¿QUÉ SON LAS CHILDREN EN REACT?

Las Children son una forma de composición de componentes, donde, valga la redundancia, podemos pasar componentes a través de propiedades.

<Component>

<ChildComponent/>

</Component>

¿CÓMO PODEMOS AGREGAR CHILDREN A UN COMPONENTE?

Esto se lleva acabo cuando escribimos un componente mediante el marcado de doble etiqueta, y entre su etiqueta de apertura y cierre, escribimos un componente hijo valido, por ejemplo:

```
<Padre>
  <Hijo1/>
  <Hijo2/>
</Padre>
```

Al hacer lo anterior, en las propiedades del componente <Padre /> aparece el atributo Children, el cual, contiene un array con la información de los dos componentes <Hijo1/> e <Hijo2 />.

Por lo tanto, en el componente <Padre /> podemos recuperar los hijos y montarlos directamente en la DOM.

```
class Padre extends React.Component{
    render(){
        const {children} = this.props;

        return(
            <div className="padre">
                <h1>{"Propiedades Children"}</h1>
                <div className="hijos">
                  {
                    children
                  }
                </div>
```

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

En código anterior:

Por destructuración obtenemos la propiedad children.

Y en el return del componente agregamos una llave y escribimos las children.

Al hacer esto, React montara dicha propiedad como componentes en nuestra DOM.

Así de sencillo podemos utilizar este tipo de propiedades.

TIPOS DE COMPONENTES VALIDOS EN LAS CHILDREN

En esencia podemos pasar tipos como:

- Componentes
- Elementos como párrafos, div, headers (H2,H3,etc).
- Textos
- Expresiones de JavaScript como Templete Strings
- Operaciones
- Funciones

```
<Padre>
  <Hijo1/>
  <p>Párrafo</p>
   Texto...
   <br/>
   {`Template String ${'Hola'}`}
   {()=>{}}
</Padre>
```

Nota: *A pesar de que podemos pasar funciones en las Children, en React no se puede renderizar una función, por lo tanto, en la consola del navegador verás un error. Para solucionar esto, deberás interceptar antes del render las propiedades Children y aplicar un filtro, ya se para separar dicha función, o eliminarla del render.*

¿CÓMO MANIPULAR Y FILTRAR CHILDREN?

Podemos realizar algunas acciones antes de montar las props Children en la DOM, a esto se le llama filtro, ya que eliminas aquellos componentes no validos y/o separas funciones que necesitas pero que no quieres que se monten en la DOM.

PROPS: RELACIÓN DE CHILDREN Y PROPS

React inyecta automáticamente children en las props, sólo si encuentra alguno.

introducción 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.

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.

Nota: *React 16.8.0 es la primera versión que es compatible con Hooks. Al actualizar, no olvides actualizar todos los paquetes, incluyendo React DOM. React Native es compatible con Hooks desde la versión 0.59 de React Native.*

Antes de continuar, debes notar que los Hooks son:

Completamente opcionales. Puedes probar Hooks en unos pocos componentes sin reescribir ningún código existente.

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.

No hay planes para remover el modelo de clases de React. Puedes leer más sobre la estrategia de adopción gradual de Hooks en la sección inferior de esta página.

Los Hooks no reemplazan tu conocimiento de los conceptos de React. Por el contrario, los Hooks proporcionan una API más directa a los conceptos que ya conoces de React: props, estado, contexto, referencias, y ciclo de vida. Como veremos más adelante, los hooks también ofrecen una nueva y poderosa forma de combinarlos.

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.

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.

Los componentes complejos se vuelven difíciles de entender

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.

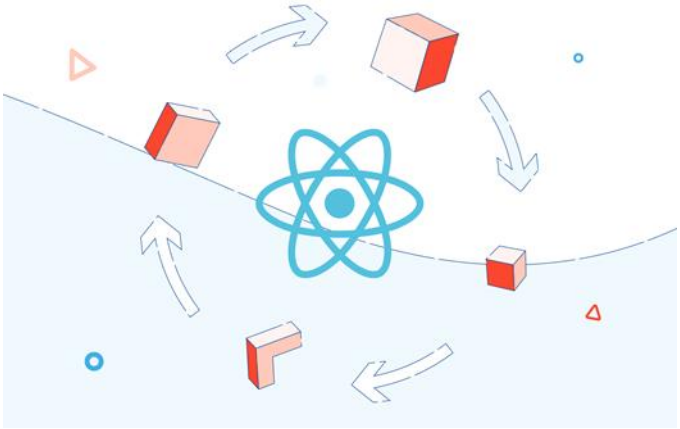
Las clases confunden tanto a las personas como a las máquinas

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.

Ciclo de vida



El ciclo de vida no es más que una serie de estados por los cuales pasa todo componente a lo largo de su existencia.

Esos estados tienen correspondencia en diversos métodos,

que podemos implementar para realizar acciones cuando se van produciendo.

