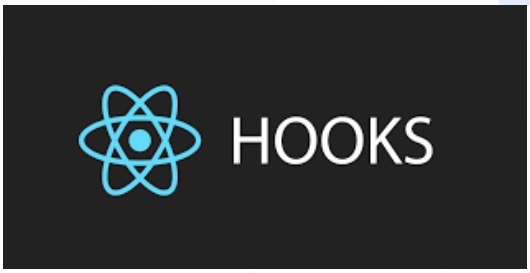

Ing. Luis Guillermo Molero Suárez

Presentando Hooks

Hooks son una nueva característica en React 16.8. Estos te permiten usar el estado y otras características de React sin escribir una clase.

```
import React, { useState } from 'react';

function Example() {
  // Declara una nueva variable de estado, la cual llamaremos
“count”  const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Esta nueva función useState es el primer “Hook” que vamos a aprender, pero este ejemplo es solo una introducción. ¡No te preocupes si aún no tiene sentido!
Puedes empezar a aprender Hooks en la siguiente página. En esta página, continuaremos explicando por qué estamos agregando Hooks a React y cómo estos pueden ayudarte a escribir aplicaciones grandiosas.

En el React Conf 2018, Sophie Alpert y Dan Abramov presentaron Hooks, seguidos por Ryan Florence demostrando cómo refactorizar una aplicación usándolos. Mira el video aquí:

Sin cambios con rupturas

Antes de continuar, debes notar que los Hooks son:

- **Completamente opcionales.** Puedes probar Hooks en unos pocos componentes sin reescribir ningún código existente. Pero no tienes que aprender o usar Hooks ahora mismo si no quieres.
- **100% compatibles con versiones anteriores.** Los Hooks no tienen cambios con rupturas con respecto a versiones existentes.
- **Disponibles de inmediato.** Los Hooks ya están disponibles con el lanzamiento de la versión v16.8.0.

Motivación

Los Hooks resuelven una amplia variedad de problemas aparentemente desconectados en React que hemos encontrado durante más de cinco años de escribir y mantener decenas de miles de componentes. Ya sea que estés aprendiendo React, usándolo diariamente o incluso prefieras una librería diferente con un modelo de componentes similar, es posible que reconozcas algunos de estos problemas.

Es difícil reutilizar la lógica de estado entre componentes

React no ofrece una forma de "acoplar" comportamientos re-utilizables a un componente (Por ejemplo, al conectarse a un *store*). Si llevas un tiempo trabajando con React, puedes estar familiarizado con patrones como render props y componentes de orden superior que tratan resolver esto. Pero estos patrones requieren que reestructures tus componentes al usarlos, lo cual puede ser complicado y hacen que tu código sea más difícil de seguir. Si observas una aplicación típica de React usando *React DevTools*, Lo más probable es que encuentres un "wrapper hell" de componentes envueltos en capas de *providers*, *consumers*, *componentes de orden superior*, *render props*, y otras abstracciones. Aunque podemos filtrarlos usando las DevTools, esto apunta a un problema aún más profundo: React necesita una mejor primitiva para compartir lógica de estado.

Con Hooks, puedes extraer lógica de estado de un componente de tal forma que este pueda ser probado y re-usado independientemente. **Los Hooks te permiten reutilizar lógica de estado sin cambiar la jerarquía de tu componente.** Esto facilita el compartir Hooks entre muchos componentes o incluso con la comunidad.

A menudo tenemos que mantener componentes que empiezan simples pero con el pasar del tiempo crecen y se convierten en un lío inmanejable de multiples lógicas de estado y efectos secundarios. Cada método del ciclo de vida a menudo contiene una mezcla de lógica no relacionada entre sí. Por ejemplo, los componentes pueden realizar alguna

consulta de datos en el componentDidMount y componentDidUpdate. Sin embargo, el mismo método componentDidMount también puede contener lógica no relacionada que cree escuchadores de eventos, y los limpie en el componentWillUnmount. El código relacionado entre sí y que cambia a la vez es separado, pero el código que no tiene nada que ver termina combinado en un solo método. Esto hace que sea demasiado fácil introducir errores e inconsistencias.

En muchos casos no es posible dividir estos componentes en otros más pequeños porque la lógica de estado está por todas partes. También es difícil probarlos. Esta es una de las razones por las que muchas personas prefieren combinar React con una librería de administración de estado separada. Sin embargo, esto a menudo introduce demasiada abstracción, requiere que saltes entre diferentes archivos, y hace que la reutilización de componentes sea más difícil.

Para resolver esto, **Hooks te permite dividir un componente en funciones más pequeñas basadas en las piezas relacionadas (como la configuración de una suscripción o la consulta de datos)**, en lugar de forzar una división basada en los métodos del ciclo de vida. También puedes optar por administrar el estado local del componente con un *reducer* para hacerlo más predecible.

Además de dificultar la reutilización y organización del código, hemos descubierto que las clases pueden ser una gran barrera para el aprendizaje de React. Tienes que entender cómo funciona this en JavaScript, que es muy diferente a cómo funciona en la mayoría de los lenguajes. Tienes que recordar agregar *bind* a tus manejadores de eventos. Sin inestables propuestas de sintaxis, el código es muy verboso. Las personas pueden entender *props*, el estado, y el flujo de datos de arriba hacia abajo perfectamente, pero todavía tiene dificultades con las clases. La distinción entre componentes de función y de clase en React y cuándo usar cada uno de ellos lleva a desacuerdos incluso entre los desarrolladores experimentados de React.

Además, React ha estado en el mercado durante unos cinco años, y queremos asegurarnos de que siga siendo relevante en los próximos cinco años. Como muestran Svelte, Angular, Glimmer, y otros, la compilación anticipada de componentes tiene mucho potencial a futuro. Especialmente si no se limita a las plantillas. Recientemente, hemos estado experimentando con el encarpetado de componentes usando Prepack, y hemos visto resultados preliminares prometedores. Sin embargo, encontramos que los componentes de clase pueden fomentar patrones involuntarios que hacen que estas optimizaciones nos lleven a un camino más lento. Las clases también presentan problemas para las herramientas de hoy en día. Por ejemplo, las clases no minifican muy bien, y hacen que la recarga en caliente sea confusa y poco fiable. Queremos presentar una API que hace más probable que el código se mantenga en la ruta optimizable.

Para resolver estos problemas, **Hooks te permiten usar más de las funciones de React sin clases.** Conceptualmente, los componentes de React siempre han estado más cerca de las funciones. Los Hooks abarcan funciones, pero sin sacrificar el espíritu práctico de React. Los Hooks proporcionan acceso a vías de escape imprescindibles y no requieren que aprendas técnicas complejas de programación funcional o reactiva.

Hook de estado

Este ejemplo renderiza un contador. Cuando haces click en el botón, incrementa el valor:

```
import React, { useState } from 'react';
function Example() {
  // Declara una nueva variable de estado, que llamaremos "count".
const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Aquí, useState es un *Hook* (hablaremos de lo que esto significa en un momento). Lo llamamos dentro de un componente de función para agregarle un estado local. React mantendrá este estado entre re-renderizados. useState devuelve un par: el valor de estado *actual* y una función que le permite actualizarlo. Puedes llamar a esta función desde un controlador de eventos o desde otro lugar. Es similar a this.setState en una clase, excepto que no combina el estado antiguo y el nuevo. (Mostraremos un ejemplo comparando useState con this.state en Usando el Hook de estado).

El único argumento para useState es el estado inicial. En el ejemplo anterior, es 0 porque nuestro contador comienza desde cero. Ten en cuenta que a diferencia de this.state, el estado aquí no tiene que ser un objeto — aunque puede serlo si quisieras. El argumento de estado inicial solo se usa durante el primer renderizado.

Declarando múltiples variables de estado

Puedes usar el Hook de estado más de una vez en un mismo componente:

```
function ExampleWithManyStates() {
  // Declarar múltiple variables de estado!
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
  // ...
}
```

La sintaxis de desestructuración de un array nos permite dar diferentes nombres a las variables de estado que declaramos llamando a useState. Estos nombres no forman parte de la API useState. En su lugar, React asume que si llamas a useState muchas veces, lo haces en el mismo orden durante cada renderizado. Volveremos a explicar por qué esto funciona y cuándo será útil más adelante.

¿Pero qué es un Hook?

Los Hooks son funciones que te permiten “enganchar” el estado de React y el ciclo de vida desde componentes de función. Los hooks no funcionan dentro de las clases — te permiten usar React sin clases. (No recomendamos reescribir tus componentes existentes de la noche a la mañana, pero puedes comenzar a usar Hooks en los nuevos si quieres). React proporciona algunos Hooks incorporados como useState. También puedes crear tus propios Hooks para reutilizar el comportamiento con estado entre diferentes componentes. Primero veremos los Hooks incorporados.

Hook de efecto {#effect-hook}

Es probable que hayas realizado recuperación de datos, suscripciones o modificación manual del DOM desde los componentes de React. Llamamos a estas operaciones “efectos secundarios” (o “efectos” para abreviar) porque pueden afectar a otros componentes y no se pueden hacer durante el renderizado.

El Hook de efecto, useEffect, agrega la capacidad de realizar efectos secundarios desde un componente de función. Tiene el mismo propósito que componentDidMount, componentDidUpdate y componentWillUnmount en las clases React, pero unificadas en una sola API. (Mostraremos ejemplos comparando useEffect con estos métodos en Usando el Hook de efecto).

Por ejemplo, este componente establece el título del documento después de que React actualiza el DOM:

```
import React, { useState, useEffect } from 'react';
function Example() {
  const [count, setCount] = useState(0);

  // Similar a componentDidMount y componentDidUpdate:
useEffect(() => {    // Actualiza el título del documento usando
la Browser API    document.title = `You clicked ${count} times`;
});
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Cuando llamas a useEffect, le estás diciendo a React que ejecute tu función de “efecto” después de vaciar los cambios en el DOM. Los efectos se declaran dentro del componente para que tengan acceso a sus props y estado. De forma predeterminada, React ejecuta los efectos después de cada renderizado — *incluyendo* el primer renderizado. (Hablaremos más sobre cómo se compara esto con los ciclos de vida de una clase en Usando el Hook de efecto).

Los efectos también pueden especificar opcionalmente cómo “limpiar” después de ellos devolviendo una función. Por ejemplo, este componente utiliza un efecto para suscribirse al estado en línea de un amigo, y se limpia al anular su suscripción:

```
import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
ChatAPI.subscribeToFriendStatus(props.friend.id,
```

```
handleStatusChange);    return () => {
ChatAPI.unsubscribeFromFriendStatus(props.friend.id,
handleStatusChange);    };  });
  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

En este ejemplo, React cancelará la suscripción de nuestra ChatAPI cuando se desmonte el componente, así como antes de volver a ejecutar el efecto debido a un renderizado posterior. (Si prefieres, hay una manera de decirle a React que omita la re-suscripcion si el props.friend.id que pasamos a la ChatAPI no ha cambiado).

