Todo lo que tenes que saber!

Principiante

¿Qué es React?

React es una biblioteca de JavaScript de código abierto para construir interfaces de usuario. Está basada en la componetización de la UI: la interfaz se divide en componentes independientes, que contienen su propio estado. Cuando el estado de un componente cambia, React vuelve a renderizar la interfaz.

Esto hace que React sea una herramienta muy útil para construir interfaces complejas, ya que permite dividir la interfaz en piezas más pequeñas y reutilizables.

Fue creada en 2011 por Jordan Walke, un ingeniero de software que trabajaba en Facebook y que quería simplificar la forma de crear interfaces de usuario complejas.

Es una biblioteca muy popular y es usada por muchas empresas como Facebook, Netflix, Airbnb, Twitter, Instagram, etc.

Enlaces de interés:

- Documentación oficial de React en Español
- Documentación oficial de React actualizada en inglés

¿Cuáles son las características principales de React?

Las características principales de React son:

- **Componentes**: React está basado en la componetización de la UI. La interfaz se divide en componentes independientes, que contienen su propio estado. Cuando el estado de un componente cambia, React vuelve a renderizar la interfaz.
- **Virtual DOM**: React usa un DOM virtual para renderizar los componentes. El DOM virtual es una representación en memoria del DOM real. Cuando el estado de un componente cambia, React vuelve a renderizar la interfaz. En lugar de modificar el DOM real, React modifica el DOM virtual y, a

continuación, compara el DOM virtual con el DOM real. De esta forma, React sabe qué cambios se deben aplicar al DOM real.

- **Declarativo**: React es declarativo, lo que significa que no se especifica cómo se debe realizar una tarea, sino qué se debe realizar. Esto hace que el código sea más fácil de entender y de mantener.
- **Unidireccional**: React es unidireccional, lo que significa que los datos fluyen en una sola dirección. Los datos fluyen de los componentes padres a los componentes hijos.
- **Universal**: React se puede ejecutar tanto en el cliente como en el servidor. Además, puedes usar React Native para crear aplicaciones nativas para Android e iOS.

¿Qué significa exactamente que sea declarativo?

No le decimos cómo debe renderizar la interfaz a base de instrucciones. Le decimos qué debe renderizar y React se encarga de renderizarlo.

Un ejemplo entre declarativo e imperativo:

```
// Declarativoconst element = <h1>Hello, world</h1>// Imperativoconst ele
ment = document.createElement('h1')
element.innerHTML = 'Hello, world'
```

¿Qué es un componente?

Un componente es una pieza de código que renderiza una parte de la interfaz. Los componentes pueden ser parametrizados, reutilizados y pueden contener su propio estado.

En React los componentes se crean usando funciones o clases.

¿Qué es JSX?

React usa JSX para declarar qué debe renderizar. JSX es una extensión de JavaScript que permite escribir un código más cercano visualmente a HTML, que mejora la legibilidad del código y hace que sea más fácil de entender.

Sin JSX, deberíamos usar `React.createElement` para crear los elementos de la interfaz manualmente de esta forma:

```
import { createElement } from 'react'function Hello() {
  // un componente es una función! 👀  return React.createElement(
    'h1', // elemento a renderizar    null, // atributos del elemento    'Hola Mun
do 👋🌍!' // contenido del elemento  )
}
```

Esto es muy tedioso y poco legible. Por eso, React usa JSX para declarar qué debe renderizar. Por eso usamos JSX de esta forma:

```
function Hello() {
  return <h1>Hola Mundo 👋🌍!</h1>}
```

Ambos códigos son equivalentes.

¿Cómo se transforma el JSX?

El JSX se transforma en código JavaScript compatible en el navegador usando un *transpilador* o *compilador*. El más famoso es a día de hoy Babel, que utiliza una serie de plugins para ser compatible con la transformación, pero existen otros como SWC.

Puedes ver cómo se transforma el JSX en el playground de código de Babel.

Hay casos especiales en los que un transpilador no es necesario. Por ejemplo, **Deno tiene soporte nativo para la sintaxis JSX** y no es necesario transformar el código para hacerlo compatible.

¿Cuál es la diferencia entre componente y elemento en React?

Un componente es una función o clase que recibe props y devuelve un elemento.
Un elemento es un objeto que representa un nodo del DOM o una instancia de un componente de React.

```
// Elemento que representa un nodo del DOM{
  type: 'button',  props: {
    className: 'button button-blue',    children: {
      type: 'b',      props: {
        children: 'OK!'      }
    }
```

```
  }
}
// Elemento que representa una instancia de un componente{
  type: Button,  props: {
    color: 'blue',    children: 'OK!'  }
}
```

¿Cómo crear un componente en React?

Los componentes en React son funciones o clases que devuelven un elemento de React. Hoy en día lo más recomendado es usar funciones:

```
function HelloWorld() {
  return <h1>Hello World!</h1>}
```

Pero también puedes usar una clase para crear un componente React:

```
import { Component } from 'react'class HelloWorld extends Component {
  render() {
    return <h1>Hello World!</h1>  }
}
```

Lo importante es que el nombre de la función o clase empiece con una letra mayúscula. Esto es necesario para que React pueda distinguir entre componentes y elementos HTML.

¿Qué son las props en React?

Las props son las propiedades de un componente. Son datos que se pasan de un componente a otro. Por ejemplo, si tienes un componente `Button` que muestra un botón, puedes pasarle una prop `text` para que el botón muestre ese texto:

```
function Button(props) {
  return <button>{props.text}</button>}
```

Podríamos entender que el componente `Button` es un botón genérico, y que la prop `text` es el texto que se muestra en el botón. Así estamos creando un componente reutilizable.

Debe considerarse además que al usar cualquier expresión JavaScript dentro de JSX debe envolverlos con `{}`, en este caso el objeto `props`, de otra forma JSX lo considerará como texto plano.

Para usarlo, indicamos el nombre del componente y le pasamos las props que queremos:

```
<Button text="Haz clic aquí" /><Button text="Seguir a @EmiDev" />
```

Las props son una forma de parametrizar nuestros componentes igual que hacemos con las funciones. Podemos pasarle cualquier tipo de dato a un componente, incluso otros componentes.

¿Qué es y para qué sirve la prop `children` en React?

La prop `children` es una prop especial que se pasa a los componentes. Es un objeto que contiene los elementos que envuelve un componente.

Por ejemplo, si tenemos un componente `Card` que muestra una tarjeta con un título y un contenido, podemos usar la prop `children` para mostrar el contenido:

```
function Card(props) {
 return (
  <div className='card'>    <h2>{props.title}</h2>    <div>{props.child
ren}</div>   </div> )
}
```

Y luego podemos usarlo de la siguiente forma:

```
<Card title='Título de la tarjeta'> <p>Contenido de la tarjeta</p></Card>
```

En este caso, la prop `children` contiene el elemento `<p>Contenido de la tarjeta</p>`.

Conocer y saber usar la prop `children` es muy importante para crear componentes reutilizables en React.

¿Qué diferencia hay entre props y state?

Las *props* son un objeto que **se pasan como argumentos de un componente padre a un componente hijo**. Son inmutables y no se pueden modificar desde el componente hijo.

El *state* es un valor que **se define dentro de un componente**. Su valor es inmutable (no se puede modificar directamente) pero se puede establecer un valor nuevo del estado para que React vuelva a renderizar el componente.

Así que mientras tanto *props* como *state* afectan al renderizado del componente, su gestión es diferente.

¿Se puede inicializar un estado con el valor de una prop? ¿Qué pasa si lo haces y qué hay que tener en cuenta?

Sí, se puede inicializar el estado con el valor de una prop. Pero hay que tener en cuenta que, si la prop cambia, el estado no se actualizará automáticamente. Esto es porque el estado se inicializa una vez, cuando el componente se monta por primera vez.

Por ejemplo, con componentes funcionales:

```
const Counter = () => {
  const [count, setCount] = useState(0)
  return (
    <div>     <Count count={count} />     <button onClick={() => setCount(c
ount + 1)}>Aumentar</button>    </div>  )
}
const Count = ({ count }) => {
  const [number, setNumber] = useState(count)
  return <p>{number}</p>}
```

En este caso, el componente `Count` inicializa su estado con el valor de la prop `count`. Pero si cambia el valor de la prop `count`, el estado no se actualizará automáticamente. Por lo que al hacer click, siempre veremos el número 0 en pantalla.

En este ejemplo, lo mejor sería simplemente usar la prop `count` en el componente `Count` y así siempre se volvería a renderizar.

Es una buena práctica evitar al máximo los estados de nuestros componentes y, siempre que se pueda, simplemente calcular el valor a mostrar a partir de las props.

En el caso que necesites inicializar un estado con una prop, es una buena práctica añadir el prefijo de `initial` a la prop para indicar que es el valor inicial del estado y que luego no lo usaremos más:

```
const Input = ({ initialValue }) => {
  const [value, setValue] = useState(initialValue)
  return <input value={value} onChange={e => setValue(e.target.value)} />}
```

Es un error muy común pensar que la prop actualizará el estado, así que tenlo en cuenta.

¿Qué es el renderizado condicional en React?

El renderizado condicional es la forma de mostrar un componente u otro dependiendo de una condición.

Para hacer renderizado condicional en React usamos el operador ternario:

```
function Button({ text }) {
  return text ? <button>{text}</button> : null}
```

En este caso, si la prop `text` existe, se renderiza el botón. Si no existe, no se renderiza nada.

Es común encontrar implementaciones del renderizado condicional con el operador `&&`, del tipo:

```
function List({ listArray }) {
  return listArray?.length && listArray.map(item => item)
}
```

Parece que tiene sentido... si el `length` es positivo (mayor a cero) pintamos el map. !Pues no! ❌ Cuidado, si tiene `length` de cero ya que se pintará en el navegador un 0.

Es preferible utilizar el operador ternario. *Kent C. Dodds* tiene un artículo interesante hablando del tema. Use ternaries rather than && in JSX

¿Cómo puedes aplicar clases CSS a un componente en React y por qué no se puede usar `class`?

Para aplicar clases CSS a un componente en React usamos la prop `className`:

```
function Button({ text }) {
```

```
  return <button className='button'>{text}</button>}
```

La razón por la que se llama `className` es porque `class` es una palabra reservada en JavaScript. Por eso, en JSX, tenemos que usar `className` para aplicar clases CSS.

¿Cómo puedes aplicar estilos en línea a un componente en React?

Para aplicar estilos CSS en línea a un componente en React usamos la prop `style`. La diferencia de cómo lo haríamos con HTML, es que en React los estilos se pasan como un objeto y no como una cadena de texto (esto puede verse más claro con los dobles corchetes, los primeros para indicar que es una expresión JavaScript, y los segundos para crear el objeto):

```
function Button({ text }) {
  return <button style={{ color: 'red', borderRadius: '2px' }}>{text}</button
>}
```

Fíjate que, además, los nombres de las propiedades CSS están en camelCase.

¿Cómo puedo aplicar estilos de forma condicional a un componente en React?

Puedes aplicar estilos de forma condicional a un componente en React usando la prop `style` y un operador ternario:

```
function Button({ text, primary }) {
  return <button style={{ color: primary ? 'red' : 'blue' }}>{text}</button>}
```

En el caso anterior, si la prop `primary` es `true`, el botón tendrá el color rojo. Si no, tendrá el color azul.

También puedes seguir la misma mecánica usando clases. En este caso, usamos el operador ternario para decidir si añadir o no la clase:

```
function Button({ text, primary }) {
  return <button className={primary ? 'button-primary' : ''}>{text}</button
>}
```

También podemos usar bibliotecas como `classnames`:

```
import classnames from 'classnames'function Button({ text, primary }) {
  return <button className={classnames('button', { primary })}>{text}</but
ton>}
```

En este caso, si la prop `primary` es `true`, se añadirá la clase `primary` al botón. Si no, no se añadirá. En cambio la clase `button` siempre se añadirá.

¿Qué es el renderizado de listas en React?

El renderizado de listas es la forma de iterar un array de elementos y renderizar elementos de React para cada uno de ellos.

Para hacer renderizado de listas en React usamos el método `map` de los arrays:

```
function List({ items }) {
  return (
    <ul>     {items.map(item => (
      <li key={item.id}>{item.name}</li>     ))}    </ul>  )
}
```

En este caso, se renderiza una lista de elementos usando el componente `List`. El componente `List` recibe una prop `items` que es un array de objetos del tipo `[{ id: 1, name: "John Doe" }]`. El componente `List` renderiza un elemento `li` por cada elemento del array.

El elemento `li` tiene una prop `key` que es un identificador único para cada elemento. Esto es necesario para que React pueda identificar cada elemento de la lista y actualizarlo de forma eficiente. Más adelante hay una explicación más detallada sobre esto.

¿Cómo puedes escribir comentarios en React?

Si vas a escribir un comentario fuera del renderizado de un componente, puedes usar la sintaxis de comentarios de JavaScript sin problemas:

```
function Button({ text }) {
  // Esto es un comentario  /* Esto es un comentario  de varias líneas */  ret
urn <button>{text}</button>}
```

Si vas a escribir un comentario dentro del renderizado de un componente, debes envolver el comentario en llaves y usar siempre la sintaxis de comentarios de bloque:

```
function Button({ text }) {
  return (
    <button>      {/* Esto es un comentario en el render */}      {text}    </butto
n>  )
}
```

¿Cómo añadir un evento a un componente en React?

Para añadir un evento a un componente en React usamos la sintaxis `on` y el nombre del evento nativo del navegador en *camelCase*:

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

En este caso, el componente `Button` recibe una prop `onClick` que es una función. Cuando el usuario hace clic en el botón, se ejecuta la función `onClick`.

¿Cómo puedo pasar un parámetro a una función que maneja un evento en React?

Para pasar un parámetro a una función que maneja un evento en React podemos usar una función anónima:

```
function Button({ id, text, onClick }) {
  return <button onClick={() => onClick(id)}>{text}</button>}
```

Cuando el usuario hace clic en el botón, se ejecuta la función `onClick` pasándole como parámetro el valor de la prop `id`.

También puedes crear una función que ejecuta la función `onClick` pasándole el valor de la prop `id`:

```
function Button({ id, text, onClick }) {
  const handleClick = event => {
    // handleClick recibe el evento original    onClick(id)
```

```
  }
  return <button onClick={handleClick}>{text}</button>}
```

¿Qué es el estado en React?

