



**Tabla de contenido**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| [Introducción](#page3) | | | 1.1 | | |
|  |  |  |  |  |  |
| [[WIP] Reaccionar de adentro hacia afuera](#page5) | | | 1.2 | | |
|  |  |  |  |  |  |
| [El Reaccionar del Ciclo de Vida](#page6) | | | 1.3 | | |
|  |  |  |  |  |  |
|  | [Ciclo de Vida Métodos Descripción](#page8) | | 1.3.1 | | |
|  |  |  |  |  | |
|  | [Nacimiento / de montaje en profundidad](#page10) | | 1.3.2 | | |
|  |  |  |  |  | |
|  |  | [Inicialización y Construcción](#page13) | 1.3.2.1 | | |
|  |  |  |  | | |
|  |  | [Pre-montaje con componentWillMount ()](#page17) | 1.3.2.2 | | |
|  |  |  |  | | |
|  |  | [Componente render ()](#page20) | 1.3.2.3 | | |
|  |  |  |  | | |
|  |  | [La gestión de la Infancia y de componentes de montaje](#page23) | 1.3.2.4 | | |
|  |  |  |  | | |
|  |  | [Post-montaje con componentDidMount ()](#page25) | 1.3.2.5 | | |
|  |  |  |  | | |
|  | [Crecimiento / actualización en profundidad](#page31) | | 1.3.3 | | |
|  |  |  |  |  | |
|  |  | [Actualización y componentWillReceiveProps ()](#page34) | 1.3.3.1 | | |
|  |  |  |  | | |
|  |  | [Usando shouldComponentUpdate ()](#page37) | 1.3.3.2 | | |
|  |  |  |  | | |
|  |  | [El aprovechamiento de componentWillUpdate ()](#page41) | 1.3.3.3 | | |
|  |  |  |  | | |
|  |  | [Re-representación y actualizaciones niños](#page43) | 1.3.3.4 | | |
|  |  |  |  | | |
|  |  | [Post-Procesamiento con componentDidUpdate ()](#page45) | 1.3.3.5 | | |
|  |  |  |  | | |
|  | [Muerte / desmontar en profundidad](#page48) | | 1.3.4 | | |
|  |  | |  |  | |
|  | [La Crónica del Ciclo de Vida](#page50) | | 1.3.5 | | |
|  |  | |  |  | |
| [Componente Evolución y Composición](#page52) | | | 1.4 | | |
|  |  | |  | |  |
|  | [La evolución de un componente List](#page54) | | 1.4.1 | | |
|  |  | |  |  | |
|  | [Representar el contenido diferente](#page59) | | 1.4.2 | | |
|  |  | |  |  | |
|  | [Componentes de orden superior](#page65) | | 1.4.3 | | |
|  |  | |  |  | |
| [Sobre los autores](#page70) | | | 1.5 | | |
|  |  |  |  |  |  |



2

Introducción



**Reaccionar en profundidad: Una exploración de desarrollo de interfaz de usuario**

A [DevelopmentArc®](http://developmentarc.com) tenemos una historia de amor con el desarrollo de aplicaciones de interfaz de usuario. Verdaderamente disfrutar de la exploración de tecnologías y marcos. Este proceso nos ayuda a comprender mejor los objetivos finales y la complejidad de la pila de aplicaciones.

Descubrimos desde el principio la necesidad de aplicar verdaderos principios de desarrollo de software a la interfaz de usuario. Este descubrimiento se inició hace mucho tiempo con tecnologías como Authorware / Partnering Toolbook y director. Rápidamente nos mudamos a HTML 1.0 y comenzó a construir sistemas con marcos de JavaScript como Prototype.js. Con el lanzamiento de la estructura de Flex, nuestro enfoque cambió de nuevo a una nueva plataforma, Flash.

Para la mayoría, Flash fue una experiencia negativa y la tecnología de interfaz de usuario. Mientras que muchos argumentos son ciertos, encontramos Flash para ser el sistema de renderizado multi-plataforma más avanzada disponible en el momento. Flex nos dio el marco de aplicación necesaria para construir las aplicaciones grandes y complejas clientes estaban solicitando. Al mismo tiempo, Flex se prestó para la gestión de equipos grandes y diversas. Nuestra obsesión con Flex incluye un profundo entender de los ciclos de aplicación y la vida de los componentes, lo que resulta en un 90 páginas de papel blanco. El documento está disponible y continúa siendo referencia en la actualidad. Con el colapso de la plataforma Flash y Flex, nos encontramos de nuevo en HTML y JavaScript mundo. Esta vez nuestra obsesión es[React.js](https://facebook.github.io/react/)

A lo largo de los años hemos seguido para tratar de empujar los límites de las tecnologías de interfaz de usuario. A partir, a finales de 2014, comenzamos la investigación inicial y la adopción a continuación, llena de[React.js](https://facebook.github.io/react/) para nuestra interfaz de usuario de la capa de aplicaciones web.

El proceso inicial de movimiento para reaccionar era una mezcla de emoción y, a veces, la frustración pura. Reaccionar trae en ambos paradigmas de interfaz de usuario existentes y también nuevos patrones que pueden tomar un poco de tiempo para ajustar su propia mente-ajustado a. Una vez que habían asimilado completamente Reaccionar, encontramos que ha abierto las posibilidades para nuestros proyectos actuales y futuros.

