
Renderizado de Múltiples Componentes

Puedes hacer colecciones de elementos e incluirlos en JSX usando llaves {}.
Debajo, recorreremos el array numbers usando la función map() de Javascript.
Devolvemos un elemento <li> por cada ítem . Finalmente, asignamos el array de elementos resultante a listItems:

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

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

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

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

Componente básico de lista

Usualmente renderizarías listas dentro de un componente.
Podemos refactorizar 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('ro
ot')
  );
```

Cuando ejecutes este código, serás advertido que una key debería ser proporcionada para ítems de lista. Una “key” es un atributo especial string que debes incluir al crear listas de elementos. Vamos a discutir por qué esto es importante en la próxima sección.

Vamos a asignar una key a nuestra lista de ítems dentro de numbers.map() y arreglar el problema de la falta de key.

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

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

Keys

Las keys 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 para darle a los elementos una identidad estable:

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

La mejor forma de elegir una key es usando un string que identifique únicamente a un elemento de la lista entre sus hermanos. Habitualmente vas a usar IDs de tus datos como key:

```
const todoItems = todos.map((todo) =>
  <li key={todo.id}>    {todo.text}
  </li>
);
```

Cuando no tengas IDs estables para renderizar, puedes usar el índice del ítem como una key como último recurso:

```
const todoItems = todos.map((todo, index) =>
  // Only do this if items have no stable IDs  <li key={index}>
 {todo.text}
  </li>
);
```

No recomendamos usar índices para keys si el orden de los ítems puede cambiar. Esto puede impactar negativamente el rendimiento y puede causar problemas con el estado del componente. Revisa el artículo de Robin Pokorny para una explicación en profundidad de los impactos negativos de usar un índice como key. Si eliges no asignar una key explícita a la lista de ítems, React por defecto usará índices como keys.
Aquí hay una explicación en profundidad sobre por qué las keys son necesarias si estás interesado en aprender más.

Extracción de componentes con keys

Las keys solo tienen sentido en el contexto del array que las envuelve.
Por ejemplo, si extraes un componente ListItem, deberías mantener la key en los elementos <ListItem /> del array en lugar de en el elemento <li> del propio ListItem.

Ejemplo: Uso Incorrecto de Key

```
function ListItem(props) {
    const value = props.value;
    return (
      // Mal! No hay necesidad de especificar la key aquí:    <li
key={value.toString()}>      {value}
      </li>
    );
  }

  function NumberList(props) {
    const numbers = props.numbers;
    const listItems = numbers.map((number) =>
      // Mal! La key debería haber sido especificada aquí:    <Lis
tItem value={number} />  );
    return (
      <ul>
        {listItems}
      </ul>
    );
  }

  const numbers = [1, 2, 3, 4, 5];
  ReactDOM.render(
    <NumberList numbers={numbers} />,
    document.getElementById('root')
  );
  Ejemplo: Uso Correcto de Key
  function ListItem(props) {
    // Correcto! No hay necesidad de especificar la key aquí:  ret
urn <li>{props.value}</li>;}

  function NumberList(props) {
    const numbers = props.numbers;
    const listItems = numbers.map((number) =>
      // Correcto! La key debería ser especificada dentro del arra
y.    <ListItem key={number.toString()} value={number} />  );
    return (
```

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

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

Una buena regla es que los elementos dentro de map() necesitan keys.
Las keys deben ser únicas solo entre hermanos.

Las keys usadas dentro de arrays deberían ser únicas entre sus hermanos. Sin embargo, no necesitan ser únicas globalmente. Podemos usar las mismas keys cuando creamos dos arrays diferentes:

```
function Blog(props) {
    const sidebar = (    <ul>
        {props.posts.map((post) =>
          <li key={post.id}>          {post.title}
          </li>
        )}
      </ul>
    );
    const content = props.posts.map((post) =>    <div key={post.id
}>      <h3>{post.title}</h3>
        <p>{post.content}</p>
      </div>
    );
    return (
      <div>
        {sidebar}      <hr />
        {content}    </div>
    );
  }

  const posts = [
    {id: 1, title: 'Hello World', content: 'Welcome to learning Re
act!'},
    {id: 2, title: 'Installation', content: 'You can install React
 from npm.'}
```