En React es fundamental el ciclo de vida, porque hay determinadas acciones que necesariamente debemos realizar en el momento correcto de ese ciclo.

Conocer estos ciclos nos ayudará a optimizar la aplicación, siguiendo las reglas básicas que pone React

Hay más reglas pero por ahora tengamos en mente las más básicas:

- No bloquear el rendering con tareas pesadas y sincrónicas.
- Ejecutar tareas asincrónicas con efectos secundarios luego del montaje (mount).

LAS TRES CLASIFICACIONES DE ESTADOS DENTRO DE UN CICLO DE VIDA

- El montaje se produce la primera vez que un componente va a generarse, incluyéndose en el DOM.
- La actualización se produce cuando el componente ya generado se está actualizando.
- El desmontaje se produce cuando el componente se elimina del DOM.

Nota: Cada método tiene un prefijo will o did dependiendo de si ocurren antes o después de cierta acción.

Montado

La primera fase ocurre solo una vez por componente cuando este se crea y monta en la UI. Esta fase se divide en 4 funciones.

constructor(props)

Este método se ejecuta cuando se instancia un componente. Nos permite definir el estado inicial del componente, hacer bind de métodos y definir propiedades internas en las que podemos guardar muchos datos diferentes, por ejemplo, la instancia de una clase (un parser, un validador, etc.).

componentWillMount()

Este método se ejecuta cuando el componente se está por renderizar. En este punto es posible modificar el estado del componente sin causar una actualización (y por lo tanto no renderizar dos veces el componente). Es importante sin embargo evitar causar cualquier clase de efecto secundario (petición HTTP por ejemplo) ya que este método se ejecuta en el servidor y hacer esto puede causar problemas de memoria.

render()

En este momento de la fase de montado se van a tomar las propiedades, el estado y el contexto y se va a generar la UI inicial de este componente. Esta función debe ser pura (no puede tener efectos secundarios) y no debe modificar nunca el estado del componente.

Actualizar el estado en este punto puede causar un ciclo infinito de renderizados, ya que cada cambio al estado genera que el componente se renderice de nuevo (y vuelva a cambiar el estado).

componentDidMount()

Este último método de la fase de montado se ejecuta una vez el componente se renderizó en el navegador (este no se ejecuta al renderizar en el servidor) y nos permite interactuar con el DOM o las otras APIs del navegador (geolocation, navigator, notificaciones, etc.).

También es el mejor lugar para realizar peticiones HTTP o suscribirse a diferentes fuentes de datos (un Store o un WebSocket) y al recibir una respuesta, actualizar el estado. Cambiar el estado en este método causa que se vuelva a renderizar el componente.

Actualización

Esa fase puede ocurrir múltiples veces (o incluso ninguna), sucede cuando algún dato del componente (ya sea una propiedad, un estado o el contexto) se modifica y por lo tanto requiere que la UI se vuelva a generar para representar ese cambio de datos.

componentWillReceiveProps(nextProps)

Este método se ejecuta inmediatamente después que el componente reciba nuevas propiedades. En este punto es posible actualizar el estado para que refleje el cambio de propiedades, ya sea reiniciando su valor inicial o cambiándolo por uno nuevo.

Hay que tener en cuenta que React puede llegar a ejecutar este método incluso si las propiedades no cambiaron. Por eso es importante validar que las nuevas propiedades (nextProps) sean diferentes de las anteriores (this.props).

shouldComponentUpdate(nextProps, nextState)

Este método (el cual debe ser puro) se ejecuta antes de empezar a actualizar un componente, cuando llegan las nuevas propiedades (nextProps) y el nuevo estado (nextState).

Acá es posible validar que estos datos sean diferentes de los anteriores (this.props y this.state) y devolver true o false dependiendo de si queremos volver a renderizar o no el componente.

Los componentes creados al extender React.PureComponent implementan esta validación sin necesidad de que hagamos nada y de una forma que no afecte al rendimiento. El resto de componentes devuelven siempre true por defecto.

Hay que tener cuidado con este método ya que si nuestro componente tiene otros componentes con estado como hijos, devolver false acá puede impedir que estos sub-componentes no se actualicen al detectar un cambio.

componentWillUpdate(nextProps, nextState)

Una vez el método anterior devolvió true se ejecuta este método, acá es posible realizar cualquier tipo de preparación antes de que se actualice la UI.

Es importante tener encuenta que acá no se puede ejecutar this.setState para actualizar el estado. Si queremos actualizar el estado con base a un cambio de propiedades debemos hacerlo en componentWillReceiveProps.

render()

Al igual que en el montado acá se va a generar la UI, esta vez con los datos que hayan cambiado. Como antes este método debe ser puro.

componentDidUpdate(prevProps, prevState)

Esta última parte de la actualización de un componente ocurre justo después de que se renderiza en el DOM nuestro componente. Al igual que con componentDidMount() acá es posible interactuar con el DOM y cualquier API de los navegadores.

