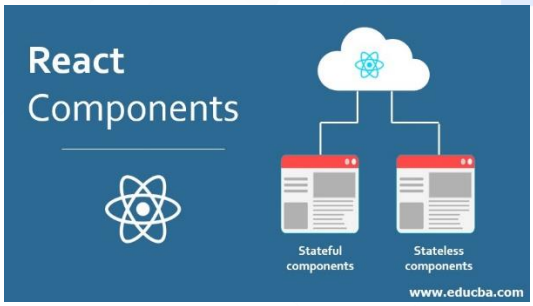

Ing. Luis Guillermo Molero Suárez

React.Component

Esta página contiene una referencia detallada de la API de React sobre los componentes definidos a través de clases. Asumimos que estas familiarizado con los conceptos fundamentales de React, como Componentes y props, así también Estado y ciclo de vida. Si no, léelas primero.

Resumen

React te permite definir componentes como clases o funciones. Los componentes definidos como clases actualmente proporcionan una serie de características extra que explicamos en esta página. Para definir una clase de componente React, necesitas extender React.Component:

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

El único método que *debes* definir en una subclase de React.Component es render(). Todos los demás métodos descritos en esta página son opcionales.

Recomendamos encarecidamente no crear tus propias clases de componentes base. En los componentes de React, la reutilización de código se consigue principalmente a través de la composición en lugar de la herencia.

Nota:

React no te obliga a usar la sintaxis de clases ES6. Si prefieres evitarlo, puede utilizar el módulo create-react-class o una abstracción personalizada similar en su lugar. Échale un vistazo a Usando React sin ES6 para aprender mas.

El ciclo de vida del componente

Cada componente tiene varios “métodos de ciclo de vida” que puedes sobrescribir para ejecutar código en momentos particulares del proceso. **Puedes usar** este diagrama de ciclo de vida **como una hoja de referencia.** En la lista de abajo, los métodos de ciclo de vida comúnmente usados están marcados en **negrita**. El resto de ellos existen para casos de uso relativamente raros.

Montaje

Estos métodos se llaman cuando se crea una instancia de un componente y se inserta en el DOM:

- constrºuctor()
- static getDerivedStateFromProps()
- render()
- componentDidMount()

Nota:

Estos métodos están considerados *legacy* (deprecados) y debes evitarlos en código nuevo:

- UNSAFE_componentWillMount()

Actualización

Una actualización puede ser causada por cambios en las props o el estado. Estos métodos se llaman en el siguiente orden cuando un componente se vuelve a renderizar:

- static getDerivedStateFromProps()
- shouldComponentUpdate()
- render()
- getSnapshotBeforeUpdate()
- componentDidUpdate()

Nota:

Estos métodos están considerados *legacy* (deprecados) y debes evitarlos en código nuevo:

- UNSAFE_componentWillUpdate()
- UNSAFE_componentWillReceiveProps()

Desmontaje

Este método es llamado cuando un componente se elimina del DOM:

- componentWillUnmount()

Manejo de errores

Estos métodos se invocan cuando hay un error durante la renderización, en un método en el ciclo de vida o en el constructor de cualquier componente hijo.

- static getDerivedStateFromError()
- componentDidCatch()

Otras APIs

Cada componente también proporciona algunas otras APIs:

- setState()
- forceUpdate()

Propiedades de clase

- defaultProps
- displayName

Propiedades de instancia

- props
- estado

Referencia

Funciones del ciclo de vida usadas comúnmente

Los métodos que verás en esta sección cubren la gran mayoría de casos de uso que encontrarás cuando crees componentes en React. **Para una referencia visual, revisa** este diagrama de los ciclos de vida**.**
render()

```
render()
```

El método render() es el único método requerido en un componente de clase.

Cuando se llama, debe examinar a this.props y this.state y devolver uno de los siguientes tipos:

- **Elementos de React.** normalmente creados a través de JSX. Por ejemplo, y <MyComponent /> son elementos de React que enseñan a React a renderizar un nodo DOM, u otro componente definido por el usuario, respectivamente.
- **Arrays y fragmentos.** Permiten que puedas devolver múltiples elementos desde el render. Consulta la documentación sobre fragmentos para más detalles.
- **Portales**. Te permiten renderizar hijos en otro subárbol del DOM. Consulta la documentación sobre portales para más detalles.
- **String y números.** Estos son renderizados como nodos de texto en el DOM.
- **Booleanos o** nulos. No renderizan nada. (Principalmente existe para admitir el patrón return test && <Child />, donde test es booleano.)