```
  ];
  ReactDOM.render(
    <Blog posts={posts} />,
    document.getElementById('root')
  );
```

Las keys sirven como una sugerencia para React pero no son pasadas a tus componentes. Si necesitas usar el mismo valor en tu componente, pásasela explícitamente como una propiedad con un nombre diferente:

```
const content = posts.map((post) =>
  <Post
    key={post.id}    id={post.id}    title={post.title} />
);
```

Con el ejemplo de arriba, el componente Post puede leer props.id, pero no props.key. Integrar map() en JSX.

En los ejemplos de arriba declaramos una variable separada listItems y la incluimos en JSX:

```
function NumberList(props) {
    const numbers = props.numbers;
    const listItems = numbers.map((number) =>    <ListItem key={nu
mber.toString()}              value={number} />  );  return (
      <ul>
        {listItems}
      </ul>
    );
  }
```

JSX permite integrar cualquier expresión en llaves así que podemos alinear el resultado de map():

```
function NumberList(props) {
    const numbers = props.numbers;
    return (
      <ul>
        {numbers.map((number) =>        <ListItem key={number.toSt
ring()}                  value={number} />      )}    </ul>
    );
  }
```

Algunas veces esto resulta más claro en código, pero este estilo también puede ser abusado. Como en JavaScript, depende de ti decidir cuando vale la pena extraer una

variable por legibilidad. Ten en mente que si el cuerpo de map() está muy anidado, puede ser un buen momento para extraer un componente.

Formularios

Los elementos de formularios en HTML funcionan un poco diferente a otros elementos del DOM en React, debido a que los elementos de formularios conservan naturalmente algún estado interno. Por ejemplo, este formulario solamente en HTML, acepta un solo nombre.

```
<form>
  <label>
    Name:
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Submit" />
</form>
```

Este formulario tiene el comportamiento predeterminado en HTML que consiste en navegar a una nueva página cuando el usuario envía el formulario. Si deseas este comportamiento en React, simplemente ya funciona así. Pero en la mayoría de casos, es conveniente tener una función en Javascript que se encargue del envío del formulario, y que tenga acceso a los datos que el usuario introdujo en el formulario. La forma predeterminada para conseguir esto es una técnica llamada “componentes controlados”.

Componentes controlados

En HTML, los elementos de formularios como los <input>, <textarea> y el <select> normalmente 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 estado de los componentes, y solo se actualiza con setState().

Podemos combinar ambos haciendo que el estado de React sea la “única fuente de la verdad”. De esta manera, los componentes React que rendericen un formulario también controlan lo que pasa en ese formulario con las subsecuentes entradas del usuario. Un campo de un formulario cuyos valores son controlados por React de esta forma es denominado “componente controlado”.

Por ejemplo, si queremos hacer que el ejemplo anterior muestre el nombre que esta siendo suministrado, podemos escribir el formulario como un componente controlado:

```
class NameForm extends React.Component {
     constructor(props) {
      super(props);
      this.state = {value: ''};
      this.handleChange = this.handleChange.bind(this);
      this.handleSubmit = this.handleSubmit.bind(this);
    }

    handleChange(event) {    this.setState({value: event.target.va
lue});  }
    handleSubmit(event) {
      alert('A name was submitted: ' + this.state.value);
      event.preventDefault();
    }

    render() {
      return (
        <form onSubmit={this.handleSubmit}>        <label>
            Name:
            <input type="text" value={this.state.value} onChange={
this.handleChange} />        </label>
          <input type="submit" value="Submit" />
        </form>
      );
    }
  }
```

Ya que el atributo value es agregado en nuestro elemento del formulario, el valor mostrado siempre será el de this.state.value, haciendo que el estado de React sea la fuente de la verdad. Ya que handleChange corre cada vez que una tecla es oprimida para actualizar el estado de React, el valor mostrado será actualizado mientras que el usuario escribe.

Con un componente controlado, el valor del input siempre está dirigido por el estado de React. Si bien esto significa que tendrás que escribir un poco más de código, ahora podrás pasar también el valor a otros elementos de la interfaz de usuario, o reiniciarlo desde otros manejadores de eventos.

La etiqueta textarea

En HTML, el elemento <textarea> define su texto por sus hijos:

```
<textarea>
  Hello there, this is some text in a text area
</textarea>
```

En React, un <textarea> utiliza un atributo value en su lugar. De esta manera, un formulario que hace uso de un <textarea> puede ser escrito de manera similar a un formulario que utiliza un campo en una sola línea:

```
class EssayForm extends React.Component {
    constructor(props) {
      super(props);
      this.state = {      value: 'Please write an essay about your
 favorite DOM element.'    };
      this.handleChange = this.handleChange.bind(this);
      this.handleSubmit = this.handleSubmit.bind(this);
    }

    handleChange(event) {    this.setState({value: event.target.va
lue});  }
    handleSubmit(event) {
      alert('An essay was submitted: ' + this.state.value);
      event.preventDefault();
    }

    render() {
      return (
        <form onSubmit={this.handleSubmit}>
          <label>
            Essay:
            <textarea value={this.state.value} onChange={this.hand
leChange} />        </label>
          <input type="submit" value="Submit" />
        </form>
      );
    }
  }
```

Recuerda que this.state.value es inicializado en el constructor, de manera que el área de texto empiece con algo de texto.

La etiqueta select

En HTML, <select> crea una lista desplegable. Por ejemplo, este HTML crea una lista desplegable de sabores:

```
<select>
  <option value="grapefruit">Grapefruit</option>
  <option value="lime">Lime</option>
  <option selected value="coconut">Coconut</option>
  <option value="mango">Mango</option>
</select>
```

Ten en cuenta que la opción Coco es inicialmente seleccionada, debido al atributo selected. React, en lugar de utilizar el atributo selected, utiliza un atributo value en la raíz de la etiqueta select. Esto es más conveniente en un componente controlado debido a que solo necesitas actualizarlo en un solo lugar, por ejemplo:

```
class FlavorForm extends React.Component {
    constructor(props) {
      super(props);
      this.state = {value: 'coconut'};
      this.handleChange = this.handleChange.bind(this);
      this.handleSubmit = this.handleSubmit.bind(this);
    }

    handleChange(event) {    this.setState({value: event.target.va
lue});  }
    handleSubmit(event) {
      alert('Your favorite flavor is: ' + this.state.value);
      event.preventDefault();
    }

    render() {
      return (
        <form onSubmit={this.handleSubmit}>
          <label>
            Pick your favorite flavor:
            <select value={this.state.value} onChange={this.handle
Change}>              <option value="grapefruit">Grapefruit</option>
              <option value="lime">Lime</option>
              <option value="coconut">Coconut</option>
              <option value="mango">Mango</option>
            </select>
          </label>
          <input type="submit" value="Submit" />
```

```
        </form>
      );
    }
  }
```

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.

Nota

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

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

La etiqueta file input

En HTML, un <input type="file"> permite que el usuario escoja uno o varios archivos de su dispositivo de almacenamiento para ser cargados a un servidor o ser manipulados por Javascript mediante el API de Archivos.

```
<input type="file" /></select>
```

Ya que su valor es solo de lectura, es un componente no controlado en React. Es explicado en detalle junto a otros componentes no controlados más adelante en la documentación.

Manejando múltiples inputs

Cuando necesitas manejar múltiples elementos input controlados, puedes agregar un atributo name a cada uno de los elementos y dejar que la función controladora decida que hacer basada en el valor de event.target.name.

Por ejemplo:

```
class Reservation extends React.Component {
    constructor(props) {
      super(props);
      this.state = {
        isGoing: true,
        numberOfGuests: 2
      };

      this.handleInputChange = this.handleInputChange.bind(this);
    }

    handleInputChange(event) {
      const target = event.target;
      const value = target.type === 'checkbox' ? target.checked :
target.value;
      const name = target.name;
      this.setState({
        [name]: value    });
    }

    render() {
      return (
        <form>
          <label>
            Is going:
            <input
              name="isGoing"            type="checkbox"
              checked={this.state.isGoing}
              onChange={this.handleInputChange} />
          </label>
          <br />
          <label>
            Number of guests:
            <input
              name="numberOfGuests"            type="number"
              value={this.state.numberOfGuests}
              onChange={this.handleInputChange} />
          </label>
        </form>
      );
    }
  }
```

Ten en cuenta como utilizamos la sintaxis de la propiedad name computada de ES6 para actualizar la clave del estado correspondiente al nombre del input.