Al igual que con useState, puedes usar más de un solo efecto en un componente:

```
function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  useEffect(() => {    document.title = `You clicked ${count}
times`;
  });

  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
ChatAPI.subscribeToFriendStatus(props.friend.id,
handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id,
handleStatusChange);
    };
  });

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }
  // ...
```

Los Hooks te permiten organizar efectos secundarios en un componente según qué partes están relacionadas (como agregar y eliminar una suscripción), en lugar de forzar una división basada en métodos del ciclo de vida.

Reglas de Hooks {#rules-of-hooks}

Los Hooks son funciones de JavaScript, pero imponen dos reglas adicionales:

- Solo llamar Hooks **en el nivel superior**. No llames Hooks dentro de loops, condiciones o funciones anidadas.
- Solo llamar Hooks **desde componentes de función de React**. No llames Hooks desde las funciones regulares de JavaScript. (Solo hay otro lugar válido para llamar Hooks: tus propios Hooks personalizados. En un momento aprenderemos sobre estos).

Proporcionamos un plugin de linter para forzar estas reglas automáticamente. Entendemos que estas reglas pueden parecer limitantes o confusas al principio, pero son esenciales para hacer que los Hooks funcionen bien.

Construyendo tus propios Hooks

A veces, queremos reutilizar alguna lógica de estado entre componentes. Tradicionalmente, había dos soluciones populares para este problema: componente de orden superior y render props. Los Hooks personalizados te permiten hacer esto, pero sin agregar más componentes a tu árbol.

Anteriormente en esta página, presentamos un componente FriendStatus que llama a los Hooks useState y useEffect para suscribirse al estado en línea de un amigo. Digamos que también queremos reutilizar esta lógica de suscripción en otro componente.

Primero, extraeremos esta lógica en un Hook personalizado llamado useFriendStatus:

```
import React, { useState, useEffect } from 'react';

function useFriendStatus(friendID) {  const [isOnline,
setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID,
handleStatusChange);
    };
  });

  return isOnline;
```

```
}
```

Toma friendID como argumento, y retorna si nuestro amigo está en línea o no.

Ahora lo podemos usar desde ambos componentes:

```
function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);
  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
function FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);
  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

El estado de cada componente es completamente independiente. Los Hooks son una forma de reutilizar *la lógica de estado*, no el estado en sí. De hecho, cada *llamada* a un Hook tiene un estado completamente aislado — por lo que incluso puedes usar el mismo Hook personalizado dos veces en un componente.

Los Hooks personalizados son más una convención que una funcionalidad. Si el nombre de una función comienza con ”use” y llama a otros Hooks, decimos que es un Hook personalizado. La convención de nomenclatura useSomething es cómo nuestro plugin de linter puede encontrar errores en el código usando Hooks.

Puedes escribir Hooks personalizados que cubran una amplia gama de casos de uso como manejo de formularios, animación, suscripciones declarativas, temporizadores y probablemente muchos más que no hemos considerado. Estamos muy entusiasmados de ver los Hooks personalizados que la comunidad de React creará.

Otros Hooks

Hay algunos Hooks incorporados de uso menos común que pueden resultarte útiles. Por ejemplo, useContext te permite suscribirte al contexto React sin introducir el anidamiento:

```
function Example() {
  const locale = useContext(LocaleContext);  const theme =
useContext(ThemeContext);  // ...
}
```

Y useReducer te permite manejar el estado local de componentes complejos con un reducer:

```
function Todos() {
  const [todos, dispatch] = useReducer(todosReducer);  // ...
```

Usando el Hook de estado

Los *Hooks* son una nueva incorporación en React 16.8. Te permiten usar estado y otras características de React sin escribir una clase.

La página de introducción utilizó este ejemplo para familiarizarte con los Hooks:

```
import React, { useState } from 'react';

function Example() {
  // Declaración de una variable de estado que llamaremos "count"
const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Empezaremos aprendiendo sobre los Hooks comparando este código con uno equivalente en una clase.

Ejemplo equivalente en forma de clase

Si has usado clases en React previamente, este código te resultará familiar:

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count:
this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

El estado empieza como { count:0 } y se incrementa state.count cuando el usuario hace click a un botón llamando a this.setState(). Usaremos fragmentos de esta clase en toda la página.

Hooks y componentes de función

Como recordatorio, un componente de función en React se ve así:

```
const Example = (props) => {
  // Puedes usar Hooks aquí!
  return <div />;
}
```

o así:

```
function Example(props) {
  // Puedes usar Hooks aquí!
  return <div />;
}
```

Puedes haber conocido previamente estos componentes como “componentes sin estado”. Actualmente estamos presentando la habilidad de usar el estado de React desde ellos por lo que preferimos el nombre “componentes de función”.

Los Hooks **no** funcionan en clases, pero los puedes usar en lugar de escribir clases.

¿Qué es un Hook?

Nuestro nuevo ejemplo empieza importando el Hook useState desde React:

```
import React, { useState } from 'react';
function Example() {
  // ...
}
```

¿Qué es un Hook? Un Hook es una función especial que permite “conectarse” a características de React. Por ejemplo, useState es un Hook que te permite añadir el estado de React a un componente de función. Más adelante hablaremos sobre otros Hooks.
¿Cuándo debería usar un Hook? Si creas un componente de función y descubres que necesitas añadirle estado, antes había que crear una clase. Ahora puedes usar un Hook dentro de un componente de función existente. ¡Vamos a hacerlo ahora mismo!

