React Hooks: Manual Desde Cero

De Developero

Acerca de este Ebook

Este es un Ebook creado por Juan Correa, fundador de Developero, espacio para compartir conocimientos sobre desarrollo de software.

En venta en Amazon y Leanpub.

Comparte el conocimiento.

Otros Recursos de Developero

Aprende más temas de desarrollo en Javascript consultando los diferentes cursos.

Curso: Unit, Integration y E2E Testing en Fullstack Javascript.

Este es uno de los cursos MÁS COMPLETOS de testing automatizado con Javascript.

Vas a aprender todo lo necesario para crear pruebas automatizadas tanto en el Frontend como en el Backend (React JS y Node JS con Express).

En las casi 14 horas de contenido, vamos a revisar paso a paso y desde cero lo siguiente:

- Fundamentos del Testing automatizado.
- Herramientas de análisis estático de código.
- Introducción a Jest y Mocks.
- Pruebas unitarias en Frontend y Backend.
- Pruebas de integración en Frontend y Backend.
- Pruebas E2E de una app completa full stack Javascript.

Da click aquí para obtener un cupón de descuento.

Curso: Test Driven Development en React JS.

Mejora tus habilidades en React usando TDD - Jest, React Testing Library, MSW, React Router, Hooks y más.

Algunos de los temas que se ven en este curso son:

- Comprender los principios de TDD.
- Aplicar REALMENTE TDD en React JS moderno (incluyendo hooks) en tu día a día.

- Aplicar fuertemente buenas prácticas - Código limpio y refactors en React JS.
- Unit e Integration testing con React Testing Library.

Da click aquí para obtener un cupón de descuento.

Curso: Guía definitiva: Aprende los 9 Patrones Avanzados en ReactJS

En este curso original te muestro cómo crear componentes en React JS realmente reutilizables, escalables y fáciles de mantener paso a paso.

Esto es posible implementando los Patrones Avanzados en React JS así como los principios SOLID que los fundamentan.

- En cada patrón partiremos creando componentes normales según un caso de uso, después veremos las desventajas de ese componente y cómo el patrón que estemos revisando nos va a ayudar a resolver esas desventajas.

- Vas a tener acceso al código fuente de los ejemplos de cada patrón con la posibilidad de poder ejecutarlo y editarlo para que consolides mejor tu aprendizaje.

Da click aquí para obtener un cupón de descuento.

Curso: Guía completa React Hooks: Manual Desde Cero con Ejemplos

Conoce los por qué, cómo y en qué casos aplicar cada hook. Vamos a revisar en cada hook cuáles son los escenarios donde conviene aplicarlos y cómo hacerlo con buenas prácticas.

Comparativas entre clases y hooks. Para facilitar el aprendizaje vamos a ver ejemplos en componentes de tipo clase y después haremos la versión equivalente en componente funcionales con hooks.

Acceso a ejemplos con código fuente. Leer sin aplicar no sirve de mucho. Es por eso que tienes acceso al código fuente de los ejemplos para que puedas ejecutarlo, editarlo y ver los resultados.

Da click aquí para obtener un cupón de descuento.

Introducción a los React Hooks

Los React Hooks fueron incorporados a partir de la versión 16.8 de React y nos permiten utilizar las características de esta biblioteca en componentes funcionales.

Para dar solo un ejemplo: a partir de ahora puedes declarar variables de estado en componentes funcionales, algo que antes era posible sólo en componentes de tipo clase.

¿Qué aspecto tiene el uso de los hooks?

Un ejemplo a continuación.

Ejemplo de useState

```
1 import React, { useState } from "react";
2
3 const Count = () => {
4   const [count, setCount] = React.useState(0);
5
6   const handleClick = () => {
7     setCount(count + 1);
8   };
9
10   return (
11     <>
12       <p>Clicks: {count}</p>
13       <button onClick={handleClick}>Clickeame</button>
14     </>
15   );
16 };
```

Esto es sólo una muestra introductoria para que veas como luce un hook. No te preocupes si este ejemplo no hace sentido. Ya veremos en capítulos posteriores cada hook paso a paso :).

Conforme avances en estas páginas te darás cuenta del **PODER** de los hooks y los nuevos caminos que se abren para ti al momento en que diseñas tus componentes.

¿Los React Hooks van a reemplazar los componentes de tipo clase?

No realmente. Un buen acierto de React es que los componentes de tipo clase van a seguir siendo soportados, por lo que puedes tener la tranquilidad de que tu código fuente que las contenga va a seguir funcionando.

Lo cierto es que a partir de ahora puedes elegir crear componentes de tipo clase o tipo función según las necesidades de la aplicación y de tus criterios como desarrollador.

Al momento de escribir este manual, hay un caso de uso en el que los componentes de tipo clase pueden hacer algo que los hooks no pueden: el manejo de límite de errores (Error Boundaries).

Si no estás familiarizado con esto, es simplemente la capacidad de un componente de manejar un error de ejecución y te provee acceso a actuar en consecuencia, como mostrar un mensaje y canalizar el error en un servicio de informes como Sentry. Todo esto sin tirar la aplicación.

Los hooks nos proveen de una manera más limpia para manejar la misma lógica que nos permiten manejar los ciclos de vida de una clase como veremos en los ejemplos de cada hook.

Ya que te beneficia tener código simple en vez de complejo, probablemente termines eligiendo los hooks la mayoría de las veces.

Aunque ojo: aún con hooks puedes terminar con código complejo si sigues malas prácticas. Este manual te va a guiar en las buenas prácticas para que saques el máximo provecho a los hooks.

¿Qué necesidades cubren los React Hooks?

Los hooks vienen para resolver los siguientes problemas:

Reusar lógica entre componentes: La manera de reusar lógica antes de los hooks era por medio de patrones o técnicas como los render

props y los componentes de orden superior (HOC por sus siglas en inglés).

El problema de esto es que la jerarquía de componentes tiende a terminar siendo un árbol complejo de envolvedores (wrapper hell).

Veremos en un apartado posterior cómo puedes crear tus propios hooks, de modo que ahora podrás reusar lógica entre componentes de una manera más limpia y sin necesidad de modificar la jerarquía de los mismos.

Componentes complejos: En este contexto, un componente complejo es aquel que para funcionar termina ejecutando varias sentencias de código que pueden estar o no relacionadas en los mismos métodos de ciclos de vida de un componente.

Por ejemplo: puede que cuando se monte el componente, en el método componentDidMount, consumas una api y en ese mismo método inicialices una o varias suscripciones de eventos del navegador (o de algo diferente a la api).

Lo anterior ocasiona que tengamos una lógica diferente mezclada en el mismo lugar.

El inconveniente de esto es que el código se hace difícil de leer y por lo tanto da pie a los errores humanos.

Ahora pasemos a la acción y veamos cada uno de los hooks, cómo, por qué y en qué contextos usarlos.

Te veo en el siguiente capítulo :)

Hook de estado: useState

El hook useState te permite poder usar variables de estado dentro de un componente funcional.

Este es el hook que probablemente más usarás por lo que comprenderlo bien va a ser muy importante.

Comparación de componente tipo clase y funcional al usar variables de estado

A continuación vamos a hacer un componente de tipo clase que sirva para contar los clicks que hace el usuario a un botón.

Uso de estado en componente tipo clase

```
class Count extends React.Component {
  state = {
    count: 0
  };

  handleClick = () => this.setState({ count: this.state.count + 1 });

  render() {
    const { count } = this.state;
    return (
      <>
        <p>Clicks: {count}</p>
        <button onClick={this.handleClick}>Clickeame</button>
      </>
    );
  }
}
```

No usamos el constructor para iniciar el valor del state.