El estado es un objeto que contiene datos que pueden cambiar en el tiempo. En React, el estado se usa para controlar los cambios en la interfaz.

Para que entiendas el concepto, piensa en el interruptor de una habitación. Estos interruptores suelen tener dos estados: encendido y apagado. Cuando accionamos el interruptor y lo ponemos en `on` entonces la luz se enciende y cuando lo ponemos en `off` la luz se apaga.

Este mismo concepto se puede aplicar a la interfaz de usuario. Por ejemplo, el botón Me Gusta de Facebook tendría el estado de `meGusta` a `true` cuando el usuario le ha dado a Me Gusta y a `false` cuando no lo ha hecho.

No solo podemos tener en el estado valores booleanos, también podemos tener objetos, arrays, números, etc.

Por ejemplo, si tienes un componente `Counter` que muestra un contador, puedes usar el estado para controlar el valor del contador.

Para crear un estado en React usamos el hook `useState`:

```
import { useState } from 'react'function Counter() {
  const [count, setCount] = useState(0)
  return (
    <div>     <p>Contador: {count}</p>     <button onClick={() => setCount
(count + 1)}>Aumentar</button>    </div>  )
}
```

Al usar el hook `useState` este devolverá un `array` de dos posiciones:

1. El valor del estado.
2. La función para cambiar el estado.

Suele usarse desestructuración para facilitar la lectura y ahorrarnos algunas líneas de código. Por otro lado, al pasarle un dato como parámetro al `useState` le estamos indicando su estado inicial.

Con un componente de clase, la creación del estado sería así:

```
import { Component } from 'react'class Counter extends Component {
  constructor(props) {
    super(props)
    this.state = { count: 0 }
  }
  render() {
    return (
      <div>       <p>Contador: {this.state.count}</p>       <button onClick={()
⇒ this.setState({ count: this.state.count + 1 })}>         Aumentar
        </button>      </div>    )
  }
}
```

¿Qué son los hooks?

Los Hooks son una API de React que nos permite tener estado, y otras características de React, en los componentes creados con una function.

Esto, antes, no era posible y nos obligaba a crear un componente con `class` para poder acceder a todas las posibilidades de la librería.

Hooks es gancho y, precisamente, lo que hacen, es que te permiten enganchar tus componentes funcionales a todas las características que ofrece React.

¿Qué hace el hook `useState`?

El hook `useState` es utilizado para crear variables de estado, quiere decir que su valor es dinámico, que este puede cambiar en el tiempo y eso requiere una re-renderización del componente donde se utiliza

Recibe un parámetro:

- El valor inicial de nuestra variable de estado.

Devuelve un array con dos variables:

- En primer lugar tenemos la variable que contiene el valor
- La siguiente variable es una función set, requiere el nuevo valor del estado, y este modifica el valor de la variable que anteriormente mencionamos
- Cabe destacar que la función proporciona cómo parámetro el valor actual del propio estado. Ex: `setIsOpen(isOpen ⇒ !isOpen)`

En este ejemplo mostramos como el valor de `count` se inicializa en 0, y también se renderiza cada vez que el valor es modificado con la función `setCount` en el evento `onClick` del button:

```
import { useState } from 'react'function Counter() {
  const [count, setCount] = useState(0)
  return (
    <>      <p>Contador: {count}</p>      <button onClick={() => setCount(co
unt => count + 1)}>Aumentar</button>    </>  )
}
```

¿Qué significa la expresión “subir el estado”?

Cuando varios componentes necesitan compartir los mismos datos de un estado, entonces se recomienda elevar ese estado compartido hasta su ancestro común más cercano.

Dicho de otra forma. Si dos componentes hijos comparten los mismos datos de su padre, entonces mueve el estado al padre en lugar de mantener un estado local en sus hijos.

Para entenderlo, lo mejor es que lo veamos con un ejemplo. Imagina que tenemos una lista de regalos deseados y queremos poder añadir regalos y mostrar el total de regalos que hay en la lista.

```
import { useState } from 'react'export default function App() {
  return (
    <>      <h1>Lista de regalos</h1>      <GiftList />      <TotalGifts />    </>  )
}
function GiftList() {
  const [gifts, setGifts] = useState([])
  const addGift = () => {
    const newGift = prompt('¿Qué regalo quieres añadir?')
    setGifts([...gifts, newGift])
  }
  return (
    <>      <h2>Regalos</h2>      <ul>        {gifts.map(gift => (
          <li key={gift}>{gift}</li>        ))}      </ul>      <button onClick={addGif
t}>Añadir regalo</button>    </>  )
```

```
}
function TotalGifts() {
  const [totalGifts, setTotalGifts] = useState(0)
  return (
    <>      <h2>Total de regalos</h2>      <p>{totalGifts}</p>    </>  )
}
```

¿Qué pasa si queremos que el total de regalos se actualice cada vez que añadimos un regalo? Como podemos ver, no es posible porque el estado de `totalGifts` está en el componente `TotalGifts` y no en el componente `GiftList`. Y como no podemos acceder al estado de `GiftList` desde `TotalGifts`, no podemos actualizar el estado de `totalGifts` cuando añadimos un regalo.

Tenemos que subir el estado de `gifts` al componente padre `App` y le pasaremos el número de regalos como prop al componente `TotalGifts`.

```
import { useState } from 'react'export default function App() {
  const [gifts, setGifts] = useState([])
  const addGift = () => {
    const newGift = prompt('¿Qué regalo quieres añadir?')
    setGifts([...gifts, newGift])
  }
  return (
    <>      <h1>Lista de regalos</h1>      <GiftList gifts={gifts} addGift={addGi
ft} />      <TotalGifts totalGifts={gifts.length} />    </>  )
}
function GiftList({ gifts, addGift }) {
  return (
    <>      <h2>Regalos</h2>      <ul>        {gifts.map(gift => (
          <li key={gift}>{gift}</li>        ))}      </ul>      <button onClick={addGif
t}>Añadir regalo</button>    </>  )
}
function TotalGifts({ totalGifts }) {
  return (
    <>      <h2>Total de regalos</h2>      <p>{totalGifts}</p>    </>  )
}
```

Con esto, lo que hemos hecho es *elevar el estado*. Lo hemos movido desde el componente `GiftList` al componente `App`. Ahora pasamos como prop los regalos

al componente `GiftList` y una forma de actualizar el estado, y también hemos pasado como prop al componente `TotalGifts` el número de regalos.

¿Qué hace el hook `useEffect`?

El hook `useEffect` se usa para ejecutar código cuando se renderiza el componente o cuando cambian las dependencias del efecto.

Recibe dos parámetros:

- La función que se ejecutará al cambiar las dependencias o al renderizar el componente.
- Un array de dependencias. Si cambia el valor de alguna dependencia, ejecutará la función.

En este ejemplo mostramos un mensaje en consola cuando carga el componente y cada vez que cambia el valor de `count`:

```
import { useEffect, useState } from 'react'function Counter() {
  const [count, setCount] = useState(0)
  useEffect(() => {
    console.log('El contador se ha actualizado')
  }, [count])
  return (
    <>      <p>Contador: {count}</p>      <button onClick={() => setCount(co
unt + 1)}>Aumentar</button>    </>  )
}
```

Explica casos de uso del hook `useEffect`

Podemos usar el hook `useEffect` de diferentes formas, tales como:

- Ejecutar código cuando se renderiza el componente, cuando cambian las dependencias del efecto o cuando se desmonta el componente.
- Por eso puede ser útil para hacer llamadas a APIs, ya que sea nada más montar el componente o cuando cambian las dependencias.
- Realizar tracking de eventos, como Google Analytics, para saber qué páginas visitan los usuarios.

- Podemos validar un formulario para que cada vez que cambie el estado, podamos actualizar la UI y mostrar dónde están los errores.
- Podemos suscribirnos a eventos del navegador, como por ejemplo el evento `resize` para saber cuando el usuario cambia el tamaño de la ventana.

Cómo suscribirse a un evento en `useEffect`

Dentro de `useEffect` nos podemos suscribir a eventos del navegador, como el evento `resize` para saber cuando el usuario cambia el tamaño de la ventana. Es importante que nos desuscribamos cuando el componente se desmonte para evitar fugas de memoria. Para ello, tenemos que devolver una función dentro del `useEffect` que se ejecutará cuando el componente se desmonte.

```
import { useEffect } from 'react'function Window() {
  useEffect(() => {
    const handleResize = () => {
      console.log('La ventana se ha redimensionado')
    }
    window.addEventListener('resize', handleResize)
    return () => {
      window.removeEventListener('resize', handleResize)
    }
  }, [])
  return <p>Abre la consola y redimensiona la ventana</p>}
```

¿Qué hace el hook `useId`?

`useId` es un hook para generar identificadores únicos que se pueden pasar a los atributos de las etiquetas HTML y es especialmente útil para accesibilidad.

Llama `useId` en el nivel superior del componente para generar una ID única:

```
import { useId } from 'react'function PasswordField() {
  const passwordHintId = useId()
  // ...
```

A continuación, pasa el ID generado a diferentes atributos:

```
<> <input type="password" aria-describedby={passwordHintId} /> <p id ={passwordHintId}></>
```

La etiqueta `aria-describedby` te permite especificar que dos etiquetas están relacionadas entre sí, puede generar una identificación única con useId donde incluso si `PasswordField` aparece varias veces en la pantalla, las identificaciones generadas no chocarán.

El ejemplo completo sería así:

```
import { useId } from 'react'function PasswordField() {
 const passwordHintId = useId()
 return (
  <>  <label>  Password:
   <input type='password' aria-describedby={passwordHintId} />  </la bel>  <p id={passwordHintId}>  El password debe ser de 18 letras y c ontener caracteres especiales
  </p>  </> )
}
export default function App() {
 return (
  <>  <h2>Choose password</h2>  <PasswordField />  <h2>Confi rm password</h2>  <PasswordField />  </> )
}
```

Como ves en `App` estamos usando el componente dos veces. Si pusieramos una id a mano, por ejemplo `password`, entonces la ID no sería única y quedaría duplicada. Por eso es importante que generes la ID automáticamente con `useId`.

¿Cómo podemos ejecutar código cuando el componente se monta?

Podemos ejecutar código cuando el componente se monta usando el hook `useEffect` sin pasarle ninguna dependencia. En este caso, la función que se pasa como primer parámetro se ejecutará cuando el componente se monte.

```
import { useEffect } from 'react'function Component() {
 useEffect(() => {
  console.log('El componente se ha montado')
```

```
  }, [])
  return <p>Abre la consola y re-dimensiona la ventana</p>}
```

¿Qué son los Fragments en React?

Los *Fragments* son una forma de agrupar elementos sin añadir un elemento extra al DOM, ya que React no permite devolver varios elementos en un componente, solo un elemento raíz.

Para crear un Fragment en React usamos el componente `Fragment`:

```
import { Fragment } from 'react'function App() {
  return (
    <Fragment>      <h1>Titulo</h1>      <p>Párrafo</p>    </Fragment>  )
}
```

También podemos usar la sintaxis de abreviatura:

```
function App() {
  return (
    <>      <h1>Titulo</h1>      <p>Párrafo</p>    </>  )
}
```

¿Por qué es recomendable usar Fragment en vez de un div?

Las razones por las que es recomendable usar Fragment en vez de un `div` a la hora de envolver varios elementos son:

- Los `div` añaden un elemento extra al DOM, mientras que los Fragments no. Esto hace que el número de elementos HTML y la profundidad del DOM sea menor.
- Los elementos envueltos con Fragment son afectados directamente por las propiedades *flex* o *grid* de CSS de su elemento padre. Si usas un `div` es posible que tengas problemas con el alineamiento de los elementos.
- Los Fragments son más rápidos que los `div` ya que no tienen que ser renderizados.
- Los `div` aplican CSS por defecto (hace que lo que envuelve el `div` se comporte como un bloque al aplicar un `display: block`) mientras que los

Fragment no aplican ningún estilo por defecto.

¿Qué es el Compound Components Pattern?

Es un patrón de diseño de componentes que se basa en crear un componente padre con un solo objetivo, proporcionarle a sus hijos las propiedades necesarias para que se rendericen sin problemas.

Permite una estructura declarativa a la hora de construir nuevos componentes, además ayuda a la lectura del código por su simplicidad y limpieza.

Un ejemplo de este diseño sería una lista que renderiza los elementos hijos:

```
<List> <ListItem>Cat</ListItem> <ListItem>Dog</ListItem></List>
```

```
const List = ({ children, ...props }) => <ul {...props}>{children}</ul>const Lis
tItem = ({ children, ...props }) => {
  return <li {...props}>{children}</li>}
export { List, ListItem }
```

Este es un ejemplo sencillo, pero los componentes pueden ser tan complejos como quieras y tanto el padre como los hijos pueden tener sus propios estados.

Enlaces de interés:

- Lleva tu React al siguiente nivel con Compound Pattern by dezkareid en el blog de Platzi
- Compound Components by Jenna Smith **en inglés**
- Compound Components Lesson by Kent C. Dodds **en inglés**

¿Cómo puedes inicializar un proyecto de React desde cero?

Existen diversas formas de inicializar un proyecto de React desde cero. Entre las más populares están:

- Vite

```
npm create vite@latest my-app -- --template react
```

- Create React App

```
npx create-react-app my-app
```

> La opción más popular y recomendada hoy en día es Vite. Fuente npm trends.

Usando un Framework, entre las más populares están:

- Nextjs

```
npx create-next-app@latest my-app
```

- Gatsby

```
npm init gatsby
```

> La opción más popular y recomendada hoy en día es Nextjs. Fuente npm trends

Cada uno de ellos es un empaquetador de aplicaciones web. Se encargan de resolver las dependencias de tu proyecto, levantar un entorno de desarrollo que se refresca automáticamente con cada cambio y de empaquetar tu aplicación para producción con todos los archivos estáticos necesarios y mucho más.

¿Qué es React DOM?

React DOM es la librería que se encarga de renderizar los componentes de React para el navegador. Hay que tener en cuenta que React es una biblioteca que se puede usar en diferentes entornos (dispositivos móviles, apps de escritorio, terminal...).