Declarando una variable de estado

En una clase, inicializamos el estado count a 0 estableciendo this.state a { count: 0 } en el constructor:

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {      count: 0    };  }
```

En un componente de función no existe this por lo que no podemos asignar o leer this.state. En su lugar, usamos el Hook useState directamente dentro de nuestro componente:

```
import React, { useState } from 'react';

function Example() {
  // Declaración de una variable de estado que llamaremos "count"
const [count, setCount] = useState(0);
```

¿Qué hace la llamada a useState**?** Declara una "variable de estado". Nuestra variable se llama count, pero podemos llamarla como queramos, por ejemplo banana. Esta es una forma de "preservar" algunos valores entre las llamadas de la función - useState es una nueva forma de usar exactamente las mismas funciones que this.state nos da en una clase. Normalmente, las variables "desaparecen" cuando se sale de la función, pero las variables de estado son conservadas por React.

¿Qué pasamos a useState **como argumento?** El único argumento para el Hook useState() es el estado inicial. Al contrario que en las clases, el estado no tiene por qué ser un objeto. Podemos usar números o strings si es todo lo que necesitamos. En nuestro ejemplo, solamente queremos un número para contar el número de clicks del usuario, por eso pasamos 0 como estado inicial a nuestra variable. (Si queremos guardar dos valores distintos en el estado, llamaríamos a useState() dos veces).

¿Qué devuelve useState**?** Devuelve una pareja de valores: el estado actual y una función que lo actualiza. Por eso escribimos const [count, setCount] = useState(). Esto es similar a this.state.count y this.setState en una clase, excepto que se obtienen juntos. Si no conoces la sintaxis que hemos usado, volveremos a ella al final de esta página.
Ahora que sabemos qué hace el Hook useState, nuestro ejemplo debería tener más sentido:

```
import React, { useState } from 'react';

function Example() {
  // Declaración de una variable de estado que llamaremos "count"
const [count, setCount] = useState(0);
```

Declaramos una variable de estado llamada count y le asignamos a 0. React recordará su valor actual entre re-renderizados, y devolverá el valor más reciente a nuestra función. Si se quiere actualizar el valor de count actual, podemos llamar a setCount

Leyendo el estado

Cuando queremos mostrar el valor actual de count en una clase lo obtenemos de this.state.count:

```
  <p>You clicked {this.state.count} times</p>
```

En una función podemos usar count directamente:

```
  <p>You clicked {count} times</p>
```

Actualizando el estado

En una clase, necesitamos llamar a this.setState() para actualizar el estado count:

```
  <button onClick={() => this.setState({ count: this.state.count +
1 })}>    Click me
  </button>
```

En una función ya tenemos setCount y count como variables, así que no necesitamos this:

```
  <button onClick={() => setCount(count + 1)}>    Click me
  </button>
```

Resumen

Ahora **recapitularemos lo que hemos aprendido línea por línea** y comprobaremos si lo hemos entendido.

```
1:  import React, { useState } from 'react'; 2:
3:  function Example() {
4:    const [count, setCount] = useState(0); 5:
6:    return (
7:      <div>
8:        <p>You clicked {count} times</p>
9:        <button onClick={() => setCount(count + 1)}>10:
Click me
11:        </button>
12:      </div>
13:    );
14:  }
```

- **Línea 1:** Importamos el Hook useState desde React que nos permite mantener un estado local en un componente de función.
- **Línea 4:** Dentro del componente Example declaramos una nueva variable de estado llamando al Hook useState. Este nos devuelve un par de valores, a los que damos un nombre. Llamamos count a nuestra variable porque guarda el número de clicks en el botón. La inicializamos a cero pasando 0 como único argumento a useState. El segundo elemento retornado es una función que nos permite actualizar count, por lo que le llamamos setCount.
- **Línea 9:** Cuando el usuario hace click, llamamos a setCount con un nuevo valor. React actualizará entonces el componente Example pasándole el nuevo valor de count.

Esto puede parecer mucho para empezar. ¡No tengas prisa! Si te pierdes con esta explicación repasa el código de arriba y trata de leerlo de arriba hacia abajo. Prometemos que una vez trates de "olvidar" cómo funciona el estado en las clases y mires a este código con la mente despejada cobrará sentido.

Tip: ¿Qué significan los corchetes?

Habrás observado los corchetes cuando declaramos una variable de estado:

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

Los nombres de la izquierda no son parte de la API de React. Puedes nombrar tus variables de estado como quieras:

```
const [fruit, setFruit] = useState('banana');
```

Esta sintaxis de Javascript se llama "desestructuración de arrays". Significa que estamos creando dos variables fruit y setFruit, donde fruit se obtiene del primer valor devuelto por useState y setFruit es el segundo. Es equivalente a este código:

```
var fruitStateVariable = useState('banana'); // Returns a pair
var fruit = fruitStateVariable[0]; // First item in a pair
var setFruit = fruitStateVariable[1]; // Second item in a pair
```

Cuando declaramos una variable de estado con useState, devuelve un array con dos elementos. El primero es el valor actual y el segundo es una función que nos permite actualizarlo. Usar [0] y [1] para acceder a ello es un poco confuso porque tienen un significado específico. Por ello usamos la desestructuración de arrays en su lugar.

Tip: Usando múltiples variables de estado

Declarando variables de estado como un par [something, setSomething] también es útil porque nos permite dar *diferentes* nombres a diferentes variables de estados si queremos usar más de una:

```
function ExampleWithManyStates() {
  // Declare multiple state variables!
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
```

En el componente de arriba tenemos age, fruit, y todos como variables locales y los podemos actualizar de forma individual:

```
  function handleOrangeClick() {
    // Similar a this.setState({ fruit: 'orange' })
    setFruit('orange');
  }
```

No tienes que usar obligatoriamente tantas variables de estado: las variables de estado pueden contener objetos y arrays para que puedas agrupar la información relacionada. Sin embargo, al contrario que en una clase, actualizar una variable de estado siempre la reemplaza en lugar de combinarla.

Usando el Hook de efecto

Los *Hooks* son una nueva incorporación en React 16.8. Te permiten usar estado y otras características de React sin escribir una clase.

El *Hook de efecto* te permite llevar a cabo efectos secundarios en componentes funcionales:

```
import React, { useState, useEffect } from 'react';
function Example() {
  const [count, setCount] = useState(0);

  // De forma similar a componentDidMount y componentDidUpdate
useEffect(() => {    // Actualiza el título del documento usando
la API del navegador    document.title = `You clicked ${count}
times`;  });
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Este fragmento está basado en el ejemplo de contador de la página anterior, pero le hemos añadido una funcionalidad nueva: actualizamos el título del documento con un mensaje personalizado que incluye el número de clicks.

Peticiones de datos, establecimiento de suscripciones y actualizaciones manuales del DOM en componentes de React serían ejemplos de efectos secundarios. Tanto si estás acostumbrado a llamar a estas operaciones “efectos secundarios” (o simplemente “efectos”) como si no, probablemente los has llevado a cabo en tus componentes con anterioridad.

Efectos sin saneamiento

En ciertas ocasiones, queremos **ejecutar código adicional después de que React haya actualizado el DOM.** Peticiones de red, mutaciones manuales del DOM y registros, son ejemplos comunes de efectos que no requieren una acción de saneamiento. Decimos esto porque podemos ejecutarlos y olvidarnos de ellos inmediatamente. Vamos a comparar cómo las clases y los *Hooks* nos permiten expresar dichos efectos.

Ejemplo con clases

En los componentes de React con clases, el método render no debería causar efectos secundarios por sí mismo. Sería prematuro. Normalmente queremos llevar a cabo nuestros efectos *después* de que React haya actualizado el DOM.

Y es por eso que en las clases de React ponemos los efectos secundarios en componentDidMount y componentDidUpdate. Volviendo a nuestro ejemplo, aquí tenemos el componente clase contador de React que actualiza el título del documento justo después de que React haga cambios en el DOM:

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  componentDidMount() {    document.title = `You clicked
${this.state.count} times`;  }  componentDidUpdate() {
document.title = `You clicked ${this.state.count} times`;  }
  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count:
this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

Fíjate en cómo **hemos duplicado el código en los dos métodos del ciclo de vida en la clase**
Esto es porque en muchas ocasiones queremos llevar a cabo el mismo efecto secundario sin importar si el componente acaba de montarse o si se ha actualizado.
Conceptualmente, queremos que ocurra después de cada renderizado, pero las clases de React no tienen un método que haga eso. Podríamos extraer un método, pero aún así tendríamos que llamarlo en los dos sitios.
Veamos ahora cómo podemos hacer lo mismo con el *Hook* useEffect.

Ejemplo con *Hooks*

Ya hemos visto este ejemplo al principio de la página, pero veámoslo más detenidamente:

```
import React, { useState, useEffect } from 'react';
function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {    document.title = `You clicked ${count}
times`;  });
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

¿Qué hace useEffect**?** Al usar este *Hook*, le estamos indicando a React que el componente tiene que hacer algo después de renderizarse. React recordará la función que le hemos pasado (nos referiremos a ella como nuestro "efecto"), y la llamará más tarde después de actualizar el DOM. En este efecto, actualizamos el título del documento, pero también podríamos hacer peticiones de datos o invocar alguna API imperativa.
¿Por qué se llama a useEffect **dentro del componente?** Poner useEffect dentro del componente nos permite acceder a la variable de estado count (o a cualquier prop) directamente desde el efecto. No necesitamos una API especial para acceder a ella, ya que se encuentra en el ámbito de la función. Los *Hooks* aprovechan los closures de JavaScript y evitan introducir APIs específicas de React donde JavaScript ya proporciona una solución.
¿Se ejecuta useEffect **después de cada renderizado?** ¡Sí! Por defecto se ejecuta después del primer renderizado *y* después de cada actualización. Más tarde explicaremos cómo modificar este comportamiento. En vez de pensar en términos de "montar" y "actualizar", puede resultarte más fácil pensar en efectos que ocurren "después del renderizado". React se asegura de que el DOM se ha actualizado antes de llevar a cabo el efecto.

Explicación detallada

Ahora que sabemos algo más sobre los efectos, estas líneas deberían cobrar sentido:

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

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });
}
```

Declaramos la variable de estado count y le indicamos a React que necesitamos usar un efecto. Le pasamos una función al *Hook* useEffect. Esta función que pasamos *es* nuestro efecto. Dentro de nuestro efecto actualizamos el título del documento usando la API del navegador document.title. Podemos leer el valor más reciente de count dentro del efecto porque se encuentra en el ámbito de nuestra función. Cuando React renderiza nuestro componente, recordará este efecto y lo ejecutará después de actualizar el DOM. Esto sucede en cada renderizado, incluyendo el primero.

Los desarrolladores experimentados en JavaScript se percatarán de que la función que le pasamos a useEffect es distinta en cada renderizado. Esto es intencionado. En realidad esto es lo que nos permite leer la variable count desde el interior de nuestro efecto sin preocuparnos de que su valor esté obsoleto. Cada vez que renderizamos, planificamos un *efecto* diferente, reemplazando el anterior. En cierta manera, esto hace que los efectos funcionen más como parte del resultado del renderizado. Cada efecto pertenece a su correspondiente renderizado. Más adelante aclararemos por qué esto es útil.