Lo definimos por fuera del constructor basados en la sintaxis alternativa, es decir, declarar el estado como una propiedad de la clase desde fuera del constructor.

Ahora vamos a hacer el ejemplo equivalente en componente funcional utilizando el hook useState.

Uso de estado en componente funcional

```
const Count = () => {
 const [count, setCount] = useState(0);

 const handleClick = () => {
  setCount(count + 1);
 };

 return (
  <>
   <p>Clicks: {count}</p>
   <button onClick={handleClick}>Clickeame</button>
  </>
 );
};
```

¡Mucho mejor! El componente es más simple, más limpio y con el mismo poder del manejo de estado de React :D.

Codigo fuente para ejecutar y editar aqui.

El hook useState nos retorna siempre un array con dos elementos.

Destructuracion de array en el hook useState

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

La sintaxis que vemos con los corchetes no es de React sino de ES6 y es llamada desestructuración.

En la posición cero del array nos retorna la variable de estado y la posición uno nos retorna una función que nos sirve para actualizar la variable de estado.

Por lo tanto, lo siguiente es igualmente válido para Javascript.

Declaración de variables de estado sin destructurar

```
1 const MyComponent = () => {
2   const state = React.useState(null);
3   const myState = state[0];
4   const setMyState = state[1];
5 };
```

Como puedes ver, es más limpio usar la desestructuración por lo que siempre verás ejemplos de ese modo.

Y al ser destruturado, podemos nombrar las dos variables de la manera que convenga mejor a nuestros intereses.

Lo recomendado es que sean nombres con significado, que indiquen cuál es su propósito.

Ya que la segunda variable es una función que actualiza el estado, en todos los ejemplos vas a encontrar que inicia con “set” y termina con el nombre de la variable de estado.

El hook useState recibe por parámetro el valor inicial de la variable de estado

A diferencia del estado en un componente de tipo clase, la variable de estado usando este hook no tiene que ser necesariamente un objeto.

Nuestro estado puede ser una variable de los siguientes tipos:

- String.
- Boolean.
- Number.
- Float.
- Null.
- Undefined.

- Object.
- Array.

Ejemplo de tipo de variable inicial en useState

```
const MyComponent = () => {
  const [stateVariable, setStateVariable] = React.useState(false); // <<< pasamos un\
 booleano
};
```

Para valores de estado iniciales que sean resultado de un cálculo costoso, puedes pasar como valor inicial una función que retorne dicho valor calculado.

Ejemplo de valor inicial usando una función

```
const MyComponent = () => {
  const [
    stateVariable,
    setStateVariable
  ] = React.useState(() => someExpensiveCalc(); // aqui
};
```

Ten en cuenta que los valores iniciales del estado se ejecutan solo una vez y es cuando se monta el componente

Puedes declarar más de un estado local con useState

Va a ser común que necesites más de una variable de estado en un componente.

En lugar de tener una sola variable de estado como un objeto que contenga múltiples valores, probablemente sea mejor separar por “preocupaciones” (separation of concerns principle).

Ejemplo múltiples variables de estado

```
// dentro de un componente funcional

```

```
const [username, setUsername] = useState('');
const [isAdult, setIsAdult] = useState(false);
const [age, setAge] = useState(0);
const [userTags, setUserTags] = useState([]);
```

Te recomiendo que trates de mantener tus variables de estado lo más simples posibles y separarlas en diferentes variables cada que puedas.

Sin embargo, a veces vas a necesitar usar arrays u objetos como variables de estado, en esos casos te recomiendo que trates de mantenerlos lo más planos posibles.

Para estructuras más complejas (un objeto de objetos, un array de objetos que tienen objetos, etc); lo recomendable es usar el hook useRedux como veremos en su capítulo correspondiente.

Actualizando Arrays con useState

Es importante que tengas en cuenta la manera correcta de actualizar el estado cuando usas este tipo de datos ya que puedes caer en comportamientos inesperados.

Para actualizar el estado de manera adecuada cuando manejas arrays, necesitas pasar el array completo en la función que actualiza el estado.

A continuación unos ejemplos de como NO y como SI agregar un nuevo elemento.

Ejemplo de agregar un nuevo elemento a un array como variable de estado

```
const Example = () => {
  const [tags, setTags] = useState([]);

  // NO
  tags.push('new value');

  // SI
  setTags([...tags, 'new value']);
}
```

La razón de por que no debemos mutar nuestro array con métodos como `push`, `pop`, etc, es porque React no detecta estos cambios y no hará un re renderizado.

Esta sintaxis `setTags([...tags, 'new value']);` es propia de ES6 y se llama `spread operator`.

Lo que hacemos con ella es crear un NUEVO array cuyo contenido va a ser igual a todo lo que tenga el array `tags` y aparte, un nuevo valor que es `'new value'`.

Lo mismo aplica si tienes un array de objetos en vez de un array de cadenas: `setTags([...tags, { value:'new value'}])` (asumiendo que `tags` es un array de objetos).

Modificar el contenido de un Array como variable de estado

Para actualizar elementos existentes dentro de un array podemos usar diferentes estrategias.

La más fácil es crear un nuevo array con el valor modificado usando `.map` como se ve a continuación.

Ejemplo de modificar un elemento existente en un array como variable de estado

```
1 const Example = () => {
2   const [tags, setTags] = useState(["one", "two", "three"]);
3
4   // update
5   const indexToUpdate = 1;
6   const newValue = "four?";
7   const tagsUpdated = tags.map((value, index) => {
8     if (index === indexToUpdate) {
9       return newValue;
10    }
11
12    return value;
13  });
14
```

```
15  setTags(tagsUpdated);
16 };
```

Ahora bien, un array puede contener objetos en vez de valores primitivos como cadenas. Veamos el ejemplo de como actualizar cuando es un array de objetos.

Ejemplo de modificar un elemento de un array de objetos como variable de estado

```
1 const Example = () => {
2  const [tags, setTags] = useState([
3    { id: 1, value: "one" },
4    { id: 2, value: "two" },
5    { id: 3, value: "three" }
6  ]);
7
8  // update
9  const idToUpdate = 2;
10  const newValue = { id: 2, value: "four?" };
11  const tagsUpdated = tags.map((value) => {
12    if (value.id === idToUpdate) {
13      return newValue;
14    }
15
16    return value;
17  });
18
19  setTags(tagsUpdated);
20 };
```

Eliminar un elemento de un array como variable de estado

Para eliminar un elemento necesitamos pasar un nuevo array sin el elemento que queremos eliminar.

La manera que es más cómoda para hacerlo es usando la función .filter que viene en el prototype de Array.

Ejemplo de eliminar un elemento de un array como variable de estado

```
const Example = () => {
 const [tags, setTags] = useState([
  { id: 1, value: "one" },
  { id: 2, value: "two" },
  { id: 3, value: "three" }
 ]);

 // remove
 const idToUpdate = 2;

  const tagsUpdated = tags.filter(({ id }) => id !== idToUpdate);

  setTags(tagsUpdated);
};
```

Actualizando Objetos con useState

Al igual que cuando actualizamos un array, cuando actualizamos un objeto también necesitamos pasar el objeto completo en la función que actualiza dicho estado.

Ejemplo de agregar valores a un objeto como variable de estado

```
const Example = () => {
 const [userData, setUserData] = useState({
   name: "",
   age: ""
 });

 // NO
 userData.name = "Juanito";

  // SI
  setUserData({ ...userData, name: "Juanito Banana" });
  setUserData({ ...userData, age: "20" });

```

```
14 // SI
15 const property = 'name';
16
17 setUserData({
18   ...userData,
19   [property]: "Juanito Banana"
20 });
21 };
```

Nota que también estamos usando el spread operator para crear un nuevo objeto con las propiedades correspondientes.