Mientras que la biblioteca de *React*, a secas, es el motor de creación de componentes, hooks, sistema de props y estado... *React DOM* es la librería que se encarga de renderizar los componentes de React específicamente en el navegador.

React Native, por ejemplo, haría lo mismo, pero para dispositivos móviles.

¿Qué JavaScript necesito para aprender React?

JavaScript que necesitas para aprender React

Para aprender y dominar React necesitas saber JavaScript. A diferencia de otros frameworks y bibliotecas, como *Angular* y *Vue*, que se basan en su propio *DSL* (Domain-Specific Language), React usa una extensión de la sintaxis de JavaScript llamada *JSX*. Más adelante lo veremos en detalle pero, al final, no deja de ser azúcar sintáctico para escribir menos JavaScript.

En React todo es JavaScript. Para bien y para mal. Este libro da por sentados unos conocimientos previos del lenguaje de programación pero antes de empezar vamos a hacer un pequeño repaso por algunas de las características más importantes que necesitarás conocer.

Si ya dominas JavaScript puedes saltarte este capítulo y continuar con el libro, pero recuerda que siempre podrás revisar este capítulo como referencia.

EcmaScript Modules o ESModules

Los **EcmaScript Modules** es la forma nativa que tiene JavaScript para importar y exportar variables, funciones y clases entre diferentes ficheros. Hoy en día, especialmente si trabajamos con un empaquetador de aplicaciones como Webpack, vamos a estar trabajando constantemente con esta sintaxis.

Por un lado podemos crear módulos exportándolos por defecto:

```
// sayHi.js// exportamos por defecto el módulo sayHiexport default sayHi (message) {
   console.log(message)
}
// index.js// este módulo lo podremos importar con el nombre que queramo simport sayHi from './sayHi.js'// al ser el módulo exportado por defecto pod ríamos usar otro nombreimport EmiHi from './sayHi.js'
```

También podemos hacer exportaciones nombradas de módulos, de forma que un módulo tiene un nombre asignado y para importarlo necesitamos usar exactamente el nombre usado al exportarlo:

```
// sayHi.js// podemos usar exportaciones nombradas para mejorar estoexp ort const sayHi = message ⇒ console.log(message)
// y se pueden hacer tantas exportaciones de módulos nombrados como q
```

```
ueramosexport const anotherHi = msg => alert(msg)
// index.js// ahora para importar estos módulos en otro archivo podríamos
hacerlo asíimport { sayHi, anotherHi } from './sayHi.js'
```

Los *imports* que hemos visto hasta aquí se conocen como *imports estáticos*. Esto significa que ese módulo será cargado en el momento de la carga del archivo que lo importa.

También existen los *imports dinámicos*, de forma que podamos importar módulos que se carguen en el momento de la ejecución del programa o cuando nosotros decidamos (por ejemplo, como respuesta a un click).

```
document.querySelector('button').addEventListener('click', () => {
  // los imports dinámicos devuelven una Promesa  import('./sayHi.js').then
(module => {
    // ahora podemos ejecutar el módulo que hemos cargado    module.defa
ult('Hola')
  })
})
```

Los imports dinámicos son útiles también cuando trabajamos con empaquetadores como Webpack o Vite, ya que esto creará unos *chunks* (fragmentos) que se cargarán fuera del bundle general. ¿El objetivo? Mejorar el rendimiento de la aplicación.

Existen más sintaxis para trabajar con módulos, pero con saber las que hemos visto ya sería suficiente para seguir el libro.

¿Por qué es importante?

Para empezar React te ofrece diferentes partes de su biblioteca a través de módulos que podrás importar. Además nuestros componentes los tendremos separados en ficheros y, cada uno de ellos, se podrá importar utilizando *ESModules*.

Además, por temas de optimización de rendimiento, podremos importar de forma dinámica componentes y así mejorar la experiencia de nuestros usuarios al necesitar cargar menos información para poder utilizar la página.

Operador condicional (ternario)

Las ternarias son una forma de realizar condiciones sin la necesidad de usar la sintaxis con `if`. Se podría decir que es una forma de atajo para evitar escribir tanto código.

```
if (number % 2 === 0) {
  console.log('Es par')
} else {
  console.log('Es impar')
}
// usando ternarianumber % 2 === 0 ? console.log('Es par') : console.log
('Es impar')
```

¿Por qué es importante?

En las interfaces gráficas es muy normal que, dependiendo del estado de la aplicación o los datos que nos lleguen, vamos a querer renderizar una cosa u otra en pantalla. Para realizar esto, en lugar de utilizar `if` se usan las ternarias ya que queda mucho más legible dentro del *JSX*.

Funciones flecha o Arrow Functions

Las *funciones flecha* o *arrow function* fueron añadidas a JavaScript en el estándar ECMAScript 6 (o ES2015). En principio parece que simplemente se trata de una sintaxis alternativa más simple a la hora de crear expresiones de funciones:

```
const nombreDeLaFuncion = function (param1, param2) {
  // instrucciones de la función}
const nombreDeLaFuncion = (param1, param2) => {
  // con arrow function  // instrucciones de la función}
```

Pero además del cambio de sintaxis existen otras características de las funciones flechas que se usan constantemente en React.

```
// return implícito al escribir una sola líneaconst getName = () => 'EmiDev'//
ahorro de parentésis para función de un parámetroconst duplicateNumber
= num => num * 2// se usan mucho como callback en funciones de arraysc
onst numbers = [2, 4, 6]
```

```
const newNumbers = numbers.map(n => n / 2)
console.log(newNumbers) // [1, 2, 3]
```

También tiene algunos cambios respecto al valor de `this` pero, aunque es aconsejable dominarlo, no es realmente necesario para poder seguir con garantías el libro.

¿Por qué es importante?

Aunque hace unos años con React se trabajaba principalmente con clases, desde la irrupción de los hooks en la versión 16.8 ya no se usan mucho. Esto hace que se usen mucho más funciones.

Las funciones flecha, además, puedes verlas fácilmente conviviendo dentro de tus componentes. Por ejemplo, a la hora de renderizar una lista de elementos ejecutarás el método `.map` del array y, como callback, seguramente usarás una función flecha anónima.

Parámetros predeterminados (default values)

En JavaScript puedes proporcionar valores por defecto a los parámetros de una función en caso que no se le pase ningún argumento.

```
// al parámetro b le damos un valor por defecto de 1function multiply(a, b =
1) {
  return a * b
}
// si le pasamos un argumento con valor, se ignora el valor por defectocons
ole.log(multiply(5, 2)) // 10// si no le pasamos un argumento, se usa el valor
por defectoconsole.log(multiply(5)) // 5// las funciones flecha también pue
den usarlosconst sayHi = (msg = 'Hola React!') => console.log(msg)
sayHi() // 'Hola React!'
```

¿Por qué es importante?

En React existen dos conceptos muy importantes: **componentes y hooks**. No vamos a entrar en detalle ahora en ellos pero lo importante es que ambos son construidos con funciones.

Poder añadir valores por defecto a los parámetros de esas funciones en el caso que no venga ningún argumento **es clave** para poder controlar React con éxito.

Los componentes, por ejemplo, pueden no recibir parámetros y, pese a ello, seguramente vas a querer que tengan algún comportamiento por defecto. Lo podrás conseguir de esta forma.

Template Literals

Los template literals o plantillas de cadenas llevan las cadenas de texto al siguiente nivel permitiendo expresiones incrustadas en ellas.

```
const inicio = 'Hola'const final = 'React'// usando una concatenación norm
al seríaconst mensaje = inicio + ' ' + final
// con los template literals podemos evaluar expresionesconst mensaje = `
${inicio} ${final}`
```

Como ves, para poder usar los template literals, necesitas usar el símbolo ```

Además, nos permiten utilizar cadenas de texto de más de una línea.

¿Por qué es importante?

En React esto se puede utilizar para diferentes cosas. No sólo es normal crear cadenas de texto para mostrar en la interfaz... también puede ser útil para crear clases para tus elementos HTML de forma dinámica. Verás que los template literales están en todas partes.

Propiedades abreviadas

Desde *ECMAScript 2015* se puede iniciar un objeto utilizado nombre de propiedades abreviadas. Esto es que si quieres utilizar como valor una variable que tiene el mismo nombre que la key, entonces puedes indicar la inicialización una vez:

```
const name = 'Emiliano'const age = 36const book = 'React'// antes haríam
os estoconst persona = { name: name, age: age, book: book }
// ahora podemos hacer esto, sin repetirconst persona = { name, age, book
}
```

¿Por qué es importante?

En React se trata muchas veces con objetos y siempre vamos a querer escribir el menor número de líneas posible para mantener nuestro código fácil de mantener y entender.

La desestructuración

La sintaxis de *desestructuración* es una expresión de JavaScript que permite extraer valores de Arrays o propiedades de objetos en distintas variables.

```
// antesconst array = [1, 2, 3]
const primerNumero = array[0]
const segundoNumero = array[1]
// ahoraconst [primerNumero, segundoNumero] = array
// antes con objetosconst persona = { name: 'Emiliano', age: 36, book: 'Rea
ct' }
const name = persona.nameconst age = persona.age// ahora con objetosc
onst { age, name } = persona
// también podemos añadir valores por defectoconst { books = 2 } = perso
na
console.log(persona.books) // → 2// también funciona en funcionesconst g
etName = ({ name }) ⇒ `El nombre es ${name}`getName(persona)
```

¿Por qué es importante?

En React hay mucho código básico que da por sentado que conoces y dominas esta sintaxis. Piensa que los objetos y los arreglos son tipos de datos que son perfectos para guardar datos a representar en una interfaz. Así que poder tratarlos fácilmente te va a hacer la vida mucho más fácil.

Métodos de Array

Saber manipular arreglos en JavaScript es básico para considerar que se domina. Cada método realiza una operación en concreto y devuelve diferentes tipos de datos. Todos los métodos que veremos reciben un callback (función) que se ejecutará para cada uno de los elementos del array.

Vamos a revisar algunos de los métodos más usados:

```
// tenemos este array con diferentes elementosconst networks = [
  {
    id: 'youtube',    url: 'https://Emi.tube',    needsUpdate: true  },  {
    id: 'twitter',    url: 'https://twitter.com/EmiDev',    needsUpdate: true  },  {
    id: 'instagram',    url: 'https://instagram.com/Emi.dev',    needsUpdate: fal
se  }
]
```

```
// con .map podemos transformar cada elemento// y devolver un nuevo arr
aynetworks.map(singleNetwork ⇒ singleNetwork.url)
// Resultado:  [
   'https://Emi.tube',    'https://twitter.com/EmiDev',    'https://instagram.co
m/Emi.dev'  ]
// con .filter podemos filtrar elementos de un array que no// pasen una con
dición determinada por la función que se le pasa.// Devuelve un nuevo arra
y.networks.filter(singleNetwork ⇒ singleNetwork.needsUpdate === true)
// Resultado:[
  { id: 'youtube', url: 'https://Emi.tube', needsUpdate: true },  { id: 'twitter', ur
l: 'https://twitter.com/EmiDev', needsUpdate: true }
]
// con .find podemos buscar un elemento de un array que// cumpla la condi
ción definida en el callbacknetworks.find(singleNetwork ⇒ singleNetwork.i
d === 'youtube')
// Resultado:{ id: 'youtube', url: 'https://Emi.tube', needsUpdate: true }
// con .some podemos revisar si algún elemento del array cumple una cond
iciónnetworks.some(singleNetwork ⇒ singleNetwork.id === 'tiktok') // fals
enetworks.some(singleNetwork ⇒ singleNetwork.id === 'instagram') // tru
e
```

¿Por qué es importante?

En React es muy normal almacenar los datos que tenemos que representar en la UI como array. Esto hace que muchas veces necesitemos tratarlos, filtrarlos o extraer información de ellos. Es primordial entender, conocer y dominar al menos estos métodos, ya que son los más usados.

Sintaxis Spread

La sintaxis de spread nos permite expandir un iterable o un objeto en otro lugar dónde se espere esa información. Para poder utilizarlo, necesitamos utilizar los tres puntos suspensivos `...` justo antes.

```
const networks = ['Twitter', 'Twitch', 'Instagram']
const newNetwork = 'Tik Tok'// creamos un nuevo array expandiendo el ar
ray networks y// colocando al final el elemento newNetwork// utilizando la
sintaxis de spreadconst allNetworks = [...networks, newNetwork]
```

```
console.log(allNetworks)
// → [ 'Twitter', 'Twitch', 'Instagram', 'Tik Tok' ]
```

Esto mismo lo podemos conseguir con un objeto, de forma que podemos expandir todas sus propiedades en otro objeto de forma muy sencilla.

```
const Emi = { name: 'Emiliano', twitter: '@EmiDev' }
const EmiWithNewInfo = {
  ...Emi,  youtube: 'https://youtube.com/EmiDev',  books: ['Aprende React'],}
console.log(EmiWithNewInfo)
// {//   name: 'Emiliano',//   twitter: '@EmiDev',//   youtube: 'https://youtube.
com/EmiDev',//   books: [ 'Aprende React' ]// }
```

Es importante notar que esto hace una copia, sí, pero superficial. Si tuviéramos objetos anidados dentro del objeto entonces deberíamos tener en cuenta que podríamos mutar la referencia. Veamos un ejemplo.

```
const Emi = {
  name: 'Emiliano',  twitter: '@EmiDev',  experience: {
    years: 18,    focus: 'javascript',  },}
const EmiWithNewInfo = {
  ...Emi,  youtube: 'https://youtube.com/EmiDev',  books: ['Aprende React'],}
// cambiamos un par de propiedades de la "copia" del objetoEmiWithNewIn
fo.name = 'Emiliano Ángel'EmiWithNewInfo.experience.years = 19// hacem
os un console.log del objeto inicialconsole.log(Emi)
// en la consola veremos que el nombre no se ha modificado// en el objeto
original pero los años de experiencia sí// ya que hemos mutado la referenci
a original// {//   name: 'Emiliano',//   twitter: '@EmiDev',//   experience: { yea
rs: 19, focus: 'javascript' }// }
```

¿Por qué es importante?

En React es muy normal tener que añadir nuevos elementos a un array o crear nuevos objetos sin necesidad de mutarlos. El operador Rest nos puede ayudar a conseguir esto. Si no conoces bien el concepto de valor y referencia en JavaScript, sería conveniente que lo repases.