La función render () debe ser pura, lo que significa que no modifica el estado del componente, devuelve el mismo resultado cada vez que se invoca y no interactúa directamente con el navegador.
Si necesitas interactuar con el navegador, realiza tu trabajo en componentDidMount() o en los demás métodos de ciclo de vida. Mantener render() pure hace que los componentes sean más fáciles de considerar.

Nota

render() no será invocado si shouldComponentUpdate() devuelve falso.

constructor()

```
constructor(props)
```

Si no inicializas el estado y no enlazas los métodos, no necesitas implementar un constructor para tu componente React.

El constructor para un componente React es llamado antes de ser montado. Al implementar el constructor para una subclase React.Component, deberías llamar a super(props) antes que cualquier otra instrucción. De otra forma, this.props no estará definido en el constructor, lo que puede ocasionar a errores.

Normalmente, los constructores de React sólo se utilizan para dos propósitos:

- Para inicializar un estado local asignando un objeto al this.state.
- Para enlazar manejadores de eventos a una instancia.

No debes llamar setState() en el constructor(). En su lugar, si su componente necesita usar el estado local, **asigna directamente el estado inicial al** this.state directamente en el constructor:

```
constructor(props) {
  super(props);
  // No llames this.setState() aquí!
  this.state = { counter: 0 };
  this.handleClick = this.handleClick.bind(this);
}
```

El constructor es el único lugar donde debes asignar this.state directamente. En todos los demás métodos, debes usar this.setState() en su lugar.

Evita introducir cualquier efecto secundario o suscripciones en el constructor. Para estos casos de uso, use componentDidMount() en su lugar.

Nota

¡Evita copiar las props en el estado! Es un error muy común:

```
constructor(props) {
 super(props);
 // No hagas esto!
 this.state = { color: props.color };
}
```

El problema es que es innecesario (puedes usar this.props.color directamente en su lugar), esto crea errores (actualizaciones a la prop color no se reflejarán en el estado).

Sólo utiliza este patrón si deseas ignorar intencionalmente las actualizaciones de prop. En ese caso, tiene sentido renombrar la prop a initialColor o defaultColor. Puedes forzar al componente a "limpiar" su estado interno cambiando su key cuando sea necesario.

Lee nuestro post en el blog sobre como evitar estados derivados para aprender qué hacer si crees que necesitas algún estado que dependa de las props.

componentDidMount()

```
componentDidMount()
```

componentDidMount() se invoca inmediatamente después de que un componente se monte (se inserte en el árbol). La inicialización que requiere nodos DOM debería ir aquí. Si necesita cargar datos desde un punto final remoto, este es un buen lugar para instanciar la solicitud de red.

Este método es un buen lugar para establecer cualquier suscripción. Si lo haces, no olvides darle de baja en componentWillUnmount().

Puedes llamar setState() **inmediatamente** en componentDidMount(). Se activará un renderizado extra, pero sucederá antes de que el navegador actualice la pantalla. Esto garantiza que, aunque en este caso se invocará dos veces el render(), el usuario no verá el estado intermedio. Utiliza este patrón con precaución porque a menudo causa problemas de rendimiento. En la mayoría de los casos, deberías ser capaz de asignar el estado inicial en el constructor() en su lugar. Sin embargo, puede ser necesario para casos como modales y tooltips cuando se necesita medir un nodo DOM antes de representar algo que depende de su tamaño o posición.

componentDidUpdate()

```
componentDidUpdate(prevProps, prevState, snapshot)
```

componentDidUpdate() se invoca inmediatamente después de que la actualización ocurra. Este método no es llamado para el renderizador inicial.