En el segundo ejemplo estamos modificando la propiedad name de manera dinámica usando [property]: "Juanito Banana" donde property = 'name';.

Para editar una propiedad es exactamente igual a como cuando le asignamos un valor.

Recapitulación del hook useState

En este capítulo hemos visto los principales casos de uso que vas a necesitar en tu dia a dia programando:

- Definir uno o varios hooks.
- Definir el estado inicial del hook considerando valores iniciales complejos.
- Actualizar un array como variable de estado.
- Actualizar un objeto como variable de estado.

¡Te veo en el siguiente capítulo!

Hook de efectos secundarios useEffect

EL hook useEffect es el segundo hook que más vas a usar debido a que nos ayuda a todo lo relativo de efectos secundarios.

¿Y qué rayos es un efecto secundario, Juanito?

Un efecto secundario o simplemente efecto, lo puedes ver como la ley de causa y efecto: todo efecto es como consecuencia de una o varias causas que lo provocan.

Tal como lo dice la documentación de React, los siguientes son ejemplos de efectos secundarios:

- Peticiones de datos.
- Establecimiento de suscripciones.
- Actualizaciones manuales del DOM.

Esto ya se viene haciendo en React en los componentes de tipo clase por medio de los ciclos de vida.

Pues bien, estos efectos secundarios ahora también los podemos hacer en componentes funcionales.

La manera en que luce el hook `useEffect` es la siguiente.

Ejemplo del hook useEffect

```
const MyComponent = () => {
  useEffect(() => {
    console.log('useEffect ejecutado');
  })

  return ':)';
}
```

Vamos a entrar más en materia a continuación.

Características de use Effect

1. Recibe dos parámetros: el primero una función y el segundo un array cuyos valores serán variables de las que depende nuestro efecto (este array es opcional).
2. Se ejecuta en cada renderizado incluyendo el primero.
3. Puedes usar más de un useEffect dentro de un componente.
4. Está diseñado para que si la función que pasamos por parámetro retorna a su vez otra función, React va a ejecutar dicha función cuando se desmonte el componente.

Ejemplos de useEffect

```
const MyComponent = ({ someProp }) => {
 useEffect(() => {
  console.log("se ejecuta cada render");
  console.log("incluso el inicial");
 });

 useEffect(() => {
  console.log("nuevo valor de someProp", someProp);
 }, [someProp]);

 useEffect(() => {
  // code
  return () => console.log("componente desmontado");
 });

 return ":)";
};
```

Esto es lo mínimo necesario que necesitas saber para comenzar a usar useEffect.

Pero para ponerla más fácil, vamos a ver ejemplos de casos prácticos del mundo real en el uso de este hook.

Petición de datos (componentDidMount y componentDidUpdate)

El siguiente ejemplo va a ser sólo para fines educativos y es simular una llamada a una API.

Lo importante de este ejemplo es comparar la llamada a una api cuando se monta un componente cuando usas un componente de tipo clase y cómo es su equivalente con un hook.

Ejemplo de llamada a una api al montarse el componente de tipo clase

```
class PostData extends Component {
  state = {
    data: []
  };

  componentDidMount() {
    const data = myApi.fakeFetch();
    this.setState({ data });
  }

  render() {
    return this.state.data.map(
      ({ label }) => (<p>{label}</p>)
    );
  }
}
```

Ahora vamos a hacer su equivalente con useEffect.

Ejemplo de llamada a una api con useEffect

```
const SecondPostData = () => {
  const [data, setData] = useState([]);

  useEffect(() => {
    const data = myApi.fakeFetch();
    setData(data);
```

```
}, []);

return data.map(({ label }) =>
  <p>{label}</p>
);
};
```

Como puedes ver, el código es menos y más simple.

Este efecto se va a ejecutar sólo cuando el componente se monte la primera vez y no volverá a ejecutarse.

Para lograr esto pasamos a useEffect un array vacío como segundo parámetro.

Recuerda que ese array de segundo parámetro sirve para indicarle a React las dependencias de ese efecto.

Si el array de dependencias está vacío, no habrá nada que lo vuelva a ejecutar, por lo que hacerlo de este modo equivale al ciclo de vida componentDidMount.

Ahora vamos a extender el ejemplo en el escenario en el que se usa el ciclo componentDidUpdate para hacer una segunda llamada a una api cuando depende de un dato como el id de un usuario.

Ejemplo de llamada a api al montarse y actualizarse un componente de tipo clase

```
class PostData extends Component {
 state = {
  data: []
 };

 componentDidMount() {
  const { userId } = this.props;
  const data = myApi.fakeFetch(userId);
  this.setState({
   data
  });
```

```
  }

  // nuevo
  componentDidUpdate(prevProps) {
    const { userId } = this.props;

    if (prevProps.userId !== userId) {
      const data = myApi.fakeFetch(userId);
      this.setState({
        data
      });
    }
  }

  render() {
    return this.state.data.map(
    ({ label }) => <p>{label}</p>);
  }
}
```

Nada fuera de lo común. Estamos haciendo una llamada a la api cada vez que el user id cambie.

Ahora veamos su equivalente en un componente funcional.

Ejemplo de llamada a api al montarse y actualizarse un componente funcional

```
const SecondPostData = ({ userId }) => {
  const [data, setData] = useState([]);

  useEffect(() => {
    const data = myApi.fakeFetch(userId);
    setData(data);
  }, [userId]); // nuevo

  return data.map(({ label }) => <p>{label}</p>);
};
```

Como puedes ver, sólo hemos pasado la variable userId como dependencia dentro del array del useEffect.

Esto significa que nuestro efecto se va a ejecutar al montarse el componente y cada vez que la variable userId cambie, logrando así nuestro objetivo.

El código anterior lo puedes ejecutar y editar [en este link]

Iniciar y eliminar suscripciones (componentDidMount, componentWillUnmount)

Vamos a usar de ejemplo la suscripción del evento de cuando se presiona la tecla Enter del teclado.

Ya que es un efecto secundario, iniciamos creando la suscripción dentro de un useEffect.

Cuando realizamos una suscripción por lo común vamos a necesitar "sanarla" o removerla cuando se desmonta el componente como en este ejemplo.

Para ello en nuestro useEffect donde inicializamos la suscripción vamos a retornar una función que contenga el código necesario para removerla.

Ejemplo de suscripción

```
const MyComponent = ({ someProp }) => {
 useEffect(() => {
  window.addEventListener("keydown", handleKeydown);

  return () =>
    window.removeEventListener(
      "keydown",
      handleKeydown,
    );
```

```
});

const handleKeydown = ({ keyCode }) => {
  const enterKeyCode = 13;

  if (keyCode === enterKeyCode) {
    alert("tecla Enter presionada");
  }
};

return "Presiona Enter :)";
};
```

El mismo principio aplica para cualquier tipo de suscripciones que realices.

Recapitulación del hook useEffect

En este capítulo hemos visto lo siguiente:

- El hook `useEffect` recibe dos parámetros: una función que React ejecutará en cada renderizado y un array de dependencias como opcional.
- Para emular `componentDidMount` pasa un array vacío.
- Para emular `componentWillUnmount` retorna una función con la lógica que quieres correr cuando se desmonte el componente, usado comúnmente en las suscripciones.
- Para emular `componentDidUpdate` pasa las variables dependientes del estado en el array de dependencias.
- Puedes usar más de un `useEffect` en el mismo componente.