Operador Rest

La sintaxis `...` hace tiempo que funciona en JavaScript en los parámetros de una función. A esta técnica se le llamaba *parámetros rest* y nos permitía tener un número indefinido de argumentos en una función y poder acceder a ellos después como un array.

```
function suma(...allArguments) {
  return allArguments.reduce((previous, current) => {
    return previous + current
  })
}
```

Ahora el operador rest también se puede utilizar para agrupar el resto de propiedades un objeto o iterable. Esto puede ser útil para extraer un elemento en concreto del objeto o el iterable y crear una copia superficial del resto en una nueva variable.

```
const Emi = {
  name: 'Emiliano',  twitter: '@EmiDev',  experience: {
    years: 18,    focus: 'javascript',  },}
const { name, ...restOfEmi } = Emi
console.log(restOfEmi)
// → {//   twitter: '@EmiDev',//   experience: {//     years: 18,//     focus: 'javas
cript'//   }// }
```

También podría funcionar con arrays:

```
const [firstNumber, ...restOfNumbers] = [1, 2, 3]
console.log(firstNumber) // → 1console.log(restOfNumbers) // → [2, 3]
```

¿Por qué es importante?

Es una forma interesante de *eliminar* (de forma figurada) una propiedad de un objeto y creando una copia superficial del resto de propiedades. A veces puede ser interesante para extraer la información que queremos de unos parámetros y dejar el resto en un objeto que pasaremos hacia otro nivel.

Encadenamiento opcional (Optional Chaining)

El operador de encadenamiento opcional `?.` te permite leer con seguridad el valor de una propiedad que está anidada dentro de diferentes niveles de un

objeto.

De esta forma, en lugar de revisar si las propiedades existen para poder acceder a ellas, lo que hacemos es usar el encadenamiento opcional.

```
const author = {
  name: 'Emiliano',  libro: {
    name: 'Aprendiendo React',  },  writeBook() {
    return 'Writing!'  },}
// sin optional chainingauthor === null || author === undefined  ? undefine
d  : author.libro === null || author.libro === undefined    ? undefined    : auth
or.libro.name// con optional chainingauthor?.libro?.name
```

¿Por qué es importante?

Un objeto es una estructura de datos que es perfecta a la hora de representar muchos elementos de la UI. ¿Tienes un artículo? Toda la información de un artículo seguramente la tendrás representada en un objeto.

Conforme tu UI sea más grande y compleja, estos objetos tendrán más información y necesitarás dominar el encadenamiento opcional `?.` para poder acceder a su información con garantías.

Intermedio

¿Cómo crear un hook personalizado (*custom hook*)?

Un hook personalizado es una función que empieza con la palabra `use` y que puede utilizar otros hooks. Son ideales para reutilizar lógica en diferentes componentes. Por ejemplo, podemos crear un hook personalizado para extraer la gestión del estado de un contador:

```
// ./hooks/useCounter.jsexport function useCounter() {
  const [count, setCount] = useState(0)
  const increment = () => setCount(count + 1)
  const decrement = () => setCount(count - 1)
  return { count, increment, decrement }
}
```

Para usarlo en un componente:

```
import { useCounter } from './hooks/useCounter.js'function Counter() {
  const { count, increment, decrement } = useCounter()
  return (
    <>      <button onClick={decrement}>-</button>      <span>{count}</spa
n>      <button onClick={increment}>+</button>    </>  )
}
```

¿Cuántos `useEffect` puede tener un componente?

Aunque normalmente los componentes de React solo cuentan con un `useEffect` lo cierto es que podemos tener tantos `useEffect` como queramos en un componente. Cada uno de ellos se ejecutará cuando se renderice el componente o cuando cambien las dependencias del efecto.

¿Cómo podemos ejecutar código cuando el componente se desmonta del árbol?

Podemos ejecutar código cuando el componente se desmonta usando el hook `useEffect` y dentro devolver una función con el código que queremos ejecutar. En este caso, la función que se pasa como primer parámetro del `useEffect` se ejecutará cuando el componente se monte, y la función que es retornada se ejecutará cuando se desmonte.

```
import { useEffect } from 'react'function Component() {
  useEffect(() => {
    console.log('El componente se ha montado')
    return () => {
      console.log('El componente se ha desmontado')
    }
  }, [])
  return <h1>Ejemplo</h1>}
```

Esto es muy útil para limpiar recursos que se hayan creado en el componente, como por ejemplo, eventos del navegador o para cancelar peticiones a APIs.

Cómo puedes cancelar una petición a una API en `useEffect` correctamente

Cuando hacemos una petición a una API, podemos cancelarla para evitar que se ejecute cuando el componente se desmonte usando `AbortController` como hacemos en este ejemplo:

```
useEffect(() => {
  // Creamos el controlador para abortar la petición  const controller = new AbortController()
  // Recuperamos la señal del controlador  const { signal } = controller
  // Hacemos la petición a la API y le pasamos como options la señal  fetch('https://jsonplaceholder.typicode.com/posts/1', { signal })
    .then(res => res.json())
    .then(json => setMessage(json.title))
    .catch(error => {
      // Si hemos cancelado la petición, la promesa se rechaza      // con un error de tipo AbortError      if (error.name !== 'AbortError') {
        console.error(error.message)
      }
    })
  // Si se desmonta el componente, abortamos la petición  return () => controller.abort()
}, [])
```

Esto también funciona con `axios`:

```
useEffect(() => {
  // Creamos el controlador para abortar la petición  const controller = new AbortController()
  // Recuperamos la señal del controlador  const { signal } = controller
  // Hacemos la petición a la API y le pasamos como options la señal  axios
    .get('https://jsonplaceholder.typicode.com/posts/1', { signal })
    .then(res => setMessage(res.data.title))
    .catch(error => {
      // Si hemos cancelado la petición, la promesa se rechaza      // con un error de tipo AbortError      if (error.name !== 'AbortError') {
        console.error(error.message)
      }
    })
  // Si se desmonta el componente, abortamos la petición  return () => contr
```

```
oller.abort()
}, [])
```

¿Cuáles son las reglas de los hooks en React?

Los hooks en React tienen dos reglas fundamentales:

- Los hooks solo se pueden usar en componentes funcionales o *custom hooks*.
- Los hooks solo se pueden llamar en el nivel superior de un componente. No se pueden llamar dentro de bucles, condicionales o funciones anidadas.

¿Qué diferencia hay entre `useEffect` y `useLayoutEffect`?

Aunque ambos son muy parecidos, tienen una pequeña diferencia en el momento en el que se ejecutan.

`useLayoutEffect` se ejecuta de forma síncrona inmediatamente después que React haya actualizado completamente el DOM tras el renderizado. Puede ser útil si necesitas recuperar un elemento del DOM y acceder a sus dimensiones o posición en pantalla.

`useEffect` se ejecuta de forma asíncrona tras el renderizado, pero no asegura que el DOM se haya actualizado. Es decir, si necesitas recuperar un elemento del DOM y acceder a sus dimensiones o posición en pantalla, no podrás hacerlo con `useEffect` porque no tienes la garantía de que el DOM se haya actualizado.

Normalmente, el 99% de las veces, vas a querer utilizar `useEffect` y, además, tiene mejor rendimiento, ya que no bloquea el renderizado.

¿Qué son mejores los componentes de clase o los componentes funcionales?

Desde que en *React 16.8.0* se incluyeron los hooks, los componentes de funciones pueden hacer casi todo lo que los componentes de clase.

Aunque no hay una respuesta clara a esta pregunta, normalmente los componentes funcionales son más sencillos de leer y escribir y pueden tener un mejor rendimiento en general.

Además, **los hooks solo se pueden usar en los componentes funcionales**. Esto es importante, ya que con la creación de custom hooks podemos reutilizar la lógica y podría simplificar nuestros componentes.

Por otro lado, los componentes de clase nos permiten usar el ciclo de vida de los componentes, algo que no podemos hacer con los componentes funcionales donde solo podemos usar `useEffect`.

¿Cómo mantener los componentes puros y qué ventajas tiene?

Los componentes puros son aquellos que no tienen estado y que no tienen efectos secundarios. Esto quiere decir que no tienen ningún tipo de lógica que no sea la de renderizar la interfaz.

Son más fáciles de testear y de mantener. Además, son más fáciles de entender porque no tienen lógica compleja.

Para crear un componente puro en React usamos una function:

```
function Button({ text }) {
  return <button>{text}</button>}
```

En este caso, el componente `Button` recibe una prop `text` que es un string. El componente `Button` renderiza un botón con el texto que recibe en la prop `text`.

¿Qué es la hidratación (hydration) en React?

Cuando renderizamos nuestra aplicación en el servidor, React genera un HTML estático. Este HTML estático es simplemente un string que contiene el HTML que se va a mostrar en la página.

Cuando el navegador recibe el HTML estático, lo renderiza en la página. Sin embargo, este HTML estático no tiene interactividad. No tiene eventos, no tiene lógica, no tiene estado, etc. Podríamos decir que *no tiene vida*.

Para hacer que este HTML estático pueda ser interactivo, React necesita que el HTML estático se convierta en un árbol de componentes de React. Esto se llama **hidratación**.

De esta forma, en el cliente, React reutiliza este HTML estático y se dedica a adjuntar los eventos a los elementos, ejecutar los efectos que tengamos en los componentes y conciliar el estado de los componentes.

¿Qué es el Server Side Rendering y qué ventajas tiene?

El *Server Side Rendering* es una técnica que consiste en renderizar el HTML en el servidor y enviarlo al cliente. Esto nos permite que el usuario vea la interfaz de la aplicación antes de que se cargue el JavaScript.

Esta técnica nos permite mejorar la experiencia de usuario y mejorar el SEO de nuestra aplicación.

¿Cómo puedes crear un Server Side Rendering con React desde cero?

Para crear un Server Side Rendering con React desde cero podemos usar el paquete `react-dom/server` que nos permite renderizar componentes de React en el servidor.

Veamos un ejemplo de cómo crear un Server Side Rendering con React desde cero con Express:

```
import express from 'express'import React from 'react'import { renderToStr
ing } from 'react-dom/server'const app = express()
app.get('/', (req, res) => {
  const html = renderToString(<h1>Hola mundo</h1>)
  res.send(html)
})
```

Esto nos devolverá el HTML de la aplicación al acceder a la ruta `/`.

```
<h1 data-reactroot="">Hola mundo</h1>
```

¿Puedes poner un ejemplo de efectos colaterales en React?

Igual que las funciones en JavaScript, los componentes de React también pueden tener *side effects* (efectos colaterales). Un efecto colateral significa que el componente manipula o lee información que no está dentro de su ámbito.

Aquí puedes ver un ejemplo simple de un componente que tiene un efecto colateral. Un componente que lee y modifica una variable que está fuera del componente. Esto hace que sea imposible saber qué renderizará el

componente cada vez que se use, ya que no sabemos el valor que tendrá `count`:

```
let count = 0function Counter() {
  count = count + 1  return (
    <p>Contador: {count}</p>  )
}
export default function Counters() {
  return (
    <>     <Counter />     <Counter />     <Counter />    </>  )
```

¿Qué diferencia hay entre componentes controlados y no controlados? ¿Qué ventajas y desventajas tienen?

A la hora de trabajar con formularios en React, tenemos dos tipos de componentes: los componentes controlados y los componentes no controlados.

Componentes controlados:
son aquellos que tienen un estado que controla el valor del componente. Por lo tanto, el valor del componente se actualiza cuando el estado cambia.

La ventaja de este tipo de componentes es que son más fáciles de testear porque no dependen de la interfaz. También nos permiten crear validaciones muy fácilmente. La desventaja es que son más complejos de crear y mantener. Además, pueden tener un peor rendimiento, ya que provocan un re-renderizado cada vez que cambia el valor del input.

Componentes no controlados: son aquellos que no tienen un estado que controle el valor del componente. El estado del componente lo controla el navegador de forma interna. Para conocer el valor del componente, tenemos que leer el valor del DOM.

La ventaja de este tipo de componentes es que se crean de forma muy fácil y no tienes que mantener un estado. Además, el rendimiento es mejor, ya que no tiene que re-renderizarse al cambiar el valor del input. Lo malo es que hay que tratar más con el DOM directamente y crear código imperativo.

```
// Controlado:const [value, setValue] = useState('')
const handleChange = () => setValue(event.target.value)
<input type="text" value={value} onChange={handleChange} />// No contr
```

```
olado:<input type="text" defaultValue="foo" ref={inputRef} />// Usamos `in
putRef.current.value` para leer el valor del input
```

¿Qué son los High Order Components (HOC)?

Los High Order Components son funciones que reciben un componente como parámetro y devuelven un componente.

```
function withLayout(Component) {
  return function (props) {
    return (
      <main>      <section>      <Component {...props} />      </section>
</main>    )
  }
}
```

En este caso, la función `withLayout` recibe un componente como parámetro y devuelve un componente. El componente devuelto renderiza el componente que se le pasa como parámetro dentro de un layout.

Es un patrón que nos permite reutilizar código y así podemos inyectar funcionalidad, estilos o cualquier otra cosa a un componente de forma sencilla.

Con la llegada de los hooks, los HOCs se han vuelto menos populares, pero todavía se usan en algunos casos.

¿Qué son las render props?

Son un patrón de diseño de React que nos permite reutilizar código entre componentes e inyectar información en el renderizado de los componentes.

```
<DataProvider render={data => <h1>Hello {data.target}</h1>} />
```

En este caso, el componente `DataProvider` recibe una función `render` como prop. Ahí le indicamos qué es lo que debe renderizar usando la información que recibe como parámetro.

La implementación del `DataProvider` con funciones podría ser la siguiente:

```
function DataProvider({ render }) {
  const data = { target: 'world' }
```

```
  return render(data)
}
```

También se puede encontrar este patrón usando la prop `children` en los componentes.

```
<DataProvider>{data => <h1>Hello {data.target}</h1>}</DataProvider>
```

Y la implementación sería similar:

```
function DataProvider({ children }) {
  const data = { target: 'world' }
  return children(data)
}
```

Este patrón es usado por grandes bibliotecas como `react-router`, `formik` o `react-motion`.

¿Por qué no podemos usar un `if` en el renderizado de un componente?