Use esto como una oportunidad para operar en DOM cuando el componente se haya actualizado. Este es también un buen lugar para hacer solicitudes de red siempre y cuando compare los accesorios actuales con los anteriores (por ejemplo, una solicitud de red puede no ser necesaria si las props no han cambiado).

```
componentDidUpdate(prevProps) {
  // Uso tipico (no olvides de comparar las props):
  if (this.props.userID !== prevProps.userID) {
    this.fetchData(this.props.userID);
  }
}
```

Puedes llamar setState() **inmediatamente** en componentDidUpdate() pero ten en cuenta que **debe ser envuelto en una condición** como en el ejemplo anterior, o causará un bucle infinito. También causaría una renderización adicional que, aunque no sea visible para el usuario, puede afectar el rendimiento del componente. Si estás intentando crear un "espejo" desde un estado a una prop que viene desde arriba, considera usar la prop directamente en su lugar. Lee más sobre por qué copiar props en el estado causa errores. Si tu componente implementa el ciclo de vida getSnapshotBeforeUpdate()(que es raro), el valor que devuelve se pasará como un tercer parámetro "snapshot" a componentDidUpdate(). De lo contrario, este parámetro será indefinido.

Nota

componentDidUpdate() no será invocado si shouldComponentUpdate() devuelve falso.

componentWillUnmount()

```
componentWillUnmount()
```

componentWillUnmount() se invoca inmediatamente antes de desmontar y destruir un componente. Realiza las tareas de limpieza necesarias en este método, como la invalidación de temporizadores, la cancelación de solicitudes de red o la eliminación de las suscripciones que se crearon en componentDidMount().

No debes llamar setState() en componentWillUnmount() porque el componente nunca será vuelto a renderizar. Una vez que una instancia de componente sea desmontada, nunca será montada de nuevo.

Métodos de ciclo de vida raramente utilizados

Los métodos de esta sección corresponden a casos de uso poco común. Son útiles alguna vez, pero la mayoría de sus componentes probablemente no necesitan ninguno de ellos. **Puedes ver la mayoría de los métodos a continuación en** este diagrama de ciclo de vida **si haces clic en la casilla de verificación "Mostrar ciclos de vida menos comunes" en la parte superior de él.**

shouldComponentUpdate()

```
shouldComponentUpdate(nextProps, nextState)
```

Usa shouldComponentUpdate() para avisar a React si la salida de un componente no se ve afectada por el cambio actual en el estado o los accesorios. El comportamiento predeterminado es volver a procesar cada cambio de estado y, en la gran mayoría de los casos, debe confiar en el comportamiento predeterminado.

shouldComponentUpdate() es invocado antes de renderizar cuando los nuevos accesorios o el estado están siendo recibidos. Por defecto es true. Este método no es llamado para el renderizado inicial o cuando forceUpdate() es usado.

Este método sólo existe como optimización de rendimiento**.** No confíes en él para "prevenir" un renderizado, ya que esto puede conducir a errores. **Considere usar el componente integrado** PureComponent en lugar de escribir shouldComponentUpdate() a mano. PureComponent realiza una comparación superficial de props y estado, y reduce la posibilidad de saltar una actualización necesaria.

Si estás seguro de querer escribirlo a mano, puedes comparar this.props con nextProps y this.state con nextState y devolver false para indicar a React que se puede omitir la actualización. Devolver false no previene a los componentes hijos de volverse a renderizar cuando *su* estado cambia.

No recomendamos realizar comprobaciones de igualdad profundas ni utilizar JSON.stringify() en shouldComponentUpdate(). Es muy ineficiente y dañará el rendimiento.

Actualmente, si shouldComponentUpdate() devuelve false, entonces componentWillUpdate(), render(), y componentDidUpdate() no serán invocados. Ten en cuenta que en el futuro React puede tratar a shouldComponentUpdate() como una

sugerencia en lugar de una directiva estricta, y devolver false puede aun dar como resultado una nueva renderización del componente.

static getDerivedStateFromProps()

```
static getDerivedStateFromProps(props, state)
```

getDerivedStateFromProps se invoca justo antes de llamar al método de render, tanto en la montura inicial como en las actualizaciones posteriores. Debes devolver un objeto para actualizar el estado, o null para no actualizar nada.