Aunque acá podríamos realizar una petición HTTP y actualizar el estado hay que tener cuidado, ya que de hacerlo podríamos causar un bucle infinito de actualizaciones y peticiones HTTP.

Desmontado

Esta última fase consiste en un solo método que se ejecuta antes de que un componente se elimine (desmonte) de la UI de nuestra aplicación.

componentWillUnmount()

Este único método de la fase de desmontado nos permite realizar cualquier tipo de limpieza antes de remover el componente.

Acá es posible dejar de escuchar eventos de window, document o el DOM, desuscribirse de un WebSocket o Store o cancelar peticiones HTTP que hayan quedado pendientes.

Es importante hacer esta limpieza ya que si alguna petición pendiente se completa luego del desmontado, va a tratar de actualizar el estado y va a dar un error (y hasta un posible problema de memoria) ya que el componente no existe más.

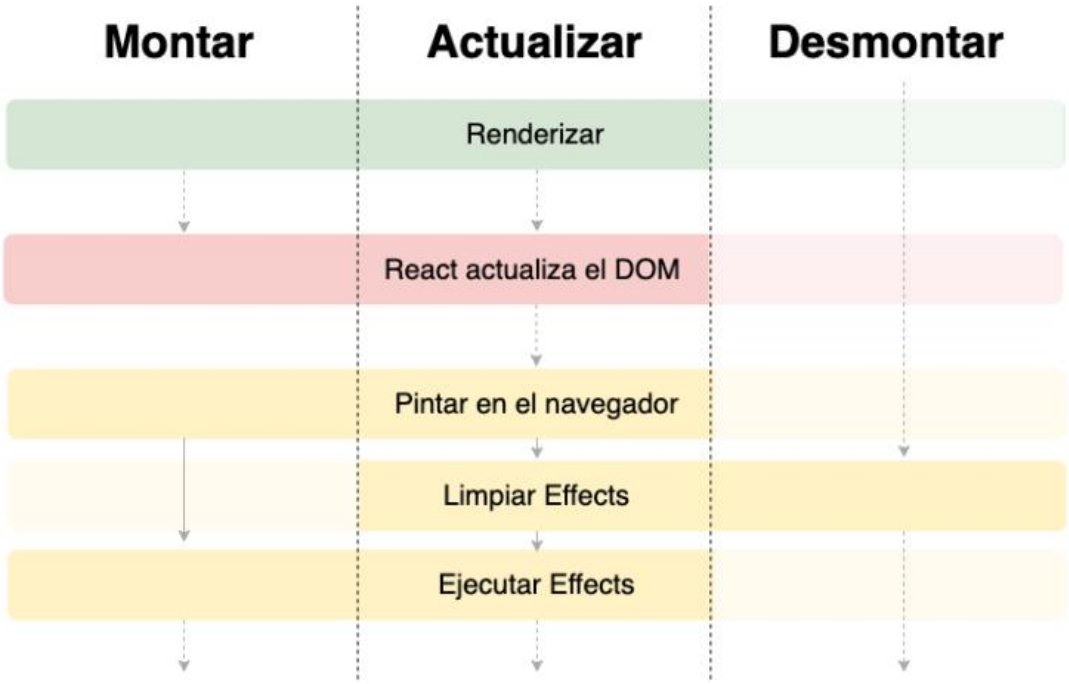
useEffect

Este React hook, useEffect, nos permite ejecutar y controlar efectos colaterales, como pueden ser peticiones a servicios, subscribirse a eventos, modificar el DOM o cualquier funcionalidad que no pueda ejecutarse en el cuerpo de nuestro componente función porque no pertenece al flujo lineal del mismo.

Flujo de useEffect

La función effectFn se ejecuta después de que el navegador ya ha pintado el componente en pantalla por primera vez (montar). También por defecto después de cada posterior repintado (actualizar). Este comportamiento descrito tiene el mismo propósito que los métodos componentDidMount y componentDidUpdate.

El tercer propósito en useEffect se le llama limpieza, el cual lo podemos comparar con componentDidUnmount. En el primer pintado (montar) la función de limpieza no se ejecuta, solo se ejecuta en la fase de actualizar. Es decir, se ejecuta después de cada repintando, pero antes del que el cuerpo de useEffect se ejecute.



useEffect recibe un segundo parámetro, deps, el cual es un Array con la lista de dependencias que permiten decidir si ejecutar el efecto colateral o no, después de cada repintado. Sirve bastante para mejorar el rendimiento si no queremos que después de cada repintado se ejecute effectFn.

Si te das cuenta, se ha usado la palabra pintado en lugar de renderizado. Esto se debe a que efectivamente el efecto se ejecuta después de que los cambios ya estén pintados en el navegador web. El renderizado involucra al Virtual DOM, y React decide en que momento es conveniente actualizar el DOM, pero el pintado sucede un poco después

de lo anterior. Aclaremos que aunque se actualice el DOM, el navegador debe calcular estilos y el layout de los elementos para posteriormente realizar el pintado de los pixeles.