En React, no podemos usar un `if` en el renderizado de un componente porque no es una expresión válida de JavaScript, es una declaración. Las expresiones son aquellas que devuelven un valor y las declaraciones no devuelven ningún valor.

En JSX solo podemos usar expresiones, por eso usamos ternarias, que sí son expresiones.

```
// ❌ Esto no funcionafunction Button({ text }) {
  return (
    <button>     {if (text) { return text } else { return 'Click' }}    </button>  )
}
// ✅ Esto funcionafunction Button({ text }) {
  return (
    <button>     {text ? text : 'Click'}    </button>  )
}
```

De la misma forma, tampoco podemos usar `for`, `while` o `switch` dentro del renderizado de un componente.

¿Por qué debemos utilizar una función para actualizar el estado de React?

A la hora de actualizar el estado de React, debemos utilizar la función que nos facilita el hook `useState` para actualizar el estado.

```
const [count, setCount] = useState(0)
setCount(count + 1)
```

¿Por qué es esto necesario? En primer lugar, el estado en React debe ser inmutable. Es decir, no podemos modificar el estado directamente, sino que debemos siempre crear un nuevo valor para el nuevo estado.

Esto nos permite que la integridad de la UI respecto a los datos que renderiza siempre es correcta.

Por otro lado, llamar a una función le permite a React saber que el estado ha cambiado y que debe re-renderizar el componente si es necesario. Además esto lo hace de forma asíncrona, por lo que podemos llamar a `setCount` tantas veces como queramos y React se encargará de actualizar el estado cuando lo considere oportuno.

¿Qué es el ciclo de vida de un componente en React?

En los componentes de clase, el ciclo de vida de un componente se divide en tres fases:

- Montaje: cuando el componente se añade al DOM.
- Actualización: cuando el componente se actualiza.
- Desmontaje: cuando el componente se elimina del DOM.

Dentro de este ciclo de vida, existe un conjunto de métodos que se ejecutan en el componente.

Estos métodos se definen en la clase y se ejecutan en el orden que se muestran a continuación:

- constructor
- render

- componentDidMount
- componentDidUpdate
- componentWillUnmount

En cada uno de estos métodos podemos ejecutar código que nos permita controlar el comportamiento de nuestro componente.

¿Por qué puede ser mala práctica usar el `index` como key en un listado de React?

Cuando renderizamos una lista de elementos, React necesita saber qué elementos han cambiado, han sido añadidos o eliminados.

Para ello, React necesita una key única para cada elemento de la lista. Si no le pasamos una key, React usa el índice del elemento como key.

```
const List = () => {
  const [items, setItems] = useState(['Item 1', 'Item 2', 'Item 3'])
  return (
    <ul>     {items.map((item, index) => (
       <li key={index}>{item}</li>      ))}    </ul>  )
}
```

En este caso, React usa el índice del elemento como `key`. Esto puede ser un problema si la lista se reordena o se eliminan elementos del array, ya que el índice de los elementos cambia.

En este caso, React no sabe qué elementos han cambiado y puede que se produzcan errores.

Un ejemplo donde se ve el problema:

```
const List = () => {
  const [items, setItems] = useState(['Item 1', 'Item 2', 'Item 3'])
  const handleRemove = index => {
    const newItems = [...items]
    newItems.splice(index, 1)
    setItems(newItems)
  }
  return (
    <ul>     {items.map((item, index) => (
```

```
      <li key={index}>        {item}        <button onClick={() => handleRemo
ve(index)}>Eliminar</button>       </li>     ))}   </ul>  )
}
```

¿Para qué sirve el hook `useMemo`?

El hook `useMemo` es un hook que nos permite memorizar el resultado de una función. Esto quiere decir que si la función que le pasamos como parámetro no ha cambiado, no se ejecuta de nuevo y se devuelve el resultado que ya se había calculado.

```
import { useMemo } from 'react'function Counter({ count }) {
  const double = useMemo(() => count * 2, [count])
  return (
    <div>     <p>Contador: {count}</p>     <p>Doble: {double}</p>    </div
> )
}
```

En este caso, el componente `Counter` recibe una prop `count` que es un número. El componente calcula el doble de ese número y lo muestra en pantalla.

El hook `useMemo` recibe dos parámetros: una función y un array de dependencias. La función se ejecuta cuando el componente se renderiza por primera vez y cuando alguna de las dependencias cambia, en este ejemplo la prop `count`.

La ventaja es que si la prop `count` no cambia, se evita el cálculo del doble y se devuelve el valor que ya se había calculado previamente.

¿Es buena idea usar siempre `useMemo` para optimizar nuestros componentes?

No. `useMemo` es una herramienta que nos permite optimizar nuestros componentes, pero no es una herramienta mágica que nos va a hacer que nuestros componentes sean más rápidos. A veces el cálculo de un valor es tan rápido que no merece la pena memorizarlo. Incluso, en algunos casos, puede ser más lento memorizarlo que calcularlo de nuevo.

¿Para qué sirve el hook `useCallback`?

El hook `useCallback` es un hook que nos permite memorizar una función. Esto quiere decir que si la función que le pasamos como parámetro no ha cambiado, no se ejecuta de nuevo y se devuelve la función que ya se había calculado.

```
import { useCallback } from 'react'function Counter({ count, onIncrement })
{
  const handleIncrement = useCallback(() => {
    onIncrement(count)
  }, [count, onIncrement])
  return (
    <div>     <p>Contador: {count}</p>     <button onClick={handleIncreme
nt}>Incrementar</button>    </div>  )
}
```

En este caso, el componente `Counter` recibe una prop `count` que es un número y una prop `onIncrement` que es una función que se ejecuta cuando se pulsa el botón.

El hook `useCallback` recibe dos parámetros: una función y un array de dependencias. La función se ejecuta cuando el componente se renderiza por primera vez y cuando alguna de las dependencias cambia, en este ejemplo la prop `count` o la prop `onIncrement`.

La ventaja es que si la prop `count` o la prop `onIncrement` no cambian, se evita la creación de una nueva función y se devuelve la función que ya se había calculado previamente.

¿Es buena idea usar siempre `useCallback` para optimizar nuestros componentes?

No. `useCallback` es una herramienta que nos permite optimizar nuestros componentes, pero no es una herramienta mágica que nos va a hacer que nuestros componentes sean más rápidos. A veces la creación de una función es tan rápida que no merece la pena memorizarla. Incluso, en algunos casos, puede ser más lento memorizarla que crearla de nuevo.

¿Cuál es la diferencia entre `useCallback` y `useMemo`?

La diferencia entre `useCallback` y `useMemo` es que `useCallback` memoriza una función y `useMemo` memoriza el resultado de una función.

En cualquier caso, en realidad, `useCallback` es una versión especializada de `useMemo`. De hecho se puede simular la funcionalidad de `useCallback` con `useMemo`:

```
const memoizedCallback = useMemo(() => {
  return () => {
    doSomething(a, b)
  }
}, [a, b])
```

¿Qué son las refs en React?

Las refs nos permiten crear una referencia a un elemento del DOM o a un valor que se mantendrá entre renderizados. Se pueden declarar por medio del comando `createRef` o con el hook `useRef`.

¿Cómo funciona el hook `useRef`?

En el siguiente ejemplo vamos a guardar la referencia en el DOM a un elemento `<input>` y vamos a cambiar el foco a ese elemento cuando hacemos clic en el botón.

```
import { useRef } from 'react'function TextInputWithFocusButton() {
  const inputEl = useRef(null)
  const onButtonClick = () => {
    // `current` apunta al elemento inputEl montado    inputEl.current.focus()
  }
  return (
    <>      <input ref={inputEl} type='text' />      <button onClick={onButtonCl
ick}>Focus the input</button>    </>  )
}
```

Creamos una referencia `inputEl` con `useRef` y la pasamos al elemento `<input>` como prop `ref`. Cuando el componente se monta, la referencia `inputEl` apunta al elemento `<input>` del DOM.

Para acceder al elemento del DOM, usamos la propiedad `current` de la referencia.

¿Qué hace el hook `useLayoutEffect`?

`useLayoutEffect` funciona igual que el hook `useEffect`, con la excepción de que este se dispara sincrónicamente después de leer todas las mutaciones del DOM.

Llama `useLayoutEffect` en el nivel superior del componente.

```
import { useLayoutEffect } from 'react'useLayoutEffect(() => {
  return () => {}
}, [])
```

`useLayoutEffect` recibe dos argumentos:

- Una función callback que define el efecto.
- Una lista de dependencias.

Aunque el `useEffect` es el hook de renderizado más usado, si se necesita que los efectos del DOM muten cambiando la apariencia entre el efecto y el renderizado, entonces es conveniente que uses el `useLayoutEffect`.

Orden de ejecución del `useLayoutEffect`

El orden de ejecución del `useLayoutEffect`, ya que se ejecuta de forma síncrona, al momento en que React termina de ejecutar todas las mutaciones, pero antes de renderizarlo en pantalla, es el siguiente:

- El componente se actualiza por algún cambio de estado, props o el padre se re-renderiza
- React renderiza el componente
- Tu efecto es ejecutado
- La pantalla se actualiza “visualmente”

¿Qué son los componentes *stateless*?

Los componentes *stateless* son componentes que no tienen estado. Estos componentes se crean con una `function` y no tienen acceso al estado de la aplicación. La ventaja que tienen estos componentes es que hace que sea más fácil crear componentes puros (que siempre renderizan lo mismo para unas mismas props).

```
// Este es un ejemplo de componente statelessfunction Button({ text }) {
  return <button>{text}</button>}
```

¿Cómo puedes prevenir el comportamiento por defecto de un evento en React?

Para prevenir el comportamiento por defecto de un evento en React, debemos usar el método `preventDefault`:

```
function Form({ onSubmit }) {
  const handleSubmit = event => {
    event.preventDefault()
    onSubmit()
  }
  return (
    <form onSubmit={handleSubmit}>      <input type='text' />      <button ty
pe='submit'>Enviar</button>    </form>  )
}
```

¿Qué es el `StrictMode` en React?

El `StrictMode` es un componente que nos permite activar algunas comprobaciones de desarrollo en React. Por ejemplo, detecta componentes que se renderizan de forma innecesaria o funcionalidades obsoletas que se están usando.

```
import { StrictMode } from 'react'function App() {
  return (
    <StrictMode>      <Component />    </StrictMode>  )
}
```

¿Por qué es recomendable exportar los componentes de React de forma nombrada?

Los componentes de React se pueden exportar de dos formas:

- Exportación por defecto
- Exportación nombrada

Para exportar un componente por defecto, usamos la palabra reservada `default`:

```
// button.jsxexport default function Button() {
  return <button>Click</button>}
// App.jsximport Button from './button.jsx'function App() {
  return <Button />}
```

La gran desventaja que tiene la exportación por defecto es que a la hora de importarlo puedes usar el nombre que quieras. Y esto trae problemas, ya que puedes no usar siempre el mismo en el proyecto o usar un nombre que no sea correcto con lo que importas.

```
// button.jsxexport default function Button() {
  return <button>Click</button>}
// App.jsximport MyButton from './button.jsx'function App() {
  return <MyButton />}
// Otro.jsximport Button from './button.jsx'function Otro() {
  return <Button />}
```

Los exports nombrados nos obligan a usar el mismo nombre en todos los archivos y, por tanto, nos aseguramos de que siempre estamos usando el nombre correcto.

```
// button.jsxexport function Button() {
  return <button>Click</button>}
// App.jsximport { Button } from './button.jsx'function App() {
  return <Button />}
```

¿Cómo puedes exportar múltiples componentes de un mismo archivo?

Para exportar múltiples componentes de un mismo archivo, podemos usar la exportación nombrada:

```
// button.jsxexport function Button({ children }) {
  return <button>{children}</button>}
export function ButtonSecondary({ children }) {
  return <button class='btn-secondary'>{children}</button>}
```

¿Cómo puedo importar de forma dinámica un componente en React?

Para importar de forma dinámica un componente en React debemos usar la función `import()`, el método `lazy()` de React y el componente `Suspense`.

```
// App.jsximport { lazy, Suspense } from 'react'const Button = lazy(() => imp
ort('./button.jsx'))
export default function App() {
  return (
    <Suspense fallback={<div>Cargando botón...</div>}>      <Button />
</Suspense>  )
}
// button.jsxexport default function Button() {
  return <button>Botón cargado dinámicamente</button>}
```

Vamos a ver en detalle cada uno de los elementos que hemos usado:

La función `import()` es parte del estándar de ECMAScript y nos permite importar de forma dinámica un módulo. Esta función devuelve una promesa que se resuelve con el módulo importado.

El método `lazy()` de React nos permite crear un componente que se renderiza de forma diferida. Este método recibe una función que debe devolver una promesa que se resuelve con un componente. En este caso, se resolverá con el componente que tenemos en el fichero `button.jsx`. Ten en cuenta que el componente que devuelve `lazy()` **debe ser un componente de React y ser exportado por defecto** (`export default`).

El componente `Suspense` nos permite mostrar un mensaje mientras se está cargando el componente. Este componente recibe una prop `fallback` que es el mensaje que se muestra mientras se está cargando el componente.

¿Cuando y por qué es recomendable importar componentes de forma dinámica?

En React, nuestras aplicaciones están creadas a partir de componentes. Estos componentes se pueden importar de forma **estática o dinámica**.

La importación de componentes de forma estática es la forma más común de importar componentes en React. En este caso, los componentes se importan en la parte superior del fichero y se renderizan en el código. El problema es

que, si siempre lo hacemos así, es bastante probable que estemos cargando componentes que no se van a usar desde el principio.

```
import { useState } from 'react'// importamos de forma estática el compone
nte de la Modalimport { SuperBigModal } from './super-big-modal.jsx'// mo
strar modal si el usuario da click en un botónexport default function App() {
  const [showModal, setShowModal] = useState(false)
  return (
    <div>      <button onClick={() => setShowModal(true)}>Mostrar modal</b
utton>      {showModal && <SuperBigModal />}     </div>  )
}
```

Este componente `SuperBigModal` se importa de forma estática, por lo que se carga desde el principio. Pero, ¿qué pasa si el usuario no da click en el botón para mostrar la modal? En este caso, está cargando el componente pese a que no lo está usando.