Este método existe para casos de uso raros donde el estado depende de los cambios en props con el tiempo. Por ejemplo, puede ser util para implementar un componente <Transition> que compare su anterior hijo y el siguiente para decidir cual de los dos animar en la entrada y salida.

Derivar el estado conduce al código verboso y hace que tus componentes sean difíciles de pensar. Asegúrate de que estás familiarizado con alternativas más simples

- Si necesitas **realizar un efecto secundario** (por ejemplo, obtención de datos o animaciones) en una respuesta debido a un cambio en las props, utiliza componentDidUpdate.
- Si quieres **recalcular algunos datos solo cuando una prop cambie**,usa memoization.
- Si quieres **restablecer algún estado cuando una prop cambie** considera hacer un completamente controlado o un componente no controlado con una key.

Este método no tiene acceso a la instancia del componente. Si quieres, puedes reutilizar algún código entre getDerivedStateFromProps() y los otros métodos de clase mediante la extracción de funciones puras de las props del componente y el estado fuera de la definición de clase.

Ten en cuenta que este método se activa en *cada* renderizado, independientemente de la causa. En caso contrario, UNSAFE_componentWillReceiveProps, que sólo se dispara cuando el padre causa un nuevo renderizado y no como resultado de un setState local.

getSnapshotBeforeUpdate()

```
getSnapshotBeforeUpdate(prevProps, prevState)
```

getSnapshotBeforeUpdate() se invoca justo antes de que la salida renderizada más reciente se entregue, por ejemplo, al DOM. Permite al componente capturar cierta información del DOM (por ejemplo, la posición del scroll) antes de que potencialmente se

cambie. Cualquier valor que se devuelva en este método de ciclo de vida se pasará como parámetro a componentDidUpdate().

Este caso de uso no es común, pero puede ourrir en IUs como un hilo de chat que necesita manejar la posición del scroll de manera especial.
Debe devolverse un valor instantáneo (o null).

Por ejemplo:

```
class ScrollingList extends React.Component {
  constructor(props) {
    super(props);
    this.listRef = React.createRef();
  }

  getSnapshotBeforeUpdate(prevProps, prevState) {
    // ¿Estamos agregando nuevos elementos a la lista?
    // Captura la posición del scroll para que podamos ajustar el
scroll después.
    if (prevProps.list.length < this.props.list.length) {
      const list = this.listRef.current;
      return list.scrollHeight - list.scrollTop;
    }
    return null;
  }

  componentDidUpdate(prevProps, prevState, snapshot) {
    // Si tenemos un valor snapshot, agregamos nuevos elementos
    // Ajusta el scroll para que los nuevos elementos no empujen a
los viejos fuera de la vista
    // (snapshot aquí es el valor que regresa de
getSnapshotBeforeUpdate)
    if (snapshot !== null) {
      const list = this.listRef.current;
      list.scrollTop = list.scrollHeight - snapshot;
    }
  }

  render() {
    return (
      <div ref={this.listRef}>{/* ...contents... */}</div>
    );
  }
}
```

En los ejemplos anteriores, es importante leer la propiedad scrollHeight en getSnapshotBeforeUpdate porque puede haber retrasos entre los ciclos de vida de la fase “render” (como render) y los ciclos de fase de “commit” (como getSnapshotBeforeUpdate y componentDidUpdate).

Límites de error

Los límites de error son componentes de React que detectan errores de JavaScript en cualquier parte de su árbol de componentes secundarios, registran esos errores y muestran una IU alternativa en lugar del árbol de componentes que se colgó. Los límites de error capturan errores durante representación, en métodos de ciclo de vida y en constructores de todo el árbol debajo de ellos.

Un componente definido a través de un clase se convierte en un límite de error si se define uno o ambos métodos de ciclo de vida static getDerivedStateFromError() o componentDidCatch(). Actualizar el estado desde estos ciclos de vida te permite capturar eventos no controlados desde JavaScript en el árbol inferior, y mostrarlo como respuesta en la interfaz de usuario.

Usa solo límites de error para recuperar excepciones inesperadas; no intentes usarlos para controlar el flujo
Para más detalles ve el *Manejo de Errores en React 16*.