Ahora pasemos al siguiente capítulo :D

Hook para memorizar funciones useCallback

El hook useCallback sirve para mejorar el rendimiento de nuestros componentes en base a memorizar funciones que se usan como callbacks.

Así es como luce:

Ejemplo de useCallback de la documentación oficial de reactjs.org

```
const memoizedCallback = useCallback(
  () => {
    doSomething(a, b);
  },
  [a, b],
);
```

Este ejemplo es sólo para que te familiarices con la manera en que se define este hook.

Antes de continuar es importante tener en cuenta ciertos detalles.

Cuando un prop cambia, se ejecuta un re render. ¿Cierto?

Pues bien, este hook es útil en el escenario en el que **pasas funciones por props a componentes hijos.**

Memorizar una función ayuda a evitar re renders innecesarios debido a la manera en que funciona Javascript como lenguaje.

Igualdad referencial en Javascript

```
true === true // true
false === false // true
1 === 1 // true
'a' === 'a' // true
{} === {} // false
[] === [] // false
() => {} === () => {} // false
const z = {}
```

```
z === z // true
```

Nota: las funciones para actualizar el estado que retorna `useState` tienen garantía de que serán estables todo el tiempo, por lo que nos podemos despreocupar de ellas.

Al usar un callback memorizado logramos que no se vuelva a definir ese callback en cada render a menos que alguna de sus dependencias cambie.

Si te suena confuso no te preocupes que para eso está este ebook :)

Vamos a ver un ejemplo a continuación, sigue leyendo :D

Ejemplo de optimización con useCallback (incluyendo React.memo)

Antes mencioné que este hook es útil cuando pasas funciones como props a componentes hijos.

Para ilustrar bien este ejemplo, vamos a hacer una mini app que consiste en tener una lista de comida con la capacidad de remover elementos de dicha lista.

Primero veremos el problema de los re renders innecesarios y después lo resolveremos con `React.memo` y el hook `useCallback`.

Nota: `React.memo` no es el hook `useMemo` que veremos en su capítulo correspondiente. Si no lo conoces, da click aquí para ver su documentación.

Vamos a hacer los siguientes componentes:

- FoodContainer: estado de lista y un input text para escribir
- FoodList: renderiza el listado de comida.
- FoodItem: renderiza un elemento de la lista de comida.

lista de alimentos

```
const food = [
  { id: 1, name: "pizza" },
```

```
  { id: 2, name: "hamburger" },
  { id: 3, name: "hot-dog" },
  { id: 4, name: "tacos" },
  { id: 5, name: "pizza again :)" }
];
```

FoodList container será un componente algo grande.

FoodContainer

```
const FoodContainer = () => {
 console.log("FoodContainer rendered");

 const [foodList, setFoodList] = useState(food);
 const [textInput, setTextInput] = useState("");

 const handleChange = ({ target }) =>
  setTextInput(target.value);

 const removeItem = (id) =>
  setFoodList(foodList.filter((foodItem) =>
    foodItem.id !== id));

 return (
  <>
   <h2>My Food List</h2>
   <p>
    New food
    <input
     value={textInput}
     onChange={handleChange}
    />
   </p>
   <FoodList
    foodList={foodList}
    removeItem={removeItem}
   />
```

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

Ahora FoodList y FoodItem.

FoodList

```
const FoodList = ({ foodList, removeItem }) => {
  console.log("FoodList rendered");
  return (
    <ul>
      {foodList.map((item) => (
        <FoodItem
          key={item.id}
          item={item}
          removeItem={removeItem}
        />
      ))}
    </ul>
  );
};
```

FoodItem

```
const FoodItem = ({ item, removeItem }) => {
  console.log("FoodItem rendered");
  return (
    <>
      <li>{item.name}</li>
      <button
        onClick={() => removeItem(item.id)}>
        Remove :(
      </button>
    </>
  );
};
```

Escribe en el input text y observa los console logs desde la herramienta donde se ejecuta el código, no de la consola de tu navegador.

En este link lo puedes ejecutar.

Como puedes ver, se hacen re renders en los componentes debido a que cambia la variable de estado que usamos para controlar el input, aunque realmente no cambien los valores de la lista de alimentos.

El primer cambio que haremos es envolver los componentes `FooList` y `FoodItem` con `React.memo` para que no se re rendericen dados los mismos props.

Nota: hacemos import de memo del modo: `import React, { memo } from 'react';`

FoodList con memo

```
const FoodList = memo(({ foodList, removeItem }) => {
  console.log("FoodList rendered");
  return (
    <ul>
      {foodList.map((item) => (
        <FoodItem
          key={item.id}
          item={item}
          removeItem={removeItem} />
      ))}
    </ul>
  );
});
const FoodItem = memo(({ item, removeItem }) => {
  console.log("FoodItem rendered");
  return (
    <>
      <li>{item.name}</li>
      <button
```

```
      onClick={() => removeItem(item.id)}>
      Remove :(
    </button>
  </>
 );
});
```

Haz este cambio en el link del código que compartí anteriormente.

Si escribes de nuevo en el input, ¡verás que se siguen re renderizando todos los componentes! ¿Por qué?

La razón es que React considera que el prop `removeItem` no es equivalente al anterior prop a través de los re renderizados.

¿Recuerdas la parte de este capítulo donde está el ejemplo de la igualdad referencial?

Pondré la parte importante de este ejemplo:

```
() => {} === () => {} // false
```

Para resolver esto ahora pasamos a usar `useCallback` :D

Dentro de FoodContainer

```
const removeItem = useCallback(
  (id) => setFoodList(foodList.filter((foodItem) =>
    foodItem.id !== id)),
  [foodList]
);
```

El componente completo luce de este modo:

```
const FoodContainer = () => {
```

```
2  console.log("FoodContainer rendered");
3
4  const [foodList, setFoodList] = useState(food);
5  const [textInput, setTextInput] = useState("");
6
7  const handleChange = ({ target }) =>
8   setTextInput(target.value);
9
10  const removeItem = useCallback(
11    (id) => setFoodList(foodList.filter((foodItem) =>
12      foodItem.id !== id)),
13    [foodList]
14  );
15
16  return (
17    <>
18      <h2>My Food List</h2>
19      <p>
20        New food <input
21          value={textInput}
22          onChange={handleChange} />
23      </p>
24      <FoodList
25        foodList={foodList}
26        removeItem={removeItem} />
27    </>
28  );
29 };
```

Usando en React.memo y el hook useCallback ahora hemos logrado optimizar los re renderizados de este ejemplo.

A decir verdad, este es un ejemplo semi forzado porque la estructura de los componentes es candidata a ser re factorizada para no tener la necesidad de pasar todos los props en el árbol de componentes.

Pero he querido hacerlo de este modo para poder ilustrar el ejemplo lo más fácil posible.

¿Debo memorizar funciones todo el tiempo?

No. Es muy importante elegir sabiamente en base a la relación costo - beneficio.

El costo por memorizar funciones es que agregamos más sentencias de código que necesitarán memoria y que además hacen nuestro código un poco verboso.

React es realmente muy rápido en los renderizados.

Probablemente notarás mejoras en el performance cuando tengas una gran cantidad de componentes cuyo costo por un re renderizado implique una gran cantidad de cálculos.

Sin embargo, es bueno que conozcas `useCallback` para que lo tengas en cuenta en tus aplicaciones.