Si queremos ofrecer la mejor experiencia a nuestros usuarios, debemos intentar que la aplicación cargue lo más rápido posible. Por eso, es recomendable importar de forma dinámica los componentes que no se van a usar desde el principio.

```
import { useState, lazy, Suspense } from 'react'// importamos de forma din
ámica el componente de la Modalconst SuperBigModal = lazy(() => import
('./super-big-modal.jsx'))
// mostrar modal si el usuario da click en un botónexport default function A
pp() {
  const [showModal, setShowModal] = useState(false)
  return (
    <div>      <button onClick={() => setShowModal(true)}>Mostrar modal</b
utton>      <Suspense fallback={<div>Cargando modal...</div>}>        {sho
wModal && <SuperBigModal />}      </Suspense>     </div>  )
}
```

De esta forma, la parte de código que importa el componente `SuperBigModal` se carga de forma dinámica, es decir, cuando el usuario da click en el botón para mostrar la modal.

Dependiendo del empaquetador de aplicaciones web que uses y su configuración, es posible que el resultado de la carga sea diferente (algunos

creará un archivo a parte del *bundle* principal, otros podrían hacer un streaming del HTML...) pero la intención del código es la misma.

Así que siempre debemos intentar cargar los componentes de forma dinámica cuando no se vayan a usar desde el principio, sobretodo cuando están detrás de la interacción de un usuario. Lo mismo podría ocurrir con rutas completas de nuestra aplicación. ¿Por qué cargar la página de *About* si el usuario está visitando la página principal?

¿Sólo se pueden cargar componentes de forma dinámica si se exportan por defecto?

No, no es necesario que los componentes se exporten por defecto para poder cargarlos de forma dinámica. Podemos exportarlos de forma nombrada y cargarlos de forma dinámica... pero no es lo más recomendable ya que el código necesario es mucho más lioso.

```
// button.jsx// exportamos el componente Button de forma nombradaexport function Button() {
  return <button>Botón cargado dinámicamente</button>}
// app.jsximport { lazy, Suspense } from 'react'// Al hacer el import dinámico, debemos especificar el nombre del componente que queremos importar// y hacer que devuelva un objeto donde la key default pasar a ser el componente nombradoconst Button = lazy(() =>  import('./button.jsx').then(({ Button }) => ({ default: Button }))
)
export default function App() {
  return (
    <div>      <Suspense fallback={<div>Cargando botón...</div>}>        <Button />      </Suspense>    </div>  )
}
```

Otra opción es tener un fichero intermedio que exporte el componente de forma por defecto y que sea el que importemos de forma dinámica.

```
// button-component.jsx// exportamos el componente Button de forma nombradaexport function Button() {
  return <button>Botón cargado dinámicamente</button>}
// button.jsxexport { Button as default } from './button-component.jsx'// ap
```

```
p.jsximport { lazy, Suspense } from 'react'const Button = lazy(() => import
('./button.jsx'))
export default function App () {
 return (
  <div>    <Suspense fallback={<div>Cargando botón...</div>}>      <Bu
tton />    </Suspense>   </div> )
}
```

¿Qué es el contexto en React? ¿Cómo puedo crearlo y consumirlo?

El contexto es una forma de pasar datos a través de la jerarquía de componentes sin tener que pasar props manualmente en cada nivel.

Para crear un contexto en React usamos el hook `createContext`:

```
import { createContext } from 'react'const ThemeContext = createContext
()
```

Para usar el contexto, debemos envolver el árbol de componentes con el componente `Provider`:

```
<ThemeContext.Provider value='dark'> <App /></ThemeContext.Provider
>
```

Para consumir el contexto, debemos usar el hook `useContext`:

```
import { useContext } from 'react'function Button() {
 const theme = useContext(ThemeContext)
 return <button className={theme}>Haz clic aquí</button>}
```

¿Qué es el `SyntheticEvent` en React?

El `SyntheticEvent` es una abstracción del evento nativo del navegador. Esto le permite a React tener un comportamiento consistente en todos los navegadores.

Dentro del `SyntheticEvent` puede encontrarse una referencia al evento nativo en su atributo `nativeEvent`

```
function App() {
  function handleClick(event) {
    console.log(event)
  }
  return <button onClick={handleClick}>Haz clic aquí</button>}
```

¿Qué es `flushSync` en React?

`flushSync(callback)` Obliga a React a ejecutar de manera síncrona todas las actualizaciones de los state dentro del callback proporcionado. Así se asegura que el DOM se actualiza inmediatamente.

```
import { flushSync } from 'react-dom'function App() {
  const handleClick = () => {
    setId(1)
    // component no hace re-render 🚫    flushSync(() => {
      setId(2)
      // component re-renderiza aquí 🔄    })
    // component ha sido re-renderizado y el DOM ha sido actualizado ✅    f
lushSync(() => {
      setName('John')
      // component no hace re-render 🚫      setEmail('john@doe.com')
      // component re-renderiza aquí 🔄    })
    // component ha sido re-renderizado y el DOM ha sido actualizado ✅  }
  return <button onClick={handleClick}>Haz clic aquí</button>}
```

NOTA: `flushSync` puede afectar significativamente el rendimiento. Úsalo con moderación.

¿Qué son los Error Boundaries en React?

Los Error Boundaries son componentes que nos permiten manejar los errores que se producen en el árbol de componentes. Para crear un Error Boundary, debemos crear un componente que implemente el método `componentDidCatch`:

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props)
```

```
    this.state = { hasError: false }
  }
  static getDerivedStateFromError(error) {
    return { hasError: true }
  }
  componentDidCatch(error, errorInfo) {
    console.log(error, errorInfo)
  }
  render() {
    if (this.state.hasError) {
      return <h1>Algo ha ido mal</h1>    }
    return this.props.children  }
}
```

De esta forma podemos capturar los errores que se producen en el árbol de componentes y mostrar un mensaje de error personalizado mientras evitamos que nuestra aplicación se rompa completamente.

Ahora podemos envolver el árbol de componentes con el componente `ErrorBoundary`:

```
<ErrorBoundary> <App /></ErrorBoundary>
```

Podemos crear un Error Boundary en cualquier nivel del árbol de componentes, de esta forma podemos tener un control más granular de los errores.

```
<ErrorBoundary> <App /> <ErrorBoundary>  <SpecificComponent /> </
ErrorBoundary></ErrorBoundary>
```

Por ahora no existe una forma nativa de crear un Error Boundary en una función de React. Para crear un Error Boundary en una función, puedes usar la librería react-error-boundary.

¿Qué son las Forward Refs?

El reenvío de referencia o *Forward Refs* es una técnica que nos permite acceder a una referencia de un componente hijo desde un componente padre.

```
// Button.jsximport { forwardRef } from 'react'export const Button = forward
Ref((props, ref) => (
  <button ref={ref} className='rounded border border-sky-500 bg-whit
e'>    {props.children}  </button>))
// Parent.jsximport { Button } from './Button'import { useRef } from 'react'co
nst Parent = () => {
  const ref = useRef()
  useEffect(() => {
    // Desde el padre podemos hacer focus    // al botón que tenemos en el h
ijo    ref.current?.focus?.()
  }, [ref.current])
  return <Button ref={ref}>My button</Button>}
```

En este ejemplo, recuperamos la referencia del botón (elemento HTML `<button>`) y la recupera el componente padre (`Parent`), para poder hacer focus en él gracias al uso de `forwardRef` en el componente hijo (`Button`).

Para la gran mayoría de componentes esto no es necesario pero puede ser útil para sistemas de diseño o componentes de terceros reutilizables.

¿Cómo puedo validar el tipo de mis props?

React proporciona una forma de validar el tipo de las props de un componente en tiempo de ejecución y en modo desarrollo. Esto es útil para asegurarnos de que los componentes se están utilizando correctamente.

El paquete se llama `prop-types` y se puede instalar con `npm install prop-types`.

```
import PropTypes from 'prop-types'function App(props) {
  return <h1>{props.title}</h1>}
App.propTypes = {
  title: PropTypes.string.isRequired,}
```

En este ejemplo, estamos validando que la prop `title` sea de tipo `string` y que sea obligatoria.

Existen una colección de *PropTypes* ya definidas para ayudarte a comprobar los tipos de las props más comunes:

```
PropTypes.number // númeroPropTypes.string // stringPropTypes.array // arrayPropTypes.object // objetoPropTypes.bool // un booleanoPropTypes.func // funciónPropTypes.node // cualquier cosa renderizable en React, como un número, string, elemento, array, etc.PropTypes.element // un elemento ReactPropTypes.symbol // un Symbol de JavaScriptPropTypes.any // cualquier tipo de dato
```

A todas estas se le puede añadir la propiedad `isRequired` para indicar que es obligatoria.

> Otra opción es usar TypeScript, un lenguaje de programación que compila a JavaScript y que ofrece validación de tipos de forma estática. Ten en cuenta que mientras que TypeScript comprueba los tipos en tiempo de compilación, las PropTypes lo hacen en tiempo de ejecución.

¿Cómo puedo validar las propiedades de un objeto con PropTypes?

Para validar las propiedades de un objeto que se pasa como prop, podemos usar la propiedad `shape` de `PropTypes`:

```
import PropTypes from 'prop-types'function App({ title }) {
  const { text, color } = title
  return <h1 style={{ color }}>{text}</h1>}
App.propTypes = {
  title: PropTypes.shape({
    text: PropTypes.string.isRequired,    color: PropTypes.string.isRequired,
}),}
```

¿Cómo puedo validar las propiedades de un array con PropTypes?

Para validar las propiedades de un array que se pasa como prop, podemos usar la propiedad `arrayOf` de `PropTypes`:

```
import PropTypes from 'prop-types'function App({ items }) {
  return (
    <ul>      {items.map(item => (
        <li key={item.text}>{item.text}</li>      ))}    </ul>  )
}
App.propTypes = {
  items: PropTypes.arrayOf(
    PropTypes.shape({
      text: PropTypes.string.isRequired,    })
  ).isRequired,}
```

En este caso estamos validando que `items` sea un array y que cada uno de sus elementos sea un objeto con la propiedad `text` de tipo `string`. Además, la prop es obligatoria.

¿Cómo puedo inyectar HTML directamente en un componente de React?

Una de las razones por las que se creó React es para evitar los ataques XSS (*Cross-Site Scripting*), impidiendo que un usuario pueda inyectar código HTML en la página.

Por ello, React al intentar evaluar un string que contiene HTML lo escapa automáticamente. Por ejemplo, si intentamos renderizar el siguiente string:

```
const html = '<h1>My title</h1>'function App() {
  return <div>{html}</div>}
```

Veremos que en lugar de renderizar el HTML, lo escapa:

```
<div>&lt;h1&gt;My title&lt;/h1&gt;</div>
```

Sin embargo, hay ocasiones en las que es necesario inyectar HTML directamente en un componente. Ya sea por traducciones que tenemos, porque viene el HTML desde el servidor y ya viene saneado, o por un componente de terceros.

Para ello, podemos usar la propiedad `dangerouslySetInnerHTML`:

```
const html = '<h1>My title</h1>'function App() {
  return <div dangerouslySetInnerHTML={{ __html: html }} />}
```

Ahora sí veremos el HTML renderizado:

```
<div><h1>My title</h1></div>
```

Como ves, **el nombre ya nos indica que es una propiedad peligrosa y que debemos usarla con cuidado.** Intenta evitarla siempre que puedas y sólo recurre a ella cuando realmente no tengas otra opción.

¿Por qué puede ser mala idea pasar siempre todas las props de un objeto a un componente?

Digamos que tenemos un componente `App` que recibe un objeto `props` con todas las props que necesita:

```
function App(props) {
  return <h1>{props.title}</h1>}
```

Y que tenemos otro componente `Layout` que recibe un objeto `props` con todas las props que necesita:

```
function Layout(props) {
  return (
    <div>     <App {...props} />    </div>  )
}
```

En este caso, `Layout` está pasando todas las props que recibe a `App`. Esto puede ser una mala idea por varias razones:

- Si `Layout` recibe una prop que no necesita, la pasará a `App` y éste puede que no la use. Esto puede ser confuso para el que lea el código.

¿Cuál es el propósito del atributo “key” en React y por qué es importante usarlo correctamente al renderizar listas de elementos?

El propósito del atributo “key” en React es proporcionar una identificación única a cada elemento en una lista renderizada dinámicamente. Esto permite a React identificar qué elementos han cambiado, añadido o eliminado de la lista cuando se realiza una actualización.

Cuando se renderiza una lista en React sin el atributo “key”, React puede tener dificultades para identificar correctamente los cambios en la lista, lo que puede resultar en un comportamiento inesperado, como la re-renderización innecesaria de elementos o la pérdida de estado de los componentes.

Por lo tanto, es importante utilizar el atributo “key” de manera correcta y única para cada elemento de la lista, preferiblemente utilizando identificadores únicos de cada elemento en lugar de índices de array, para garantizar un rendimiento óptimo y un comportamiento predecible en la aplicación.

Ejemplo de cómo utilizar el atributo “key” en React:

```
import React from 'react'const ListaItems = ({ items }) => {
  return (
    <ul>      {items.map(item => (
        <li key={item.id}>{item.nombre}</li>      ))}    </ul>  )
}
export default ListaItems
```

Experto

¿Es React una biblioteca o un framework? ¿Por qué?

Existe una fina línea hoy en día entre qué es una biblioteca o un framework. Oficialmente, React se autodenomina como biblioteca. Esto es porque para poder crear una aplicación completa, necesitas usar otras bibliotecas.

Por ejemplo, *React* no ofrece un sistema de enrutado de aplicaciones oficial. Por ello, hay que usar una biblioteca como React Router o usar un *framework* como Next.js que ya incluye un sistema de enrutado.

Tampoco puedes usar React para añadir las cabeceras que van en el `<head>` en tu aplicación, y también necesitarás otra biblioteca o framework para solucionar esto.

Otra diferencia es que React no está opinionado sobre qué empaquetador de aplicaciones usar. En cambio `Angular` en su propio tutorial ya te indica que

debes usar `@angular/cli` para crear una aplicación, en cambio React siempre te deja la libertad de elegir qué empaquetador usar y ofrece diferentes opciones.

Aún así, existe gente que considera a React como un framework. Aunque no hay una definición oficial de qué es un framework, la mayoría de la gente considera que un framework es una biblioteca que incluye otras bibliotecas para crear una aplicación completa de forma opinionada y casi sin configuración.

Por ejemplo, **Next.js se podría considerar un framework de React** porque incluye React, un sistema de enrutado, un sistema de renderizado del lado del servidor, etc.

¿Para qué sirve el hook `useImperativeHandle`?

Nos permite definir qué propiedades y métodos queremos que sean accesibles desde el componente padre.

En el siguiente ejemplo vamos a crear un componente `TextInput` que tiene un método `focus` que cambia el foco al elemento `<input>`.