Efectos con saneamiento

En el apartado anterior hemos visto cómo expresar efectos secundarios que no necesitan ningún saneamiento. Sin embargo, algunos efectos lo necesitan. Por ejemplo, **si queremos establecer una suscripción** a alguna fuente de datos externa. En ese caso, ¡es importante sanear el efecto para no introducir una fuga de memoria! Comparemos cómo se puede hacer esto con clases y con *Hooks*.

Ejemplo con clases

En una clase de React, normalmente se establece una suscripción en componentDidMount, y se cancela la suscripción en componentWillUnmount. Por ejemplo, digamos que tenemos un módulo ChatAPI que nos permite suscribirnos para saber si un amigo está conectado. Así es como podemos establecer la suscripción y mostrar ese estado usando una clase:

```
class FriendStatus extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {    ChatAPI.subscribeToFriendStatus(
this.props.friend.id,      this.handleStatusChange    );  }
componentWillUnmount() {    ChatAPI.unsubscribeFromFriendStatus(
this.props.friend.id,      this.handleStatusChange    );  }
handleStatusChange(status) {    this.setState({      isOnline:
status.isOnline    });  }
  render() {
    if (this.state.isOnline === null) {
      return 'Loading...';
    }
    return this.state.isOnline ? 'Online' : 'Offline';
  }
}
```

Fíjate en cómo componentDidMount y componentWillUnmount necesitan ser un reflejo el uno del otro. Los métodos del ciclo de vida nos obligan a separar esta lógica incluso cuando, conceptualmente, el código de ambos está relacionado con el mismo efecto.

Nota

Los lectores perspicaces podrán percatarse de que este ejemplo necesita también un método componentDidUpdate para ser completamente correcto. De momento vamos a ignorar este hecho, pero volveremos a él en una sección posterior de esta página.

Ejemplo usando *Hooks*

Veamos cómo podemos escribir este componente con Hooks.
Quizás puedas estar pensando que necesitaríamos un efecto aparte para llevar a cabo este saneamiento. Pero el código para añadir y eliminar una suscripción está tan estrechamente relacionado que useEffect está diseñado para mantenerlo unido. Si tu efecto devuelve una función, React la ejecutará en el momento de sanear el efecto:

```
import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {    function handleStatusChange(status) {
setIsOnline(status.isOnline);    }
ChatAPI.subscribeToFriendStatus(props.friend.id,
handleStatusChange);    // Especifica cómo sanear este efecto:
return function cleanup() {
ChatAPI.unsubscribeFromFriendStatus(props.friend.id,
handleStatusChange);    };  });
  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

¿Por qué hemos devuelto una función en nuestro efecto? Este es un mecanismo opcional de los efectos. Todos los efectos pueden devolver una función que los sanea más tarde. Esto nos permite mantener la lógica de añadir y eliminar suscripciones cerca la una de la otra. ¡Son parte del mismo efecto!
¿Cuándo sanea React el efecto exactamente? React sanea el efecto cuando el componente se desmonta. Sin embargo, como hemos aprendido anteriormente, los efectos no se ejecutan solo una vez, sino en cada renderizado. He aquí el motivo por el cual React *también* sanea los efectos de renderizados anteriores antes de ejecutar los efectos del renderizado actual. Más adelante analizaremos por qué esto ayuda a evitar errores y cómo omitir este funcionamiento en el caso de que provoque problemas de rendimiento.

Nota

No tenemos que nombrar la función devuelta por el efecto. La hemos llamado cleanup esta vez para clarificar su propósito, pero podemos devolver una función flecha o nombrarla de otra forma.

Recapitulación

Hemos aprendido que useEffect nos permite expresar diferentes tipos de efectos secundarios después de que un componente se renderice. Algunos efectos pueden devolver una función cuando requieran saneamiento:

```
  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id,
handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id,
handleStatusChange);
    };
  });
```

Otros efectos pueden no tener fase de saneamiento y no devolver nada.

```
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });
```

El *Hook* de efecto unifica ambos casos en una única API.

Consejos para usar efectos

Vamos a continuar profundizando en algunos aspectos de useEffect que les resultarán curiosos de alguna forma a los usuarios de React experimentados. No te sientas obligado a indagar en ello ahora mismo. Siempre puedes volver a esta página para conocer más detalles del *Hook* de efecto.

Consejo: Usa varios efectos para separar conceptos

Uno de los problemas que esbozamos en la Motivación para crear los *Hooks* es que los métodos del ciclo de vida de las clases suelen contener lógica que no está relacionada, pero la que lo está, se fragmenta en varios métodos. Este es un componente que combina la lógica del contador y el indicador de estado del amigo de los ejemplos anteriores:

```
class FriendStatusWithCounter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0, isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  handleStatusChange(status) {
    this.setState({
      isOnline: status.isOnline
    });
  }
  // ...
```

Fíjate en como la lógica que asigna document.title se divide
entre componentDidMount y componentDidUpdate. La lógica de la suscripción también

se reparte entre componentDidMount y componentWillUnmount.
Y componentDidMount contiene código de ambas tareas.

Entonces, ¿cómo resuelven los *Hooks* este problema? Del mismo modo que puedes usar el *Hook de estado* más de una vez, puedes usar varios efectos. Esto nos permite separar la lógica que no está relacionada en diferentes efectos:

```
function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  useEffect(() => {    document.title = `You clicked ${count}
times`;
  });

  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id,
handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id,
handleStatusChange);
    };
  });
  // ...
}
```

Los *Hooks* nos permiten separar el código en función de lo que hace en vez de en función del nombre de un método de ciclo de vida. React aplicará *cada* efecto del componente en el orden en el que han sido especificados.

Explicación: Por qué los efectos se ejecutan en cada actualización

Si estás familiarizado con las clases, te preguntarás por qué la fase de saneamiento de efecto ocurre después de cada rerenderizado y no simplemente cuando el componente se desmonta. Veamos un ejemplo práctico para ver por qué este diseño nos ayuda a crear componentes con menos errores.

En apartados anteriores hemos presentado el ejemplo de un componente FriendStatus que muestra si un amigo está conectado o no. Nuestra clase lee friend.id de this.props, se suscribe al estado del amigo al montarse y cancela la suscripción al desmontarse.

```
  componentDidMount() {
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }
```

¿Pero qué sucede si la propiedad friend **cambia** mientras el componente está en la pantalla? Nuestro componente continuaría mostrando el estado de un amigo diferente. Esto es un error. Además podríamos causar una fuga de memoria o un fallo crítico al desmontar dado que la llamada que cancela la suscripción usaría un identificador erróneo. En un componente de clase, necesitaríamos añadir componentDidUpdate para manejar este caso:

```
  componentDidMount() {
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  componentDidUpdate(prevProps) {    // Cancela la suscripción del friend.id anterior    ChatAPI.unsubscribeFromFriendStatus( prevProps.friend.id,      this.handleStatusChange    );    // Se suscribe al siguiente friend.id ChatAPI.subscribeToFriendStatus(      this.props.friend.id, this.handleStatusChange    );  }
  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
```

```
  }
```

No gestionar componentDidUpdate correctamente es una fuente de errores común en las aplicaciones React.

Ahora consideremos la versión de este componente que usa *Hooks*:

```
function FriendStatus(props) {
  // ...
  useEffect(() => {
    // ...
    ChatAPI.subscribeToFriendStatus(props.friend.id,
handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id,
handleStatusChange);
    };
  });
```

No padece el mismo error. (Aunque tampoco hemos hecho ningún cambio)
No hay un código especial para gestionar las actualizaciones porque useEffect las gestiona *por defecto*. Sanea los efectos anteriores antes de aplicar los nuevos. Para ilustrar esto, esta es una secuencia de llamadas de suscripción y cancelación que produciría este componente a lo largo del tiempo:

```
// Se monta con las props { friend: { id: 100 } }
ChatAPI.subscribeToFriendStatus(100, handleStatusChange);     //
Ejecuta el primer efecto

// Se actualiza con las props { friend: { id: 200 } }
ChatAPI.unsubscribeFromFriendStatus(100, handleStatusChange); //
Sanea el efecto anterior
ChatAPI.subscribeToFriendStatus(200, handleStatusChange);     //
Ejecuta el siguiente efecto

// Se actualiza con las props { friend: { id: 300 } }
ChatAPI.unsubscribeFromFriendStatus(200, handleStatusChange); //
Sanea el efecto anterior
ChatAPI.subscribeToFriendStatus(300, handleStatusChange);     //
Ejecuta el siguiente efecto

// Se desmonta
```

```
ChatAPI.unsubscribeFromFriendStatus(300, handleStatusChange); //
Sanea el último efecto
```

Este comportamiento asegura la consistencia por defecto y previene errores que son comunes en los componentes de clase debido a la falta de lógica de actualización.

Consejo: Omite efectos para optimizar el rendimiento

En algunos casos, sanear o aplicar el efecto después de cada renderizado puede crear problemas de rendimiento. En los componentes de clase podemos solucionarlos escribiendo una comparación extra con prevProps o prevState dentro de componentDidUpdate:

```
componentDidUpdate(prevProps, prevState) {
  if (prevState.count !== this.state.count) {
    document.title = `You clicked ${this.state.count} times`;
  }
}
```

Este requerimiento es tan común que está incorporado en la API del *Hook* useEffect. Puedes indicarle a React que *omita* aplicar un efecto si ciertos valores no han cambiado entre renderizados. Para hacerlo, pasa un array como segundo argumento opcional a useEffect:

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]); // Solo se vuelve a ejecutar si count cambia
```

En el ejemplo anterior pasamos [count] como segundo argumento. ¿Qué significa esto? Si count es 5, y cuando nuestro componente se vuelve a renderizar count continúa siendo igual a 5, React comparará el [5] del renderizado anterior con el [5] del siguiente renderizado. Dado que todos los elementos en el array (5 === 5), React omitirá el efecto. Esa es nuestra optimización.

Cuando renderizamos con count actualizado a 6, React comparará los elementos en el array [5] del renderizado anterior con los elementos del array [6] del siguente renderizado. En esta ocasión, React volverá a aplicar el efecto dado que 5 !== 6. Si el array contiene varios elementos, React volverá a ejecutar el efecto si cualquiera de los elementos es diferente.

Esto también funciona para efectos que tienen fase de saneamiento:

```
useEffect(() => {
  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  ChatAPI.subscribeToFriendStatus(props.friend.id,
handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id,
handleStatusChange);
  };
}, [props.friend.id]); // Solo se vuelve a suscribir si la
propiedad props.friend.id cambia
```

En el futuro, el segundo argumento podría ser añadido automáticamente por una transformación en tiempo de compilación.