Nota

Los límites de error solo capturan errores en los componentes **debajo** de ellos en el árbol. Un límite de error no puede capturar un error dentro de él mismo.

static getDerivedStateFromError()

```
static getDerivedStateFromError(error)
```

Este ciclo de vida se invoca después de que un error haya sido lanzado por un componente descendiente. Recibe el error que fue lanzado como parámetro y debe devolver un valor para actualizar el estado.

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {    // Actualiza el
state, así el siguiente renderizado lo mostrará en la IU.
return { hasError: true };  }
  render() {
    if (this.state.hasError) {      // Puedes renderizar cualquier
interfaz de usuario diferente      return <h1>Something went
wrong.</h1>;    }
    return this.props.children;
  }
}
```

Nota

getDerivedStateFromError() se llama durante la fase “render”, por lo que los efectos secundarios no están permitidos. Para estos casos de uso, use componentDidMount() en su lugar.

componentDidCatch()

```
componentDidCatch(error, info)
```

Este ciclo de vida se invoca después de que un error haya sido lanzado por un componente descendiente. Recibe dos parámetros:

1. error - Es un error que ha sido lanzado.
2. info- Un objeto con una clavecomponentStack contiene información sobre que componente ha devuelto un error.

componentDidCatch()se llama durante la fase “commit”, por lo tanto, los efectos secundarios se permiten. Debería utilizarse para cosas como errores de registro:

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // Actualiza el state, así el siguiente renderizado lo
mostrará en la IU.
    return { hasError: true };
  }

  componentDidCatch(error, info) {    // Ejemplo "componentStack":
//   in ComponentThatThrows (created by App)    //   in
ErrorBoundary (created by App)    //   in div (created by App)
//   in App    logComponentStackToMyService(info.componentStack);
}
  render() {
    if (this.state.hasError) {
      // Puedes renderizar una interfaz de usuario customizada
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

Los compilados de producción y desarrollo de React difieren ligeramente en la forma en que componentDidCatch maneja los errores.
En desarrollo, los errores subirán hacia window, esto significa que cualquier window.onerror o window.addEventListener('error', callback) interceptará los errores que han sido atrapados por componentDitCatch.
En producción, en cambio, los errores no subirán, lo que significa que cualquier manejador de errores ancestro solo recibirá errores que no hayan sido explícitamente atrapados por componentDidCatch().

Nota

En el evento de un error, puedes renderizar una interfaz de usuario con componentDidCatch() llamando a setState(), pero esto estará obsoleto en una futura versión. Usa static getDerivedStateFromError() para controlar el plan de renderizado.

Métodos de ciclo de vida obsoletos

Estos métodos de ciclo de vida a continuación están marcados como “legado”. Siguen funcionando, pero no recomendamos su uso en nuevos proyectos. Puedes aprender más sobre como migrar estos métodos a sus correspondientes actuales en este post del blog..

UNSAFE_componentWillMount()

```
UNSAFE_componentWillMount()
```

Nota

Este ciclo de vida se llamaba anteriormente componentWillMount. Ese nombre seguirá funcionando hasta la versión 17. Usa la rename-unsafe-lifecycles codemod para actualizar automaticamente tus componentes.

UNSAFE_componentWillMount() se invoca justo antes de que el montaje ocurra. Es llamado antes de render(), por lo tanto, al llamar a setState() de forma síncrona en este método no se activará una representación adicional. En general, recomendamos usar el constructor() en lugar de inicializar el estado.

Evite introducir efectos secundarios o suscripciones en este método. Para estos casos de uso, use componentDidMount() en su lugar.

Este ciclo de vida es el único método que se llama en el renderizado en el lado de servidor.

UNSAFE_componentWillReceiveProps()