```
import { useRef, useImperativeHandle } from 'react'function TextInput(prop
s, ref) {
  const inputEl = useRef(null)
  useImperativeHandle(ref, () => ({
    focus: () => {
      inputEl.current.focus()
    },  }))
  return <input ref={inputEl} type='text' />}
```

Creamos una referencia `inputEl` con `useRef` y la pasamos al elemento `<input>` como prop `ref`. Cuando el componente se monta, la referencia `inputEl` apunta al elemento `<input>` del DOM.

Para acceder al elemento del DOM, usamos la propiedad `current` de la referencia.

Para que el componente padre pueda acceder al método `focus`, usamos el hook `useImperativeHandle`. Este hook recibe dos parámetros: una referencia y una función que devuelve un objeto con las propiedades y métodos que queremos que sean accesibles desde el componente padre.

¿Para qué sirve el método `cloneElement` de React?

Te permite clonar un elemento React y añadirle o modificar las props que recibe.

```
import { cloneElement } from 'react'const Hello = ({ name }) => <h1>Hello {name}</h1>const App = () => {
  const element = <Hello name='EmiDev' />  return (
    <div>      {cloneElement(element, { name: 'TMChein' })}      {cloneElement(element, { name: 'Madeval' })}      {cloneElement(element, { name: 'Gorusuke' })}    </div>  )
}
```

En este ejemplo, clonamos `element` que tenía la prop `EmiDev` y le pasamos una prop `name` diferente cada vez. Esto renderizará tres veces el componente `Hello` con los nombres `TMChein`, `Madeval` y `Gorusuke`. Sin rastro de la prop original.

Puede ser útil para modificar un elemento que ya nos viene de un componente padre y del que no tenemos posibilidad de re-crear con el componente.

¿Qué son los portales en React?

Los portales nos permiten renderizar un componente en un nodo del DOM que no es hijo del componente que lo renderiza.

Es perfecto para ciertos casos de uso como, por ejemplo, modales:

```
import { createPortal } from 'react-dom'function Modal() {
  return createPortal(
    <div className='modal'>      <h1>Modal</h1>    </div>,    document.getElementById('modal')
  )
}
```

`createPortal` acepta dos parámetros:

- El primer parámetro es el componente que queremos renderizar
- El segundo parámetro es el nodo del DOM donde queremos renderizar el componente

En este caso el modal se renderiza en el nodo `#modal` del DOM.

¿Por qué `StrictMode` renderiza dos veces la aplicación?

Cuando el modo `StrictMode` está activado, React monta los componentes dos veces (el estado y el DOM se preserva). Esto ayuda a encontrar efectos que necesitan una limpieza o expone problemas con *race conditions*.

¿Qué problemas crees que pueden aparecer en una aplicación al querer visualizar listas de miles/millones de datos?

- **Tiempo de respuesta del servidor:** Hacer peticiones de millones de datos no es, en general, una buena estrategia. Incluso en el mejor de los casos, en el que el servidor solo debe devolver los datos sin tratarlos, hay un coste asociado al *parseo* y *envío* de los mismos a través de la red. Llamadas con un tamaño desmesurado pueden incurrir en interfaces lentas, e incluso en *timeouts* en la respuesta.
- **Problemas de rendimiento:** Aunque es cierto que **React** se basa en un modelo *declarativo* en el cual no debemos tener una exhaustivo control o gestión de cómo se *renderiza*, no hay que olvidar que malas decisiones técnicas pueden conllevar aplicaciones totalmente inestables incluso con las mejores tecnologías. No es viable *renderizar* un *DOM* con millones de elementos, el *navegador* no podrá gestionarlo y, tarde o temprano, la aplicación no será usable.

Como developers, nuestra misión es encontrar el equilibrio entre rendimiento y experiencia, intentando priorizar siempre cómo el usuario sentirá la aplicación. No hay ningún caso lo suficientemente justificado para *renderizar* en pantalla miles de datos.

El espacio de visualización es limitado (*viewport*), al igual que deberían serlo los datos que añadimos al DOM.

¿Cómo puedes abortar una petición fetch con `useEffect` en React?

Si quieres evitar que exista una *race condition* entre una petición asíncrona y que el componente se desmonte, puedes usar la API de `AbortController` para abortar la petición cuando lo necesites:

```
import { useEffect, useState } from 'react'function Movies() {
  const [movies, setMovies] = useState([])
  useEffect(() => {
```

```
    // creamos un controlador para abortar la petición    const abortControlle
r = new AbortController()
    // pasamos el signal al fetch para que sepa que debe abortar    fetchMovi
es({ signal: abortController.signal })
      .then(() => {
        setMovies(data.results)
      })
      .catch(error => {
        if (error.name === 'AbortError') {
          console.log('fetch aborted')
        }
      })
    return () => {
      // al desmontar el componente, abortamos la petición      // sólo funcion
ará si la petición sigue en curso      abortController.abort()
    }
  })
  // ...}
// Debemos pasarle el parámetro signal al `fetch`// para que enlace la petici
ón con el controladorconst fetchMovies = ({ signal }) => {
  return fetch('https://api.themoviedb.org/3/movie/popular', {
    signal, // <——— pasamos el signal  }).then(response => response.json())
}
```

De esta forma evitamos que se produzca un error por parte de React de intentar actualizar el estado de un componente que ya no existe, además de evitar que se produzcan llamadas innecesarias al servidor.

¿Qué solución/es implementarías para evitar problemas de rendimiento al trabajar con listas de miles/millones de datos?

Pagination

En lugar de recibir la lista en una sola llamada a la API (lo cual sería negativo tanto para el rendimiento como para el propio servidor y tiempo de respuesta de la API), podríamos implementar un sistema de paginación en el cual la API recibirá un *offset* o *rango* de datos deseados. En el FE nuestra responsabilidad es mostrar unos controles adecuados (interfaz de paginación) y gestionar las llamadas a petición de cambio de página para siempre limitar la cantidad de

DOM renderizado evitando así una sobrecarga del *DOM* y, por lo tanto, problemas de rendimiento.

Virtualization

Existe una técnica llamada *Virtualización* que gestiona cuántos elementos de una lista mantenemos ***vivos*** en el *DOM*. El concepto se basa en solo montar los elementos que estén dentro del *viewport* más un *buffer* determinado (para evitar falta de datos al hacer scroll) y, en cambio, desmontar del *DOM* todos aquellos elementos que estén fuera de la vista del usuario. De este modo podremos obtener lo mejor de los dos mundos, una experiencia integrada y un DOM liviano que evitará posibles errores de rendimiento. Con esta solución también podremos aprovechar que contamos con los datos en memoria para realizar búsquedas/filtrados sin necesidad de más llamadas al servidor.

Puedes consultar esta librería para aplicar Virtualización con React: React Virtualized.

Hay que tener en cuenta que cada caso de uso puede encontrar beneficios y/o perjuicios en ambos métodos, dependiendo de factores como capacidad de respuesta de la API, cantidad de datos, necesidad de filtros complejos, etc. Por ello es importante analizar cada caso con criterio.

¿Qué es el hook `useDebugValue`?

Nos permite mostrar un valor personalizado en la pestaña de *React DevTools* que nos permitirá depurar nuestro código.

```
import { useDebugValue } from 'react'function useCustomHook() {
  const value = 'custom value'  useDebugValue(value)
  return value
}
```

En este ejemplo, el valor personalizado que se muestra en la pestaña de *React DevTools* es `custom value`.

Aunque es útil para depurar, no se recomienda usar este hook en producción.

¿Qué es el `Profiler` en React?

El `Profiler` es un componente que nos permite medir el tiempo que tarda en renderizarse un componente y sus hijos.

```
import { Profiler } from 'react'function App() {
  return (
    <Profiler      id='App'      onRender={(id, phase, actualDuration) ⇒ {
       console.log({ id, phase, actualDuration })
     }}    >      <Component />    </Profiler>  )
}
```

El componente `Profiler` recibe dos parámetros:

- `id`: es un identificador único para el componente
- `onRender`: es una función que se ejecuta cada vez que el componente se renderiza

Esta información es muy útil para detectar componentes que toman mucho tiempo en renderizarse y optimizarlos.

¿Cómo puedes acceder al evento nativo del navegador en React?

React no expone el evento nativo del navegador. En su lugar, React crea un objeto sintético que se basa en el evento nativo del navegador llamado `SyntheticEvent`. Para acceder al evento nativo del navegador, debemos usar el atributo `nativeEvent`:

```
function Button({ onClick }) {
  return <button onClick={e ⇒ onClick(e.nativeEvent)}>Haz clic aquí</butto
n>}
```

¿Cómo puedes registrar un evento en la fase de captura en React?

En React, los eventos se registran en la fase de burbuja por defecto. Para registrar un evento en la fase de captura, debemos añadir `Capture` al nombre del evento:

```
function Button({ onClick }) {
  return <button onClickCapture={onClick}>Haz clic aquí</button>}
```

¿Cómo puedes mejorar el rendimiento del Server Side Rendering en React para evitar que bloquee el hilo principal?

Aunque puedes usar el método `renderToString` para renderizar el HTML en el servidor, este método es síncrono y bloquea el hilo principal. Para evitar que bloquee el hilo principal, debemos usar el método `renderToPipeableStream`:

```
let didError = falseconst stream = renderToPipeableStream(<App />, {
 onShellReady() {
   // El contenido por encima de los límites de Suspense ya están listos    // Si hay un error antes de empezar a hacer stream, mostramos el error adecuado    res.statusCode = didError ? 500 : 200    res.setHeader('Content-type', 'text/html')
   stream.pipe(res)
 },  onShellError(error) {
   // Si algo ha ido mal al renderizar el contenido anterior a los límites de Suspense, lo indicamos.    res.statusCode = 500    res.send(
     '<!doctype html><p>Loading...</p><script src="clientrender.js"></script>'    )
 },  onAllReady() {
   // Si no quieres hacer streaming de los datos, puedes usar    // esto en lugar de onShellReady. Esto se ejecuta cuando    // todo el HTML está listo para ser enviado.    // Perfecto para crawlers o generación de sitios estáticos    // res.statusCode = didError ? 500 : 200    // res.setHeader('Content-type', 'text/html')    // stream.pipe(res)  },  onError(err) {
   didError = true    console.error(err)
 },})
```

¿Qué diferencia hay entre `renderToStaticNodeStream()` y `renderToPipeableStream()`?

`renderToStaticNodeStream()` devuelve un stream de nodos estáticos, esto significa que no añade atributos extras para el DOM que React usa internamente para poder lograr la hidratación del HTML en el cliente. Esto significa que no podrás hacer el HTML interactivo en el cliente, pero puede ser útil para páginas totalmente estáticas.

`renderToPipeableStream()` devuelve un stream de nodos que contienen atributos del DOM extra para que React pueda hidratar el HTML en el cliente. Esto significa

que podrás hacer el HTML interactivo en el cliente pero puede ser más lento que `renderToStaticNodeStream()`.

¿Para qué sirve el hook `useDeferredValue`?

El hook `useDeferredValue` nos permite renderizar un valor con una prioridad baja. Esto es útil para renderizar un valor que no es crítico para la interacción del usuario.

```
function App() {
  const [text, setText] = useState('¡Hola mundo!')
  const deferredText = useDeferredValue(text, { timeoutMs: 2000 })
  return (
    <div className='App'>      {/* Seguimos pasando el texto actual como v
alor del input */}      <input value={text} onChange={handleChange} />
...
      {/* Pero la lista de resultados se podría renderizar más tarde si fuera ne
cesario */}      <MySlowList text={deferredText} />    </div>  )
}
```

¿Para qué sirve el método `renderToReadableStream()`?

Este método es similar a `renderToNodeStream`, pero está pensado para entornos que soporten Web Streams como `Deno`.

Un ejemplo de uso sería el siguiente:

```
const controller = new AbortController()
const { signal } = controller
let didError = falsetry {
  const stream = await renderToReadableStream(
    <html>      <body>Success</body>    </html>,    {
      signal,      onError(error) {
        didError = true        console.error(error)
      },    }
  )
  // Si quieres enviar todo el HTML en vez de hacer streaming, puedes usar
esta línea  // Es útil para crawlers o generación estática:  // await stream.all
Ready  return new Response(stream, {
```

```
    status: didError ? 500 : 200,    headers: { 'Content-Type': 'text/html' },  })
} catch (error) {
  return new Response(
    '<!doctype html><p>Loading...</p><script src="clientrender.js"></scrip
t>',    {
      status: 500,      headers: { 'Content-Type': 'text/html' },    }
  )
}
```

¿Cómo puedo hacer testing de un componente?

Para hacer testing de un componente, puedes usar la función `render` de la librería `@testing-library/react`. Esta función nos permite renderizar un componente y obtener el resultado.

```
import { render } from '@testing-library/react'function Counter() {
  const [count, setCount] = useState(0)
  const increment = () => setCount(count + 1)
  return (
    <div>      <p>Count: {count}</p>      <button onClick={increment}>Incre
ment</button>    </div>  )
}
test('Counter', () => {
  const { getByText } = render(<Counter />)
  expect(getByText('Count: 0')).toBeInTheDocument()
  fireEvent.click(getByText('Increment'))
  expect(getByText('Count: 1')).toBeInTheDocument()
})
```

¿Cómo puedo hacer testing de un hook?

Para hacer testing de un hook, puedes usar la función `renderHook` de la librería `@testing-library/react-hooks`. Esta función nos permite renderizar un hook y obtener el resultado.