Recapitulación del hook useCallback

- Usa este hook en las funciones que pases por props a componentes hijos que ocasionen re renderizados innecesarios.
- Si la función depende de otras variables, pásalas en el array de dependencias de este hook.
- Puedes usar `React.memo` en conjunto para complementar una optimización de re renders.
- NO (en mayúsculas) necesitas aplicar `useCallback` en cada función que definas ya que tiene un costo. Hazlo sólo cuando realmente exista una mejora significativa en el performance.

Este ha sido un hook interesante, ¿No lo crees?.

Ahora vamos al siguiente capítulo donde veremos una segunda estrategia de optimización más interesante aún.

Hook para optimizar cálculos useMemo

El hook `useMemo` sirve para memorizar valores.

Recibe por parámetro una función para crear un valor a memorizar y por segundo parámetro un array de dependencias.

Ejemplo de useMemo de reactjs.org

```
const memoizedValue = useMemo(() =>
computeExpensiveValue(a, b), [a, b]);
```

Es especialmente útil cuando el valor a memorizar es producto de un cálculo que consume "mucha" memoria y procesamiento.

Ya que "mucha" tiende a ser subjetivo, considera la complejidad de un cálculo (o un algoritmo) en base a lo siguiente:

- Constant: O(1)
- Logarithmic: O(log n)
- Linear: O(n)
- Linearithmic: O(n log n)
- Quadratic: $O(n^2)$
- Expontential: O(2^n)
- Factorial: O(n!)

Un ejemplo clásico de una función que tenga una complejidad cuadrática ($O(n^2)$) es cuando tienes un ciclo dentro de otro ciclo.

función de complejidad cuadrática

```
const expensiveCalc = (myArray) => {
  for (let i = 0; i < myArray.length; i++) {
    for (let j = 0; j < myArray.length; j++) {
      // code...
    }
```

```
}
}
```

La pregunta es: **¿Cuántas veces se ejecutará el código dentro del segundo for si pasamos un array con 10,000 elementos?**

La respuesta es: 10,000 al cuadrado.

Ahora imagina que tienes algo como esto en un componente:

```
const expensiveCalc = (myArray) => {
  let c = 0;

  for (let i = 0; i < myArray.length; i++) {
    for (let j = 0; j < myArray.length; j++) {
      c = c + 1;
    }
  }

  console.log('iteraciones:', c);
}

const MyComponent = ({ someList }) => {
  const result = expensiveCalc(someList);

  return <p>resutl: {result}</p>
}
```

Prueba ejecutar lo anterior dando click aquí y observa la consola de la plataforma (no la de tu navegador).

Probablemente no tengas problemas en un inicio dependiendo de lo que necesites lograr en tu componente.

Puede que en un inicio no tengas muchos valores en tu array.

Incluso puede que logres optimizar la complejidad de ese algoritmo usando otra estructura de datos como un diccionario o una tabla hash.

Pero si quieres una solución basada en lo que tiene React, en este caso el hook useMemo viene como anillo al dedo.

¿Ya probaste el código anterior? Si no lo has hecho, te animo a que lo hagas porque ahora vamos a ver la manera de optimizarlo con useMemo.

Ejemplo de useMemo para optimizar un cálculo

```
1 const MyComponent = ({ someList }) => {
2   // Cambio aquí
3   const result = useMemo(() =>
4     expensiveCalc(someList), [someList]);
5
6   return <p>Iteraciones: {result.toLocaleString()}</p>;
7 };
```

Así de simple. Sólo cambiando una línea de código hemos logrado hacer una optimización de algo que potencialmente puede afectar el rendimiento de nuestra aplicación.

Prueba el código actualizado dando click aquí.

PureComponent, React.memo y useMemo

Primero vamos a definir la identidad de cada elemento:

- PureComponent es una API para declarar componentes de tipo clase.
- React.memo es un HOC (Componente de orden superior por sus siglas en inglés).
- useMemo es un hook del core de React.

No podemos hacer ejemplos de equivalencia entre estos elementos porque sería como comparar peras con manzanas.

Lo que podemos hacer es observar cómo funcionan cada uno de ellos y es lo que vamos a hacer.

Para quien no esté familiarizado, PureComponent es parecido a Component que usamos para definir componentes de tipo clase (ambos exportados en el import de React).

PureComponent es usado para componentes que al tener los mismos props y state, renderiza el mismo resultado.

Implementa shouldComponentUpdate internamente, por lo que dados los mismos props y state, el componente va a renderizar el mismo output.

Considera lo siguiente:

```
class Title extends PureComponent {
  render() {
    return <h1>{this.props.value}</h1>;
  }
}

// <Title value="React Hooks" />
```

Este es un componente que sólo renderiza un elemento visual, no tiene lógica de llamada a api ni nada por el estilo.

En este caso es conveniente usar PureComponent debido a que nos ahorra hacer la comparación manualmente en si debe o no renderizar.

Su equivalente en Component usando shouldComponentUpdate es:

```
class Title extends Component {
  shouldComponentUpdate(nextProps) {
    // retorna true si son diferentes
    // si son diferentes, renderiza de nuevo
    return nextProps.value !== this.props.value;
  }

  render() {
    return <h1>{this.props.value}</h1>;
```

```
10   }
11 }
```

Aunque sigue siendo preferible usar PureComponent, no sólo por ahorrarnos líneas de código sino por la propia recomendación de la (documentación en React)[https://es.reactjs.org/docs/react-api.html#reactpurecomponent].

En esta solución basta con crear un componente de tipo clase con PureComponent y React hace todo el trabajo por nosotros.

El HOC React.memo nos permite optimizar los renderizados sólo cuando los props cambian.

Consideremos el componente Title en forma de componente funcional.

```
1 const Title = (value) => <h1>{value}</h1>;
```

Ahora vamos a hacer el wrapper con React.memo para optimizar su renderizado.

```
1 const Title = (value) => <h1>{value}</h1>;
2
3 const TitleWithMemo = React.memo(Title);
4
5 // <TitleWithMemo value="memo" />
```

Y con esto es suficiente.

Para useMemo ya hemos visto un ejemplo al principio.

Lo importante a destacar aquí es que este hook nos va a ayudar a memorizar valores dentro de un componente.

Recapitulación del hook useMemo

En este capítulo hemos visto lo siguiente:

- El hook `useMemo` nos sirve para memorizar valores.
- Recibe una función que retorna el valor a memorizar y como segundo parámetro opcional, un array de dependencias.
- Si no pasamos el array de dependencias, se ejecutará en cada renderizado.
- Especialmente útil para ahorrar cálculos costosos en cada renderizado como vimos en el ejemplo práctico.
- `PureComponet` nos crea un componente que dados los mismos props y estado, ahorra el renderizado.
- `React.memo` es un HOC que permite optimizar un componente ahorrando los renderizados al tener los mismos props.

Con esto hemos cubierto gran parte del tema para hacer optimizaciones en componentes en React.

Si aún no has leído el capítulo de `useCallback` te recomiendo que lo hagas para tener un complemento de este capítulo.

Si ya lo leíste, ahora te invito a que pasemos al siguiente :D

Hook para estados complejos useReducer

Este capítulo va a abarcar varios conceptos que si no estás familiarizado con ellos, puede que te sientas un poco abrumado.

Pero descuida, he hecho mi mejor esfuerzo para ponerla fácil y que al final del capítulo logres dominar este hook.

Si ya estás familiarizado con redux, será bastante simple de entender y siéntete libre de brincar hasta la parte del ejemplo de useReducer.

Dicho lo anterior, comencemos :).

El hook useReducer nos permite manejar estados "complejos" por medio de una función reductora.

¿Y qué es un estado "complejo"?