```
this.setState({
    [name]: value});
```

Esto es equivalente a este código ES5:

```
var partialState = {};
partialState[name] = value;this.setState(partialState);
```

También, ya que setState() automáticamente combina un estado parcial al estado actual, solamente necesitamos llamarlo con las partes que han cambiado.

Valor nulo en un input controlado

Especificar la propiedad value en un componente controlado evita que el usuario cambie la entrada a menos que así lo quiera. Si has especificado un value pero la entrada aún es editable, quizás agregaste accidentalmente al value un valor undefined o null.
El código a continuación demuestra esto. (El input está bloqueado en principio, pero se vuelve editable después de un corto retraso).

```
ReactDOM.render(<input value="hi" />, mountNode);

setTimeout(function() {
  ReactDOM.render(<input value={null} />, mountNode);
}, 1000);
```

Alternativas a componentes controlados

A veces puede ser tedioso usar componentes controlados, debido a que se necesita escribir un controlador de eventos para cada forma en la que tus datos puedan cambiar y agregarlos a todos en el estado del input a través del componente React. Esto puede volverse particularmente molesto cuando estás convirtiendo una base de código existente a React, o integrando una aplicación React con una biblioteca que no integra React. En estas situaciones, puede que quieras leer acerca de componentes no controlados, una técnica alternativa para implementar inputs en formularios.

Soluciones completas

Si lo que estás buscando es una solución completa incluyendo validación, tener en cuenta los campos visitados y manejar el envío del formulario, Formik es una de las opciones populares. Sin embargo, está construido con los mismos principios de los componentes controlados y manejo de estado, así que no los dejes de aprender.

Levantando el estado

Usualmente, muchos componentes necesitan reflejar el mismo cambio en los datos. Recomendamos elevar el estado compartido al ancestro común más cercano. Veamos cómo funciona.

En esta sección, crearemos una calculadora de temperatura que calculará si el agua hervirá a una determinada temperatura.

Comenzaremos con un componente llamado BoilingVerdict. Este acepta la temperatura en celsius como una propiedad e imprime si es suficiente para que el agua hierva:

```
function BoilingVerdict(props) {
    if (props.celsius >= 100) {
      return <p>The water would boil.</p>;  }
    return <p>The water would not boil.</p>;}
```

Luego, crearemos un componente llamado Calculator. Este renderiza un <input> que permite insertar la temperatura y guarda su valor en this.state.temperature.
Además, renderiza el BoilingVerdict para el valor insertado.

```
class Calculator extends React.Component {
    constructor(props) {
      super(props);
      this.handleChange = this.handleChange.bind(this);
      this.state = {temperature: ''};  }

    handleChange(e) {
      this.setState({temperature: e.target.value});  }

    render() {
      const temperature = this.state.temperature;    return (
        <fieldset>
          <legend>Enter temperature in Celsius:</legend>
          <input          value={temperature}          onChange={t
his.handleChange} />        <BoilingVerdict          celsius={pars
eFloat(temperature)} />      </fieldset>
      );
    }
  }
```

Añadiendo una segunda entrada

Nuestro nuevo requisito es que, además de la temperatura en Celsius, proveemos la temperatura en Fahrenheit, y estas se mantienen sincronizadas.
Podemos comenzar por extraer el componente TemperatureInput de Calculator.
Añadiremos una nueva propiedad scale al mismo que podrá ser "c" o "f":

```
const scaleNames = {  c: 'Celsius',  f: 'Fahrenheit'};
class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    const scale = this.props.scale;    return (
      <fieldset>
        <legend>Enter temperature in {scaleNames[scale]}:</legend>
        <input value={temperature}
               onChange={this.handleChange} />
      </fieldset>
    );
  }
}
```

Ahora podemos cambiar Calculator para que renderice dos entradas separadas para la temperatura:

```
class Calculator extends React.Component {
    render() {
      return (
        <div>
          <TemperatureInput scale="c" />        <TemperatureInput
scale="f" />      </div>
      );
    }
  }
```

Ahora tenemos dos entradas, pero cuando insertamos la temperatura en una de ellas, la otra no se actualiza. Esto contradice nuestro requisito: queremos que se mantengan sincronizadas.

Tampoco podemos mostrar el componente BoilingVerdict de Calculator. Calculator no conoce la temperatura actual porque está escondida dentro de TemperatureInput.

Escribiendo funciones de conversión

Primeramente, escribiremos dos funciones para convertir de Celsius a Fahrenheit y viceversa:

```
function toCelsius(fahrenheit) {
    return (fahrenheit - 32) * 5 / 9;
  }

  function toFahrenheit(celsius) {
    return (celsius * 9 / 5) + 32;
  }
```

Estas dos funciones convierten números. Escribiremos otra función que tomará la cadena temperature y una función de conversión como parámetros y retornará una cadena. La usaremos para calcular el valor de una entrada basado en la otra entrada.
Retorna una cadena vacía si temperature es inválida y mantiene la salida redondeada al tercer lugar decimal:

```
function tryConvert(temperature, convert) {
    const input = parseFloat(temperature);
    if (Number.isNaN(input)) {
      return '';
    }
    const output = convert(input);
    const rounded = Math.round(output * 1000) / 1000;
    return rounded.toString();
  }
```

Por ejemplo, tryConvert('abc', toCelsius) retorna una cadena vacía, y tryConvert('10.22', toFahrenheit) retorna '50.396'.

Levantando el estado

Actualmente, ambos componentes TemperatureInput mantienen de manera independiente sus valores en el estado local:

```
class TemperatureInput extends React.Component {
    constructor(props) {
      super(props);
      this.handleChange = this.handleChange.bind(this);
      this.state = {temperature: ''};  }

    handleChange(e) {
      this.setState({temperature: e.target.value});  }

    render() {
      const temperature = this.state.temperature;    // ...
```

Sin embargo, queremos que estas dos entradas estén sincronizadas. Cuando actualicemos la entrada de Celsius, la entrada de Fahrenheit debe reflejar la conversión de temperatura, y viceversa.

En React, la compartición 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”. Eliminaremos el estado local de TemperatureInput y lo moveremos hacia Calculator.

Si Calculator posee el estado compartido, entonces se convierte en la “fuente de verdad” para la temperatura actual en ambas entradas. Este puede instruir a ambos a tener valores consistentes entre sí. Puesto que las propiedades de ambos componentes TemperatureInput vienen del mismo componente Calculator, las dos entradas siempre estarán sincronizadas.

Veamos cómo trabaja esto paso a paso.

Primeramente, reemplazaremos this.state.temperature con this.props.temperature en el componente TemperatureInput. Por ahora, pretendamos que this.props.temperature ya existe, aunque necesitaremos pasarlo de el componente Calculator en el futuro:

```
render() {
    // Before: const temperature = this.state.temperature;
    const temperature = this.props.temperature;    // ...
```

Sabemos que las propiedades son de solo lectura. Cuando temperature estaba en el estado local, TemperatureInput solo llama a this.setState() para modificarlo. Sin embargo,

ahora que temperature viene del padre como una propiedad, TemperatureInput no tiene ningún control sobre la misma.

En React, esto se resuelve usualmente haciendo un componente “controlado”. Así como el <input> del DOM acepta una propiedad value y otra onChange, también TemperatureInput puede aceptar las propiedades temperature y onTemperatureChange de su padre Calculator.

Ahora, cuando TemperatureInput quiera actualizar su temperatura, este llama a this.props.onTemperatureChange:

```
handleChange(e) {
    // Before: this.setState({temperature: e.target.value});
    this.props.onTemperatureChange(e.target.value);    // ...
```

Nota:

No existe un significado especial para los nombres de las propiedades temperature o onTemperatureChange en los componentes. Pudimos haberles dado otro nombre, como value y onChange lo cual es una convención común.

La propiedad onTemperatureChange será proporcionada de manera conjunta con temperature por el componente padre Calculator. Este manejará el cambio modificando su estado local, volviendo a renderizar ambas entradas con los nuevos valores.

Analizaremos los cambios en la implementación de Calculator en un momento.

Antes de ahondar en los cambios de Calculator, recapitulemos nuestros cambios al componente TemperatureInput. Hemos eliminado el estado local de este, y en vez de leer this.state.temperature, ahora leemos this.props.temperature. En vez de llamar a this.setState() cuando queremos hacer un cambio, ahora llamamos a this.props.onTemperatureChange(), que será proporcionado por Calculator:

```
class TemperatureInput extends React.Component {
    constructor(props) {
      super(props);
      this.handleChange = this.handleChange.bind(this);
    }

    handleChange(e) {
      this.props.onTemperatureChange(e.target.value);  }

    render() {
```

```
      const temperature = this.props.temperature;    const scale =
 this.props.scale;
      return (
        <fieldset>
          <legend>Enter temperature in {scaleNames[scale]}:</legen
d>
          <input value={temperature}
                 onChange={this.handleChange} />
        </fieldset>
      );
    }
  }
```

Ahora miremos el componente Calculator.

Guardaremos temperature y scale en su estado local. Este es el estado que hemos "levantado" de las entradas, y servirá como la "fuente de verdad" para ambos. Es la representación mínima de todos los datos que debemos conocer para renderizar ambas entradas.

Por ejemplo, si insertamos 37 en la entrada de Celsius, el estado del componente Calculator será:

```
{
    temperature: '37',
    scale: 'c'
  }
  Si luego editamos el valor de Fahrenheit para que sea 212, el es
tado de Calculator será:
  {
    temperature: '212',
    scale: 'f'
  }
```

Pudimos haber guardado el valor de ambas entradas pero resulta que no es necesario. Es suficiente con guardar el valor de la entrada recientemente cambiada, y la escala que esta representa. Entonces podemos inferir el valor de la otra entrada basados solamente en el valor actual de temperature y scale.

Las entradas se mantienen sincronizadas porque los valores son calculados a partir del mismo estado:

```
class Calculator extends React.Component {
    constructor(props) {
      super(props);
      this.handleCelsiusChange = this.handleCelsiusChange.bind(thi
s);
      this.handleFahrenheitChange = this.handleFahrenheitChange.bi
nd(this);
      this.state = {temperature: '', scale: 'c'};  }

    handleCelsiusChange(temperature) {
      this.setState({scale: 'c', temperature});  }

    handleFahrenheitChange(temperature) {
      this.setState({scale: 'f', temperature});  }

    render() {
      const scale = this.state.scale;    const temperature = this.
state.temperature;    const celsius = scale === 'f' ? tryConvert(t
emperature, toCelsius) : temperature;    const fahrenheit = scale
=== 'c' ? tryConvert(temperature, toFahrenheit) : temperature;
      return (
        <div>
          <TemperatureInput
            scale="c"
            temperature={celsius}          onTemperatureChange={th
is.handleCelsiusChange} />        <TemperatureInput
            scale="f"
            temperature={fahrenheit}          onTemperatureChange=
{this.handleFahrenheitChange} />        <BoilingVerdict
            celsius={parseFloat(celsius)} />      </div>
      );
    }
  }
```

Ahora, no importa qué entrada edites, this.state.temperature y this.state.scale en el componente Calculator se actualizan. Una de las entradas toma el valor tal cual, entonces cualquier dato del usuario se conserva, y el valor de la otra entrada es recalculado basado en este cambio.

Recapitulemos qué pasa cuando editamos una entrada:

- React llama a la función especificada como onChange en el <input> del DOM. En nuestro caso es el método handleChange en el componente TemperatureInput.

- El método handleChange en el componente TemperatureInput llama a this.props.onTemperatureChange() con el nuevo valor. Sus propiedades, incluyendo onTemperatureChange, fueron provistas para el componente padre Calculator.
- Cuando renderizó previamente, Calculator especificó que onTemperatureChange del componente TemperatureInput con la escala Celsius es el método handleCelsiusChange y onTemperatureChange del componente TemperatureInput con escala Fahrenheit es el método handleFahrenheitChange. Entonces, cada uno de estos métodos es llamado dependiendo del componente que se edite.
- Dentro de estos métodos, el componente Calculator pregunta a React para volver a renderizar a sí mismo llamando al método this.setState() con el nuevo valor y la escala actual de la entrada que acabamos de editar.
- React llama al método render del componente Calculator para saber cómo debe lucir la interfaz de usuario. Los valores de ambas entradas son recalculados en base a la temperatura actual y la escala activa. La conversión de temperatura es hecha aquí.
- React llama a los métodos render de los componentes TemperatureInput de manera individual con sus nuevas propiedades especificadas por Calculator. Aprende como sus interfaces de usuario deberían verse.
- React llama al método render del componente BoilingVerdict, pasando la temperatura en Celsius como una propiedad.
- React DOM actualiza el DOM con el componente BoilingVerdict y sincroniza los valores deseados para las entradas. La entrada que acabamos de actualizar recibe su valor actual, y la otra entrada es actualizada a su temperatura luego de hacer la conversión. Toda actualización sigue los mismos pasos y las entradas se mantienen sincronizadas.

Composición vs. Herencia

React tiene un potente modelo de composición, y recomendamos usar composición en lugar de herencia para reutilizar código entre componentes.

En esta sección consideraremos algunos problemas en los que los desarrolladores nuevos en React a menudo emplean herencia, y mostraremos cómo los podemos resolver con composición.

Contención

Algunos componentes no conocen sus hijos de antemano. Esto es especialmente común para componentes como Sidebar o Dialog que representan “cajas” genéricas. Recomendamos 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 a
nidando el JSX:
  function WelcomeDialog() {
    return (
      <FancyBorder color="blue">
        <h1 className="Dialog-
title">        Welcome      </h1>      <p className="Dialog-
message">        Thank you for visiting our spacecraft!      </p>
   </FancyBorder>
    );
  }
```