```
import { renderHook } from '@testing-library/react-hooks'function useCou
nter() {
  const [count, setCount] = useState(0)
```

```
  const increment = () => setCount(count + 1)
  return { count, increment }
}
test('useCounter', () => {
  const { result } = renderHook(() => useCounter())
  expect(result.current.count).toBe(0)
  act(() => {
    result.current.increment()
  })
  expect(result.current.count).toBe(1)
})
```

¿Qué es Flux?

Flux es un patrón de arquitectura de aplicaciones que se basa en un unidireccional de datos. En este patrón, los datos fluyen en una sola dirección: de las vistas a los stores.

No es específico de React y se puede usar con cualquier librería de vistas. En este patrón, los stores son los encargados de almacenar los datos de la aplicación. Los stores emiten eventos cuando los datos cambian. Las vistas se suscriben a estos eventos para actualizar los datos.

Esta arquitectura fue creada por Facebook para manejar la complejidad de sus aplicaciones. *Redux* se basó en este patrón para crear una biblioteca de gestión de estado global.

Errores Típicos en React

¿Qué quiere decir: Warning: Each child in a list should have a unique key prop?

Es un error bastante común en React y que puede parecernos un poco extraño si estamos empezando a aprender esta tecnología. Por suerte, es bastante sencillo de solucionar.

Básicamente, este mensaje aparece en la consola cuando estamos renderizando un listado dentro de nuestro componente, pero no le estamos indicando la propiedad “key”. React usa esta propiedad para **determinar qué elemento hijo dentro de un listado ha sufrido cambios,** por lo que funciona como una especie de identificativo.

De esta manera, React utiliza esta información para **identificar las diferencias existentes con respecto al DOM** y optimizar la renderización del listado, determinando qué elementos necesitan volverse a calcular. Esto habitualmente pasa cuando agregamos, eliminamos o cambiamos el orden de los items en una lista.

Recomendamos revisar las siguientes secciones:

- ¿Qué es el renderizado de listas en React?
- ¿Por qué puede ser mala práctica usar el ´index´ como key en un listado de React?

React Hook useXXX is called conditionally. React Hooks must be called in the exact same order in every component render

Una de las reglas de los hooks de React es que deben llamarse en el mismo orden en cada renderizado. React lo necesita para saber en qué orden se llaman los hooks y así mantener el estado de los mismos internamente. Por ello, los hooks no pueden usarse dentro de una condición `if`, ni un loop, ni tampoco dentro de una función anónima. Siempre deben estar en el nivel superior de la función.

Por eso el siguiente código es incorrecto:

```
// ❌ código incorrecto por saltar las reglas de los hooksfunction Counter()
{
  const [count, setCount] = useState(0)
  // de forma condicional, creamos un estado con el hook useState  // lo qu
e rompe la regla de los hooks  if (count > 0) {
    const [name, setName] = useState('Emi')
  }
  return (
    <div>     {count} {name}    </div>  )
}
```

También el siguiente código sería incorrecto, aunque no lo parezca, ya que estamos usando el segundo `useState` de forma condicional (pese a no estar dentro de un `if`) ya que se ejecutará sólo cuando `count` sea diferente a `0`:

```
// ❌ código incorrecto por saltar las reglas de los hooksfunction Counter()
{
  const [count, setCount] = useState(0)
  // si count es 0, no se ejecuta el siguiente hook useState  // ya que salimo
s de la ejecución aquí  if (count === 0) return null  const [name, setName]
= useState('Emi')
  return (
    <div>     {count} {name}    </div>  )
}
```

Ten en cuenta que si ignoras este error, es posible que tus componentes no se comporten de forma correcta y tengas comportamientos no esperados en el funcionamiento de tus componentes.

Para arreglar este error, como hemos comentado antes, debes asegurarte de que los hooks se llaman en el mismo orden en cada renderizado. El último ejemplo quedaría así:

```
function Counter() {
  const [count, setCount] = useState(0)
  // movemos el hook useState antes del if  const [name, setName] = useSt
ate('Emi')
  if (count === 0) return null  return (
    <div>     {count} {name}    </div>  )
}
```

Recomendamos revisar las siguientes secciones:

- ¿Cuáles son las reglas de los hooks en React?

Can’t perform a React state update on an unmounted component

Este error se produce cuando intentamos actualizar el estado de un componente que ya no está montado. Esto puede ocurrir cuando el componente se desmonta antes de que se complete una petición asíncrona, por ejemplo:

```
function Movies() {
  const [movies, setMovies] = useState([])
  useEffect(() => {
    fetchMovies().then(() => {
      setMovies(data.results)
    })
  })
  if (!movies.length) return null  return (
    <section>     {movies.map(movie => (
       <article key={movie.id}>        <h2>{movie.title}</h2>        <p>{movi
e.overview}</p>       </article>     ))}   </section>  )
}
```

Parece un código inofensivo, pero imagina que usamos este componente en una página. Si el usuario navega a otra página antes de que se complete la petición, el componente se desmontará y React lanzará el error, ya que intentará ejecutar el `setMovies` en un componente (Movies) que ya no está montado.

Para evitar este error, podemos usar una variable booleana con `useRef` que nos indique si el componente está montado o no. De esta manera, podemos evitar que se ejecute el `setMovies` si el componente no está montado:

```
function Movies() {
  const [movies, setMovies] = useState([])
  const mounted = useRef(false)
  useEffect(() => {
    mounted.current = true    fetchMovies().then(() => {
      if (mounted.current) {
        setMovies(data.results)
      }
    })
    return () => (mounted.current = false)
  })
  // ...}
```

Esto soluciona el problema pero **no evita que se haga la petición aunque el componente ya no esté montado**. Para cancelar la petición y así ahorrar

transferencia de datos, podemos abortar la petición usando la API `AbortController`:

```
function Movies() {
  const [movies, setMovies] = useState([])
  useEffect(() => {
    // creamos un controlador para abortar la petición    const abortControlle
r = new AbortController()
    // pasamos el signal al fetch para que sepa que debe abortar    fetchMovi
es({ signal: abortController.signal })
      .then(() => {
        setMovies(data.results)
      })
      .catch(error => {
        if (error.name === 'AbortError') {
          console.log('fetch aborted')
        }
      })
    return () => {
      // al desmontar el componente, abortamos la petición      // sólo funcion
ará si la petición sigue en curso      abortController.abort()
    }
  })
  // ...}
// Debemos pasarle el parámetro signal al `fetch`// para que enlace la petici
ón con el controladorconst fetchMovies = ({ signal }) => {
  return fetch('https://api.themoviedb.org/3/movie/popular', {
    signal, // <——— pasamos el signal  }).then(response => response.json())
}
```

Sólo ten en cuenta la compatibilidad de `AbortController` en los navegadores. En caniuse puedes ver que no está soportado en Internet Explorer y versiones anteriores de Chrome 66, Safari 12.1 y Edge 16.

Too many re-renders. React limits the number of renders to prevent an infinite loop

Este error indica que algo dentro de nuestro componente está generando muchos pintados que pueden desembocar en un *loop* (bucle) infinito. Algunas de las razones por las que puede aparecer este error son las siguientes:

1. **Llamar a una función que actualiza el estado en el renderizado del componente.**

```
function Counter() {
  const [count, setCount] = useState(0)
  // ❌ código incorrecto  // no debemos actualizar el estado de manera dir
ecta  setCount(count + 1)
  return <div>{count}</div>}
```

Lo que sucede en este ejemplo, es que al *renderizarse* el componente, se llama a la función `setCount` para actualizar el estado. Una vez el estado es actualizado, se genera nuevamente un *render* del componente y se repite todo el proceso infinitas veces.

Una posible solución sería:

```
function Counter() {
  // ✅ código correcto  // se pasa el valor inicial deseado en el `useState`  c
onst [count, setCount] = useState(1)
  return <div>{count}</div>}
```

Llamar directamente a una función en un controlador de eventos.

```
function Counter() {
  const [count, setCount] = useState(0)
  // ❌ código incorrecto  //se ejecuta directamente la función `setCount` y
provoca un renderizado infinito  return (
    <div>      <p>Contador: {count}</p>      <button onClick={setCount(coun
t + 1)}>Incrementar</button>    </div>  )
}
```

En este código, se está ejecutando la función `setCount` que actualiza el estado en cada renderizado del componente, lo que provoca renderizaciones infinitas.

La manera correcta sería la siguiente:

```
function Counter() {
  const [count, setCount] = useState(0)
  // ✅ código correcto  // se pasa un callback al evento `onClick`  // esto ev
ita que la función se ejecute en el renderizado  return (
```

```
    <div>      <p>Contador: {count}</p>      <button onClick={() => setCount
(count + 1)}>Incrementar</button>    </div>  )
}
```

Usar incorrectamente el Hook de `useEffect`.

Al ver este ejemplo:

```
function Counter() {
  const [count, setCount] = useState(0)
  // ❌ código incorrecto  useEffect(() => {
    setCounter(counter + 1)
  }) // 👈 no colocar el array de dependencias  return <div>{count}</div>}
```

Lo que ocurre, es que al no colocar un array de dependencias en el hook de `useEffect`, estamos provocando que el código que se encuentre dentro se ejecute en cada renderizado del componente. Al llamar al `setCounter` y actualizar el estado, obtenemos nuevamente renderizaciones infinitas.

Para solucionarlo, podemos hacer lo siguiente:

```
function Counter() {
  const [count, setCount] = useState(0)
  // ✅ código correcto  // estamos indicando que sólo queremos que el cód
igo se ejecute una vez  useEffect(() => {
    setCounter(counter + 1)
  }, []) //colocamos un array de dependencias vacío.  return <div>{count}</
div>}
```

Estas son solo algunas de las posibles causas que podemos encontrar cuando nos topamos con este mensaje de error en el código. Si quieres complementar esta información, te recomendamos revisar las siguientes secciones:

- ¿Qué es el estado en React?
- ¿Qué son los hooks?
- ¿Qué hace el hook useState?
- ¿Qué hace el hook useEffect?
- ¿Cuáles son las reglas de los hooks en React?

¿Qué diferencia existe entre Shadow DOM y Virtual DOM?

El **Shadow DOM** es una API del navegador que nos permite crear un árbol de nodos DOM independiente dentro de un elemento del DOM. Esto nos permite crear componentes que no interfieran con el resto de la aplicación. Se usa especialmente con Web Components.

El **Virtual DOM** es una representación del DOM en memoria. Esta representación se crea cada vez que se produce un cambio en el DOM. Esto nos permite comparar el DOM actual con el DOM anterior y así determinar qué cambios se deben realizar en el DOM real. Lo usa React y otras bibliotecas para hacer el mínimo número de cambios en el DOM real.

¿Qué es el Binding?

En React, el **Binding** se refiere a la forma en que se relaciona y sincroniza el **estado** *(state)* de un componente con su **vista** *(render)*. El estado de un componente es un objeto que contiene información que puede ser utilizada para determinar cómo se debe mostrar el componente. Existen **dos** tipos de binding en React: **One-Way Binding** y **Two-Way Binding**.

One-Way Binding *(Enlace unidireccional)*:

En React se refiere a la capacidad de un componente para actualizar su **estado** *(state)* y su **vista** *(render)* de manera automática cuando cambia el estado, pero no permitiendo que la vista actualice el estado. En otras palabras, el **one-way binding** significa que el flujo de datos es unidireccional, desde el estado hacia la vista, y no al revés.

Por ejemplo:

```
import React, { useState } from 'react'function OneWayBindingExample() {
  const [name, setName] = useState('Emi')
  return (
    <div>     <p>Hello, {name}</p>     <input     type='text'     placehold
er='Enter your name'     onChange={e ⇒ setName(e.target.value)}     />
</div>  )
}
export default OneWayBindingExample
```

*En este ejemplo, el componente tiene un estado inicial llamado **name** con el valor **Emi**. La función **setName** se utiliza para actualizar el estado **name***

*cuando se produce un evento **onChange** en el input. Sin embargo, la **vista** (la linea que muestra **Hello, {name}**) no tiene la capacidad de actualizar el estado **name**.*

Two-Way Binding *(Enlace bidireccional)*:

Se refiere a la capacidad de un componente para actualizar su estado y su vista de manera automática tanto cuando cambia el estado como cuando se produce un evento en la vista. En otras palabras, el **Two-Way Binding** significa que el flujo de datos es bidireccional, desde el estado hacia la vista y desde la vista hacia el estado. Para lograr esto se utilizan en conjunto con los eventos, como **onChange**, para capturar la información de los inputs y actualizar el estado, *React no proporciona un mecanismo nativo para two-way binding, pero se puede lograr utilizando librerías como react-forms o formik.*

Por ejemplo:

```
import React, { useState } from 'react'function TwoWayBindingExample() {
  const [name, setName] = useState('Emi')
  return (
    <div>    <p>Hello, {name}</p>    <input    type='text'    placehold
er='Enter your name'    value={name}    onChange={e ⇒ setName(e.ta
rget.value)}    />    </div>  )
}
export default TwoWayBindingExample
```

*En este ejemplo, el componente tiene un estado inicial llamado **name** con el valor **Emi**. La función **setName** se utiliza para actualizar el estado **name** cuando se produce un evento **onChange** en el input, y se puede ver reflejado en el valor del input. Sin embargo, en este caso se está utilizando el atributo **value** para que el valor del input sea actualizado con el valor del estado, es decir, se está actualizando tanto el estado como el input.*

Por si no quedó claro:

En términos sencillos, el **Binding** en React puede compararse con una cafetera y una taza de café. **El estado** del componente sería la *cafetera*, y **la vista** del componente sería *la taza de café*.

En el caso del **One-Way Binding**, la cafetera solo puede verter café en una dirección, hacia la taza de café. Esto significa que la cafetera puede llenar automáticamente la taza de café con café fresco, pero la taza de café no

puede devolver automáticamente el café a la cafetera. De esta manera, **el estado** del componente *(la cafetera)* puede actualizar automáticamente **la vista** *(la taza de café)* cuando cambia, pero la **vista** no puede actualizar automáticamente el **estado**.

En el caso del **Two-Way Binding**, la cafetera puede verter y recibir café en ambas direcciones, hacia y desde la taza de café (no sé por qué alguien necesitaría hacer algo así). Esto significa que la cafetera puede llenar y vaciar automáticamente la taza de café con café fresco. De esta manera, tanto **el estado** del componente como **la vista** pueden actualizarse automáticamente entre sí.

Sí quieres saber más comparto el siguiente enlace:

How To Bind Any Component to Data in React: One-Way Binding