Un estado complejo es aquel que tiene una estructura de varios niveles, por ejemplo: un objeto de objetos que contiene arrays.

Ejemplo de un estado complejo

```
{
  foo: {
    faa: {
      test: [],
      testB: ''
    }
  },
  fee: [],
  fii: {
    testC: [],
    testD: {}
  }
}
```

En caso de que requieras un estado como el ejemplo anterior y no sea viable dividirlo usando diferentes useState, lo recomendado es que uses useReducer.

¿Y cómo luce este hook?

Ejemplo de useReducer de reactjs.org

```
const [state, dispatch] = useReducer(
  reducer,
  initialArg,
  init
);
```

Donde reducer es simplemente una función de Javascript que actualiza el estado, initialArg es el estado inicial e init es una función opcional para también definir el estado inicial.

Este hook devuelve un array donde en la posición cero es el estado y en la posición uno es un dispatch que es una función que usaremos para actualizar el estado.

En realidad, es muy parecido al hook useState:

```
const initialState = false;
const [state, setState] = useState(initialState);
```

Veamos más sobre el hook useReducer haciendo una comparación con redux y ejemplos sobre su uso en los siguientes apartados.

Funciones reductoras (reducers)

Un reducer es una función de Javascript que recibe por parámetro el valor del estado actual y por segundo parámetro un objeto plano de Javascript llamado action con la información necesaria para actualizar el estado.

“Kha?”

Suena más complejo de lo que parece. Mira el siguiente ejemplo de un reducer:

Ejemplo de un reducer de reactjs.org

```
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}
```

Como podemos ver, el reducer recibe el valor del estado y un action.

El parámetro action es un objeto plano de Javascript y tendrá por lo menos una propiedad llamada type que usamos como una cadena, ejemplo: { type: 'ADD_TODO' }.

Según el valor de action.type, es el cambio de estado a ejecutar.

Es importante que sepas que no mutamos el estado sino que creamos y devolvemos uno nuevo.

Hay quienes prefieren evitar el uso de switch pero debes saber que es opcional usarlo. Puedes usar una serie de sentencias if también. Depende de ti.

Este ejemplo de la documentación oficial de reactjs.org es sólo una muestra de cómo podría lucir un reducer para un estado de un contador, pero es sólo para fines de ejemplo.

Al inicio comentamos que useReducer es recomendado para estados complejos y es lo que veremos en el ejemplo práctico.

Ejemplo de un componente con useReducer

Vamos a hacer un componente para manejar una lista de cosas por hacer (TODO list).

Antes de implementar useReducer primero vamos a definir el estado inicial y la función reductora que nos pide este hook al ejecutarlo.

Primero hay que pensar en cuál es el estado más simple que puede necesitar nuestro componente de cosas por hacer.

Consideremos lo siguiente:

- El usuario va a poder agregar nuevos elementos en la lista de cosas por hacer.
- El usuario necesita visualizar en la UI la lista de las cosas por hacer.
- El usuario va a poder marcar como completado o no completado un elemento de lista.

Te propongo que usemos la siguiente estructura:

Estado de TODO list

```
const state = [
  {
    id: 1,
    name: "Terminar de leer el capítulo de useReducer",
    isCompleted: false
  }
];
```

Un array de objetos planos en el que cada objeto tendrá un id para identificarlo (podríamos usar el índice del array pero no es lo común en aplicaciones reales), un name y un booleano isCompleted que usaremos para marcar como completado o no completado.

Ahora hagamos nuestro reducer incluyendo la lógica necesaria para agregar un nuevo elemento de nuestro TODO solamente de momento.

Reducer con lógica para agregar un TODO

```
const reducer = (state, action) => {
 if (action.type === "ADD_TODO") {
  const { name } = action.payload;

  return [
   ...state,
   {
    id: uuidv4(),
    name,
    isCompleted: false
   }
  ];
 }
};
```

Como es común, el reducer recibe el state y action. Estos valores serán pasados internamente por useReducer.

Lo que nos debemos preocupar aquí es en colocar la lógica para actualizar el estado, cosa que hacemos en el primer if.

Asumimos que tendremos un action.type igual a "ADD_TODO" y cuando lo recibamos vamos a retornar el nuevo estado.

Para retornar el nuevo estado, vamos a usar el spread operator para crear un nuevo array agregando todos los valores actuales de state, ya que state es un array.

Después agregamos el nuevo objeto con el id (usamos una biblioteca llamada uuid para que nos genere los id únicos), el name que proviene de action.payload e isCompletedcomo false por default.

Ahora vamos a agregar la lógica para actualizar isCompleted de un elemento del TODO en particular.

Veremos únicamente la parte correspondiente dentro del reducer.

Lógica para actualizar isCompleted de un elemento de la lista

```
// dentro del reducer, este IF nuevo
if (action.type === "TOGGLE_IS_COMPLETED") {
  const { id } = action.payload;

  const newState = state.map((singleTodo) => {
    if (singleTodo.id === id) {
      return {
        ...singleTodo,
        isCompleted: !singleTodo.isCompleted
      };
    }

    return singleTodo;
  });

  return newState;
}
```

Esta lógica es muy diferente a la de agregar. No hacemos spread ni nada por el estilo.

Primero obtenemos el id del elemento a actualizar por medio de action.payload, después creamos el nuevo estado por medio de state.map (.map retorna un nuevo array).

Dentro de .map está la parte importante.

Lo que hacemos aquí es comparar elemento por elemento para saber si el elemento de la lista coincide con el id que nos interesa actualizar.

Si coincide, retornamos todo lo que contenga ese elemento de la lista y su propiedad isCompleted le asignamos el valor contrario que tenga en ese momento.

Ejemplo:

```
// Si...
singleTodo.isCompleted = true

// Entonces...
!singleTodo.isCompleted // igual a false

// y viceversa

// Si...
singleTodo.isCompleted = false

// Entonces...
!singleTodo.isCompleted // igual a true
```

Si no coincide con el `id`, simplemente retornamos el elemento sin modificarlo.

Al final retornamos al nuevo estado con su valor modificado.

Nuestro `reducer` completo luce del siguiente modo:

reducer completo

```
const reducer = (state, action) => {
  if (action.type === "ADD_TODO") {
    const { name } = action.payload;

    return [
      ...state,
      {
        id: uuidv4(),
        name,
        isCompleted: false
      }
    ];
  }

  if (action.type === "TOGGLE_IS_COMPLETED") {
```

```
  const { id } = action.payload;

  const newState = state.map((singleTodo) => {
    if (singleTodo.id === id) {
      return {
        ...singleTodo,
        isCompleted: !singleTodo.isCompleted
      };
    }

    return singleTodo;
  });

  return newState;
}

return state;
};
```

Ahora que tenemos estos elementos definidos, pasemos a crear nuestro componente :)

Componente sin reducer aún

```
const Todo = () => {
 const [todoText, setTodoText] = useState("");

 const handleChange = ({ target }) =>
  setTodoText(target.value);

 return (
  <>
   <p>
    Nuevo TODO:
    <input
     type="text"
     value={todoText}
     onChange={handleChange}
    />
```

```
      <button>Agregar</button>
    </p>

    <h2>Listado</h2>
    <ul>
      <li>Test</li>
    </ul>
  </>
 );
};
```

Tenemos el input para escribir el nuevo TODO y lo controlamos con un estado usando useState.

El listado está "hardcodeado" sólo para tener una idea de cómo mostraremos los elementos.