```
UNSAFE_componentWillReceiveProps(nextProps)
```

Nota

Este ciclo de vida se llamaba anteriormente componentWillReceiveProps. Ese nombre seguirá funcionando hasta la versión 17. Usa rename-unsafe-lifecycles codemod para actualizar automaticamente tus componentes.

Nota:

El uso de este método de ciclo de vida a menudo conduce a errores e inconsistencias

- Si necesitas **realizar un efecto secundario** (por ejemplo, obtención de datos o animaciones) en una respuesta debido a un cambio en las props, utiliza componentDidUpdate.
- Si usaste componentWillReceiveProps para **re-calcular algunos datos cuando una prop cambie**, utiliza memoization.
- Si quieres **restablecer algún state cuando una prop cambie** considera hacer un completamente controlado o un componente no controlado con una key/clave.

Para otros casos de uso, sigue las recomendaciones en este blog sobre estado derivado. UNSAFE_componentWillReceiveProps() se invoca antes de que un componente montado reciba nuevas props. Si necesita actualizar el estado en respuesta a cambios de accesorios (por ejemplo, para restablecerlo), puede comparar this.props y nextProps y realizar transiciones de estado usando this.setState () en este método.

Ten en cuenta que si un componente principal hace que su componente se vuelva a generar, se llamará a este método incluso si las props no han cambiado. Asegúrate de comparar los valores actuales y los siguientes solo si deseas manejar los cambios.

React no llama UNSAFE_componentWillReceiveProps() con props inicial durante su mounting. Solo llama a este método si algunas de las props de los componentes deben ser actualizadas. Normalmente, llamar a this.setState() no provoca UNSAFE_componentWillReceiveProps().

UNSAFE_componentWillUpdate()

```
UNSAFE_componentWillUpdate(nextProps, nextState)
```

Nota

This lifecycle was previously named componentWillUpdate. That name will continue to work until version 17. Use the rename-unsafe-lifecycles codemod to automatically update your components.

UNSAFE_componentWillUpdate() se invoca justo antes de renderizar cuando llegan nuevas props o se está recibiendo el estado. Usa esto como una oportunidad para realizar la preparación antes de que ocurra una actualización. Este método no es llamado para el renderizador inicial.

No puedes llamar aquí a this.setState(); tampoco deberias hacer nada más (por ejemplo, enviar una acción de Redux) que activaría una actualización de un componente React antes de que devuelva el método UNSAFE_componentWillUpdate ().

Normalmente, este método puede ser reemplazado por componentDidUpdate(). Si estabas leyendo el DOM en este método (por ejemplo para guardar una posición de desplazamiento), puedes mover esa lógica a getSnapshotBeforeUpdate().

Nota

UNSAFE_componentWillUpdate() no será invocado
si shouldComponentUpdate() devuelve false.

Otras APIs

A diferencia de los métodos de ciclo de vida anteriores (que React llama por ti), los métodos siguientes son los métodos *que* tu puedes llamar desde tus componentes. Hay sólo dos de ellos: setState() y forceUpdate().

setState()

```
setState(updater, [callback])
```

setState() hace cambios al estado del componente y le dice a React que este componente y sus elementos secundarios deben volverse a procesar con el estado actualizado. Este es el método principal que utiliza para actualizar la interfaz de usuario en respuesta a los manejadores de eventos y las respuestas del servidor.

Considera setState() como una *solicitud* en lugar de un comando para actualizar el componente. Para un mejor rendimiento percibido, React puede retrasarlo, y luego actualizar varios componentes en una sola pasada. React no garantiza que los cambios de estado se apliquen de inmediato.

setState() no siempre actualiza inmediatamente el componente. Puede procesar por lotes o diferir la actualización hasta más tarde. Esto hace que leer this.state después de llamar a setState() sea una trampa potencial. En su lugar, usa componentDidUpdate o un callback setState (setState(updater, callback)), se garantiza que cualquiera de los dos se activará una vez la actualización haya sido aplicada. Si necesitas establecer el estado en función del estado anterior, lee a continuación sobre el argumento updater.

setState() siempre llevará al re-renderizado a menos que shouldComponentUpdate() devuelva false. Si se usan objetos mutables y no se puede implementar la lógica de representación condicional en shouldComponentUpdate(), llamar a setState() solo cuando el nuevo estado difiera del estado anterior evitará re-renderizados innecesarios.

El primer argumento es una función updater con la firma:

```
(state, props) => stateChange
```

state es una referencia al estado del componente en el momento en que se está aplicando el cambio. No debería ser mutado directamente. En cambio, los cambios deberían ser representados construyendo un nuevo objeto basado en la entrada de prevState y props. Por ejemplo, supongamos que quisiéramos incrementar un valor en el estado por props.step:

```
this.setState((state, props) => {
  return {counter: state.counter + props.step};
});
```

Ambos state y props recibidos por la función de actualizador están garantizados de estar actualizados. La salida del actualizador se fusiona de forma superficial (*shallow*) con state. El segundo parámetro para setState() es una devolución de llamada opcional que será ejecutada una vez setState sea completada y el componente sea re-renderizado. Generalmente recomendamos usar componentDidUpdate() para dicha lógica.

Opcionalmente puedes pasar un objeto como el primer argumento a setState() en lugar de una función:

```
setState(stateChange[, callback])
```

Esto realiza una combinación superficial de stateChange en el nuevo estado, por ejemplo, para ajustar una cantidad de elementos del carrito de compras:

```
this.setState({quantity: 2})
```

Esta forma de setState() también es asincrónica, y varias llamadas durante el mismo ciclo pueden agruparse juntas. Por ejemplo, si intentas aumentar la cantidad de un articulo mas de una vez en el mismo ciclo, resultará en el equivalente de:

```
Object.assign(
  previousState,
  {quantity: state.quantity + 1},
  {quantity: state.quantity + 1},
  ...
)
```

Las llamadas posteriores anularán los valores de llamadas anteriores en el mismo ciclo, por lo que la cantidad solo se incrementará una vez. Si el siguiente estado depende del estado actual, recomendamos utilizar un actualizador a través de una función formulario.

```
this.setState((state) => {
  return {quantity: state.quantity + 1};
});
```

forceUpdate()

```
component.forceUpdate(callback)
```

Por defecto, cuando el componente de tu estado o accesorio cambia, tu componente re-renderizará. Si tu método render() depende de algunos otros datos, puedes decirle a React que el componente necesita re-renderizado llamando a forceUpdate().

Llamar a forceUpdate() causará que render() sea llamado en el componente, saltando shouldComponentUpdate(). Esto activará los métodos de ciclo de vida normales para los componentes hijos, incluyendo el método shouldComponentUpdate() de cada hijo. React solo actualizará el DOM si el marcado cambia.

Normalmente, debes intentar evitar todos los usos de forceUpdate() y solo leer desde this.props y this.state en render().

Propiedades de clase

defaultProps

defaultProps puede ser definida como una propiedad en la propia clase de componente, para establecer las props predeterminadas para la clase. Esto se utiliza para props con valor undefined, pero no para props con valor null. Por ejemplo:

```
class CustomButton extends React.Component {
  // ...
}

CustomButton.defaultProps = {
  color: 'blue'
};
```

Si no se proporciona props.color, se establecerá por defecto a 'blue':

```
  render() {
    return <CustomButton /> ; // props.color será asignada a azul
  }
```

Si props.color es null, permanecerá null:

```
  render() {
    return <CustomButton color={null} /> ; // props.color se
mantendrá en null
  }
```

displayName

La cadena displayName es usada en la depuración de mensajes. Por lo general, no necesitas establecerlo explícitamente porque se deduce del nombre de la función o clase que define el componente. Es posible que desees establecerlo explícitamente si quieres mostrar un nombre diferente para la depuración o cuando creas un componente de orden superior, consulta Ajustar el Nombre de Pantalla para una Fácil Depuración para más detalles.

Propiedades de instancia

Props

this.propscontiene las props que fueron definidas por el encargado de llamar al componente. Mira Componentes y Props para una introducción a las props.
En particular, this.props.children es una prop especial, típicamente definida por las etiquetas hijas en la expresión JSX en vez de en la etiqueta como tal.

Estado

El estado contiene datos específicos de este componente que pueden cambiar con el tiempo. El estado está definido por el usuario, y debe ser un simple objeto JavaScript.
Si no se utiliza algún valor para renderizar o simplemente flujos de datos (por ejemplo, un ID de temporizador), no tiene que ponerlo en el estado. Dichos valores se pueden definir como campos en la instancia del componente.

Consulta Estado y Ciclo de Vida para más información sobre el estado.

Nunca mutes this.state directamente, ya que llamar setState() después podría reemplazar la mutación que habías hecho anteriormente. Intenta tratar this.state como si fuera inmutable.