Cualquier cosa dentro de la etiqueta JSX <FancyBorder> se pasa dentro del componente FancyBorder como la prop children. Como FancyBorder renderiza {props.children} dentro de un , los elementos que se le han pasado aparecen en el resultado final.

Aunque es menos común, a veces puedes necesitar múltiples “agujeros” en un componente. En estos casos puedes inventarte tu propia convención en lugar de usar children:

```
function SplitPane(props) {
    return (
      <div className="SplitPane">
        <div className="SplitPane-left">
          {props.left}      </div>
        <div className="SplitPane-right">
          {props.right}      </div>
      </div>
    );
  }

  function App() {
```

```
    return (
      <SplitPane
        left={
          <Contacts />      }
        right={
          <Chat />      } />
    );
  }
```

Los elementos como <Contacts /> y <Chat /> son simplemente objetos, por lo que puedes pasarlos como props como cualquier otro dato. Este enfoque puede recordarte a “huecos” (slots) en otras bibliotecas, pero no hay limitaciones en lo que puedes pasar como props en React.

Especialización

A veces pensamos en componentes como “casos concretos” de otros componentes. Por ejemplo, podríamos 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) {
    return (
      <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 vis
iting our spacecraft!" />  );
  }
  Pruébalo en CodePen
  La composición funciona igual de bien para componentes definidos
 como clases:
  function Dialog(props) {
    return (
      <FancyBorder color="blue">
        <h1 className="Dialog-title">
          {props.title}
```

```
        </h1>
        <p className="Dialog-message">
          {props.message}
        </p>
        {props.children}    </FancyBorder>
    );
  }

  class SignUpDialog extends React.Component {
    constructor(props) {
      super(props);
      this.handleChange = this.handleChange.bind(this);
      this.handleSignUp = this.handleSignUp.bind(this);
      this.state = {login: ''};
    }

    render() {
      return (
        <Dialog title="Mars Exploration Program"
                message="How should we refer to you?">
          <input value={this.state.login}                onChange={
this.handleChange} />        <button onClick={this.handleSignUp}>
          ¡Apúntame!        </button>      </Dialog>
      );
    }

    handleChange(e) {
      this.setState({login: e.target.value});
    }

    handleSignUp() {
      alert(`Bienvenido abordo, ${this.state.login}!`);
    }
  }
```

¿Entonces qué pasa con la herencia?

En Facebook usamos React en miles de componentes, y no hemos hallado ningún caso de uso en el que recomendaríamos crear jerarquías de herencia de componentes.
Las props y la composición te dan toda la flexibilidad que necesitas para personalizar el aspecto y el comportamiento de un componente de forma explícita y segura. Recuerda que los componentes pueden aceptar props arbitrarias, incluyendo valores primitivos, elementos de React y funciones.

Si quieres reutilizar funcionalidad que no es de interfaz entre componentes, sugerimos que la extraigas en un módulo de JavaScript independiente. Los componentes pueden importarlo y usar esa función, objeto, o clase, sin extenderla.

Información tomada de: https://es.reactjs.org/docs/introducing-jsx.html