Ahora pasemos a agregar el useReducer :).

```
// dentro del componente agregamos:
const [todoText, setTodoText] = useState("");

// AQUI
const [state, dispatch] = useReducer(reducer, initialState);

// ...resto del Código
```

Y modificamos la parte del render para que muestre los valores de nuestro state. Recuerda que pusimos un valor inicial en las cosas por hacer, por lo que será lo que se renderice en la lista.

renderizar listado (dentro del return del componente)

```
<ul>
 {
  state.map(({ name, isCompleted, id }) => {
    const style = {
```

```
      textDecoration: isCompleted
      ? "line-through" : "inherit"
    };

    return (
      <li key={id} style={style}>
        {name}
      </li>
    );
  })
}
</ul>
```

De momento, nuestro componente luce del siguiente modo:

```
const Todo = () => {
  const [todoText, setTodoText] = useState("");
  const [state, dispatch] = useReducer(reducer, initialState);

  const handleChange = ({ target }) =>
    setTodoText(target.value);

  return (
    <>
      <p>
        Nuevo TODO:
        <input
          type="text"
          value={todoText}
          onChange={handleChange}
        />
        <button>Agregar</button>
      </p>

      <h2>Listado</h2>

```

```
    <ul>
      {state.map(({ name, isCompleted, id }) => {
        const style = {
          textDecoration: isCompleted
          ? "line-through" : "inherit"
        };

        return (
          <li key={id} style={style}>
            {name}
          </li>
        );
      })}
    </ul>
  </>
);
};
```

Ahora vamos a colocar la lógica para que al dar click en el botón “Agregar”, nos agregue el nuevo elemento en nuestro estado.

```
const handleClick = () => {
  dispatch({
    type: "ADD_TODO",
    payload: { name: todoText }
  });
  setTodoText("");
};

// en el botón
 <button onClick={handleClick}>Agregar</button>
```

Ejecutamos el dispatch para actualizar el estado. Le pasamos el type y el payload necesarios como se ve en el ejemplo.

Al final, reseteamos el valor de todoText para limpiar el contenido del input.

Por último, vamos a agregar la capacidad de marcar un elemento como completado o no completado al dar click a un elemento del listado.

```
const handleToggle = (id) => {
  dispatch({
    type: "TOGGLE_IS_COMPLETED",
    payload: { id }
  });
};

// en el elemento li
  return (
    <li
      key={id}
      style={style}
      onClick={() => handleToggle(id)}
    >
      {name}
    </li>
);
```

El código completo luce del siguiente modo:

Ejemplo completo de useReducer

```
const Todo = () => {
  const [todoText, setTodoText] = useState("");
  const [state, dispatch] = useReducer(reducer, initialState);

  const handleChange = ({ target }) =>
    setTodoText(target.value);

```

```
8  const handleClick = () => {
9    dispatch({
10     type: "ADD_TODO",
11     payload: { name: todoText }
12   });
13   setTodoText("");
14 };
15
16 const handleToggle = (id) => {
17   dispatch({
18     type: "TOGGLE_IS_COMPLETED",
19     payload: { id }
20   });
21 };
22
23 return (
24   <>
25     <p>
26       Nuevo TODO:
27       <input
28         type="text"
29         value={todoText}
30         onChange={handleChange}
31       />
32       <button onClick={handleClick}>Agregar</button>
33     </p>
34
35     <h2>Listado</h2>
36
37     <ul>
38       {state.map(({ name, isCompleted, id }) => {
39         const style = {
40           textDecoration: isCompleted
41           ? "line-through" : "inherit"
42         };
43
44         return (
```

```
45          <li
46            key={id}
47            style={style}
48            onClick={() => handleToggle(id)}
49          >
50            {name}
51          </li>
52        );
53      })}
54    </ul>
55  </>
56 );
57 };
```

Puedes ejecutar y editar el código fuente en [este link]

Con esto hemos terminado nuestro ejemplo :D

Diferencia entre useReducer y Redux

El nombre de useReducer da pie a confundirlo con redux, pero en realidad son cosas completamente diferentes.

- useReducer es un **hook** de React para actualizar un estado interno por medio de una función llamada reducer.
- redux es una **arquitectura** que nos permite abstraer el manejo de un estado global en una aplicación.

Redux es algo más complejo que abarca el nivel de decisiones de arquitectura de una aplicación, siendo react-redux una biblioteca que nos permite integrarlo fácilmente en React.

La confusión puede caer en el uso de funciones llamadas reducers que son usadas como una parte del funcionamiento de redux.

Recapitulación del hook useReducer

En este capítulo hemos visto lo siguiente:

- Los conceptos de reducer, action y su funcionamiento con useReducer.
- Un ejemplo práctico de un TODO aplicando useReducer.
- Diferencias entre redux y useReducer.

Espero que este capítulo te haya dado luces para que puedas comenzar a implementar este hook en tus desarrollos cuando lo consideres necesario.

Te veo en el siguiente capítulo.

Hook para manejar la API de Context useContext

El hook useContext nos sirve para poder implementar la API de Context que ya existía en React desde antes de los hooks.

Ejemplo de useContext de reactjs.org

```
const value = useContext(MyContext);
```

Para quien no esté familiarizado, Context nos permite comunicar props en un árbol de componentes sin necesidad de pasarlos manualmente a través de props.

Algunos ejemplos de cuándo utilizar Context son:

- Un Tema de la UI (light theme, dark theme, etc).
- Autenticación del usuario.
- Idioma preferido.

Pero no está limitado a los ejemplos anteriores. Puedes aplicar Context a las necesidades de tu aplicación según tu criterio.

En otras palabras, no es obligatorio que Context maneje un "estado global" de la aplicación: Puede abarcar sólo una parte del estado si lo deseas.

A continuación vamos a ver ejemplos y comparaciones de cómo utilizar este hook para que puedas aprovechar todo su potencial.

Ejemplo de Context en componente clase y useContext en componente funcional

Primero implementaremos un Context para aplicar un tema en componentes usando la API tradicional y luego lo refactorizaremos usando useContext.

Esto es sólo para poder identificar las semejanzas y diferencias entre ambas opciones. No quiere decir que la manera tradicional sea obsoleta ni nada por el estilo.

Dicho lo anterior, comenzaremos creando un Context para manejar el tema del siguiente modo:

```
// theme-context.js

export const themes = {
  light: {
    color: "#555555",
    background: "#eeeeee"
  },
  dark: {
    color: "#eeeeee",
    background: "#222222"
  },
  vaporwave: {
    color: "#ffffff",
    background: "#ff71ce"
  }
};

export const ThemeContext = React.createContext(themes.light);
```

En este caso tenemos un objeto para representar temas diferentes que podemos usar en nuestra UI.

El tema por defecto es themes.light que es pasado por parámetro a createContext.

Ahora necesitamos proveer este context a nuestro árbol de componentes:

Proveer Context

```
export default function App() {
  const [currentTheme, setCurrentTheme] =
    useState(themes.light);

  return (
```

```
    <div className="App">
      <h1>React Context</h1>

      <ThemeContext.Provider value={currentTheme}>
        <MyButton>Hello World!</MyButton>
      </ThemeContext.Provider>
    </div>
  );
}
```

Estamos creando una variable de estado con `useState` para obtener el valor actual del tema.

Usamos `ThemeContext.Provider value={currentTheme}`para asignar el context y a partir de aquí, todo el árbol de componentes hijos van a poder tener acceso a context si lo desean.

Para usarlo en un componente lo hacemos del siguiente modo:

Consumir Context

```
class MyButton extends Component {
  render() {
    const theme = this.context;
    const style = {
      backgroundColor: theme.background,
      color: theme.color,
      border: "1px solid",
      borderRadius: 5
    };
    return <button style={style} {...this.props} />;
  }
}

// Esto es importante (asignar el context)
MyButton.contextType = ThemeContext;
```

Y con esto ya tenemos nuestro context funcionando.