Reglas de los Hooks

Los *Hooks* son una nueva incorporación en React 16.8. Te permiten usar estado y otras características de React sin escribir una clase.
Los Hooks son funciones de JavaScript, pero necesitas seguir dos reglas cuando los uses. Proporcionamos un plugin de linter para hacer cumplir estas reglas automáticamente.
Llama Hooks solo en el nivel superior

No llames Hooks dentro de ciclos, condicionales o funciones anidadas. En cambio, usa siempre Hooks en el nivel superior de tu función en React, antes de cualquier retorno prematuro. Siguiendo esta regla, te aseguras de que los hooks se llamen en el mismo orden cada vez que un componente se renderiza. Esto es lo que permite a React preservar correctamente el estado de los hooks entre multiples llamados a useState y useEffect. (Si eres curioso, vamos a explicar esto en detalle más abajo.)
Llama Hooks solo en funciones de React

No llames Hooks desde funciones JavaScript regulares. En vez de eso, puedes:

- Llama Hooks desde componentes funcionales de React.
- Llama Hooks desde Hooks personalizados.

Siguiendo esta regla, te aseguras de que toda la lógica del estado de un componente sea claramente visible desde tu código fuente.

Plugin de ESLint

Lanzamos un plugin de ESLint llamado eslint-plugin-react-hooks que refuerza estas dos reglas. Puedes añadir este plugin a tu proyecto si quieres probarlo:

Este plugin es incluido por defecto en Create React App.

```
npm install eslint-plugin-react-hooks --save-dev
// Tu configuración de ESLint
{
  "plugins": [
    // ...
    "react-hooks"
  ],
  "rules": {
    // ...
    "react-hooks/rules-of-hooks": "error", // Checks rules of
Hooks
    "react-hooks/exhaustive-deps": "warn" // Checks effect
dependencies
  }
}
```

Puedes pasar a la siguiente página donde explicamos cómo escribir tus propios Hooks **ahora mismo.** En esta página, vamos a continuar explicando el razonamiento detrás de estas reglas.

Explicación

Como aprendimos anteriormente, podemos usar múltiples Hooks de Estado o Hooks de Efecto en un solo componente:

```
function Form() {
  // 1. Usa la variable de estado del nombre
  const [name, setName] = useState('Mary');

  // 2. Usa un efecto para persistir el formulario
  useEffect(function persistForm() {
    localStorage.setItem('formData', name);
  });

  // 3. Usa la variable de estado del apellido
  const [surname, setSurname] = useState('Poppins');
```

```
  // 4. Usa un efecto para la actualización del título
  useEffect(function updateTitle() {
    document.title = name + ' ' + surname;
  });

  // ...
}
```

Entonces, ¿cómo hace React para saber cuál estado corresponde a cuál llamado del useState? La respuesta es que **React se basa en el orden en el cual los Hooks son llamados**. Nuestro ejemplo funciona porque el orden en los llamados de los Hooks son el mismo en cada render:

```
// ------------
// Primer render
// ------------
useState('Mary')           // 1. Inicializa la variable de estado
del nombre con 'Mary'
useEffect(persistForm)     // 2. Agrega un efecto para persistir
el formulario
useState('Poppins')        // 3. Inicializa la variable de estado
del apellido con 'Poppins'
useEffect(updateTitle)     // 4. Agrega un efecto para la
actualización del título

// -------------
// Segundo render
// -------------
useState('Mary')           // 1. Lee la variable de estado del
nombre (el argumento es ignorado)
useEffect(persistForm)     // 2. Reemplaza el efecto para
persistir el formulario
useState('Poppins')        // 3. Lee la variable de estado del
apellido (el argumento es ignorado)
useEffect(updateTitle)     // 4. Reemplaza el efecto de
actualización del título

// ...
```

Siempre y cuando el orden de los llamados a los Hooks sean los mismos entre renders, React puede asociar algún estado local con cada uno de ellos. ¿Pero qué pasa si ponemos la llamada a un Hook (por ejemplo, el efecto persistForm) dentro de una condición?

```
  // 🔴 Estamos rompiendo la primera regla al usar un Hook en una
condición
  if (name !== '') {
    useEffect(function persistForm() {
      localStorage.setItem('formData', name);
    });
  }
```

La condición name !== '' es true en el primer render, entonces corremos el Hook. Sin embargo, en el siguiente render el usuario puede borrar el formulario, haciendo la condición false. Ahora que nos saltamos este Hook durante el renderizado, el orden de las llamadas a los Hooks se vuelve diferente:

```
useState('Mary')           // 1. Lee la variable de estado del
nombre (el argumento es ignorado)
// useEffect(persistForm)  // 🔴 Este Hook fue saltado
useState('Poppins')        // 🔴 2 (pero era el 3). Falla la
lectura de la variable de estado del apellido
useEffect(updateTitle)     // 🔴 3 (pero era el 4). Falla el
reemplazo del efecto
```

React no sabría qué devolver para la segunda llamada del Hook useState. React esperaba que la segunda llamada al Hook en este componente correspondiera al efecto persistForm, igual que en el render anterior, pero ya no lo hace. A partir de este punto, cada siguiente llamada de un Hook después de la que nos saltamos también cambiaría de puesto por uno, lo que llevaría a la aparición de errores.

Es por esto que los Hooks deben ser utilizados en el nivel superior de nuestros componentes. Si queremos ejecutar un efecto condicionalmente, podemos poner esa condición *dentro* de nuestro Hook:

```
  useEffect(function persistForm() {
    // 👍 No vamos a romper la primera regla nunca más.
    if (name !== '') {
      localStorage.setItem('formData', name);
    }
  });
```