Ahora si queremos modificar el context, en este ejemplo basta con hacer lo que sigue:

modificar context

```
export default function App() {
  const [currentTheme, setCurrentTheme] = useState(themes.light);

  return (
    <div className="App">
      <h1>React Context</h1>
      <ThemeContext.Provider value={currentTheme}>
        <MyButton
          onClick={() => setCurrentTheme(themes.dark)}
        >
          Dark Theme
        </MyButton>
        <MyButton
          onClick={() => setCurrentTheme(themes.vaporwave)}
        >
          Vaporwave Theme
        </MyButton>
      </ThemeContext.Provider>
    </div>
  );
}
```

Con esto ya funciona pero por lo común vas a tener la necesidad de modificar el context desde un componente hijo.

Para eso podemos modificar nuestro context para proveer su valor y además una función que sirva para modificar su valor.

Primero modificamos nuestro context:

```
export const ThemeContext = React.createContext({
  theme: themes.light,
  updateTheme: () => {}
```

```
});
```

Ahora tendrá un objeto con las dos propiedades antes dichas.

En donde hacemos uso de Provider necesitamos modificar también su valor:

```
export default function App() {
  const [currentTheme, setCurrentTheme] = useState(themes.light);

  return (
    <div className="App">
      <h1>React Context</h1>

      {/* Nuevo value definido */}
      <ThemeContext.Provider
        value={{
          theme: currentTheme,
          updateTheme: setCurrentTheme
        }}
      >
        <MyButton
          onClick={() => setCurrentTheme(themes.dark)}
        >
          Dark Theme
        </MyButton>

        <MyButton
          onClick={() => setCurrentTheme(themes.vaporwave)}
        >
          Vaporwave Theme
        </MyButton>

        {/* Nuevo Boton sin onClick */}
        <MyButton>Light Theme</MyButton>
      </ThemeContext.Provider>
```

```
    </div>
  );
}
```

En el componente MyButton vamos a hacer que si no recibe el prop de onClick, pondremos por default updateThemeque obtenemos de ThemeContext:

```
class MyButton extends Component {
  render() {
    const { theme, updateTheme } = this.context;
    const style = {
      backgroundColor: theme.background,
      color: theme.color,
      border: "1px solid",
      borderRadius: 5
    };

    const updateLightTheme = () => {
      updateTheme(themes.light);
    };

    const onClick = this.props.handleClick ||
      updateLightTheme;

    return <button
      onClick={onClick}
      style={style}
      {...this.props}
    />;
  }
}
```

Con esto ya estamos actualizando context desde un componente hijo o de manera directa en el componente donde hacemos el provider.

Ahora vamos a hacer una nueva versión del código anterior para hacer uso de useContext y puedas apreciar las diferencias.

La definición de ThemeContext no necesitamos modificarla, ni el Provider.

Lo que nos va a servir useContext es para poder acceder al valor de un context en un componente funcional.

Veamos el refactor:

Ejemplo de useContext

```
const MyNewButton = (props) => {
  const { theme, updateTheme } = useContext(ThemeContext);

  const style = {
    backgroundColor: theme.background,
    color: theme.color,
    border: "1px solid",
    borderRadius: 5
  };

  const updateLightTheme = () => {
    updateTheme(themes.light);
  };

  const onClick = props.handleClick || updateLightTheme;

  return <button onClick={onClick} {...props} style={style} />;
};
```

Como puedes ver, lo único que cambia es que ahora podemos obtener nuestro valor de context usando const { theme, updateTheme } = useContext(ThemeContext);.

Con esto cubrimos el tema de Context en sí mismo y el hook useContext.

Recapitulación del hook useContext

En este capítulo hemos visto:

- Cómo es el uso básico api de Context.
- Algunos casos de uso en los cuáles puedes aplicar Context.
- Una manera de aplicar Context en componentes de tipo clase.
- La manera de aplicar Context con el hook useContext.

Espero que este capítulo te haya sido de utilidad.

Hook para trabajar con referencias useRef

El hook useRef nos permite trabajar con referencias en componentes funcionales.

Ejemplo de useRef de reactjs.org

```
const refContainer = useRef(initialValue);
```

Este hook nos retorna un objeto con una propiedad current que es mutable y cuyo valor persiste durante los renderizados y ciclo de vida del componente.

Un caso de uso es usar referencias al DOM o a componentes de React.

Por ejemplo, cuando queremos hacer que un input tenga un auto focus al montarse.

Primero vamos a hacer el ejemplo con un componente de tipo clase y posteriormente la versión con useRef.

Ejemplo de autofocus en componente clase

```
class MyForm extends Component {
  inputEl = null;

  componentDidMount() {
    this.inputEl.focus();
  }

  render() {
    return (
      <p>
        Name:
        <input
          type="text"
          ref={(el) => (this.inputEl = el)}
        />
```

```
    </p>
  );
 }
}
```

Usando useRef queda del siguiente modo:

Ejemplo de auto focus con useRef

```
const Form = () => {
 const inputEl = useRef(null);

 useEffect(() => inputEl.current.focus(), []);

 return (
  <p>
   Name:
   <input type="text" ref={inputEl} />
  </p>
 );
};
```

Puedes ejecutar y editar el este código en este enlace.

Otra característica importante de ref es que si su valor cambia, no causa un re render.

Emular variables de instancia en componente funcional (this)

Dentro de un componente de tipo clase podemos crear variables de instancia como el siguiente ejemplo:

```
class MyComponent extends React.Component {
 intervalId = null;

 componentDidMount() {
  // Accedemos con "this"
```

```
    this.intervalId = setInterval(() => {
      // ...
    });
  }

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

  render () {
    // code
  }
}
```

El hook `useRef` también nos sirve para tener referencias a cualquier valor.

De este modo, podemos emular variables de instancia:

Ejemplo de reactjs.org

```
function Timer() {
  const intervalRef = useRef();

  useEffect(() => {
    const id = setInterval(() => {
      // ...
    });
    intervalRef.current = id;
    return () => {
      clearInterval(intervalRef.current);
    };
  });

  // ...
}
```

Recapitulación del hook useRef

Este ha sido un capítulo corto debido a la sencillez de este hook. Lo que vimos fue:

- Cómo usar ref en un componente de tipo clase.
- Cómo usar ref en un componente de tipo función usando `useRef` para elementos de la UI.
- Cómo usar `useRef` para crear valores que se comportan parecido a como funciona una variable de instancia de una clase.

Siguientes pasos en tu carrera de React Developer

¡Muchas felicidades por haber terminado este ebook! :D

Hemos aprendido cómo funcionan los principales hooks, los cómo, por qué y en qué casos podemos aplicarlos.

¿Te quedaste con ganas de más?

Este ebook cubre las bases iniciales para poder pasar a un siguiente nivel de **patrones avanzados en React** de los cuáles muchos de ellos son aplicados mediante hooks.

Si te gustaría pasar al siguiente nivel en tus conocimientos sobre React, te recomiendo que le des un ojo a este ebook que publiqué en Amazon:

- **Curso Curso: Guía definitiva: Aprende los 9 Patrones Avanzados en ReactJS**. Obtén tu cupón de descuento en este link.
- Entra al sitio web de Developero para poder seguir más contenidos como este :)

Por último, quiero pedirte un favor. Si tienes posibilidad de dejar tu review y valoración de este ebook, te lo voy a agradecer mucho :D