Nuestro objetivo con este GitBook es documentar nuestro proceso, compartir nuestra investigación y tratar de organizar nuestras experiencias en un solo documento vivo. Para ser sincero, este es un objetivo bastante elevada. Es posible que no lograr este objetivo, pero espero que podamos conseguir algunos pensamientos positivos hacia abajo.

Hemos encontrado la más larga se escribe código, más se aprende y luego olvidarse en breve. Esto significa que nuestros escritos son apenas tanto para nosotros mismos como para los demás. Con esto en mente, esperamos que a medida que crece este documento que le ayudará, tanto como nos ha ayudado a poner nuestros propios pensamientos.



3

Introducción



*James y Aaron*



4

[WIP] Reaccionar de adentro hacia afuera



**[WIP] Reaccionar de adentro hacia afuera**

***Esta sección está siendo investigado actualmente***

Mientras tanto, hay toneladas de conseguir recursos iniciadas ya disponibles. Si usted es nuevo para reaccionar recomendamos pasar algún tiempo buscando en estos fantásticos recursos:

[Reaccionar Sitio Oficial](http://facebook.github.io/react/)



[impresionante Reaccionar](https://github.com/enaqx/awesome-react) - Una lista completa de los recursos



[Aprendizaje reaccionar, Getting Started](https://scotch.io/tutorials/learning-react-getting-started-and-concepts) - el artículo de introducción básica



Desde una perspectiva de desarrollo del libro, nuestro primer enfoque estará en el [capítulos del ciclo de vida](#page6) y vamos a la vuelta de nuevo a esta sección pronto.

**Los directores de guía para el aprendizaje Reaccionar**

A diferencia de la mayoría de los libros de introducción, queremos tener diferentes objetivos para nuestra sección de Fundamentos. Vamos a profundizar en los fundamentos de reaccionar y buscar la forma de crear aplicaciones de abajo hacia arriba. Creemos que entender el funcionamiento interno de un sistema de interfaz de usuario ayuda a las opciones de desarrollo en coche. Tenemos la intención de explorar Reaccionar de esta guía principal.



5

El Reaccionar del Ciclo de Vida



**El Reaccionar del Ciclo de Vida**

Uno de los factores que definen una forma de vida es su ciclo de vida. El camino común es nacimiento, crecimiento hacia la madurez y la muerte inevitable. aplicaciones de interfaz de usuario a menudo siguen un camino similar. Cuando se inicia por primera vez la aplicación, consideramos que este nacimiento. Los usuarios interactúan con la aplicación, que es el crecimiento. Con el tiempo, se cierra la aplicación o navegar lejos de, lo que lleva a la muerte.

Dentro de la aplicación, los elementos también siguen este patrón. En el mundo de la Reaccionar, estos elementos son nuestros componentes. El ciclo de vida de los componentes es un proceso continuo, que se produce durante toda la vida en general de nuestra aplicación. La comprensión de este proceso puede conducir a un desarrollo más rápido y consistente, fácil optimización y mejora de la salud general de la aplicación.

**fases del ciclo de vida en React componentes**

No todos los sistemas de interfaz de usuario permiten a un patrón de ciclo de vida. Esto no quiere decir que un sistema es mejor o peor si un ciclo de vida es o no se implementa. Todo lo que un ciclo de vida que hace es proporcionar un orden específico de operación y una serie de ganchos para atar en dicho sistema. El ciclo de vida Reaccionar sigue el nacimiento común, Crecimiento, y el flujo de la Muerte. El equipo Reaccionar ha proporcionado una serie de métodos que se pueden implementar / anulación para aprovechar el proceso.

[**Fase 1: Nacimiento / Montaje**](#page10)

La primera fase del ciclo de vida Reaccionar componente es la fase de nacimiento / montaje. Aquí es donde empezamos la inicialización del componente. En esta fase, el componente deaccesorios y estadose definen y configurado. El componente y todos sus hijos están montados en la interfaz de usuario nativa Pila (DOM, UIView, etc.). Por último, no podemos hacer procesamiento posterior, si es necesario. El / la fase de montaje del nacimiento se produce sólo una vez.



[**Fase 2: Crecimiento / Actualización**](#page31)

La siguiente fase del ciclo de vida es la fase de crecimiento / actualización. En esta fase, se obtiene nuevaaccesorios , cambio estado , Manejar las interacciones del usuario y comunicarse con el componente jerarquía. Aquí es donde pasamos la mayor parte de nuestro tiempo en la vida del componente. A diferencia de nacimiento o defunción, repetimos esta fase una y otra vez.



[**Fase 3: Muerte / Desmontar**](#page48)



6

El Reaccionar del Ciclo de Vida



La fase final del ciclo de vida es la fase de muerte / Desmontar. Esta fase se produce cuando una instancia de componente se desmonta desde la interfaz de usuario nativa. Esto puede ocurrir cuando el usuario se desplaza fuera, los cambios de la página de interfaz de usuario, un componente está oculto (como un cajón), etc. La muerte se produce una vez y se prepara el componente para la recolección de basura.

***Lo siguiente:*** [Ciclo de Vida Métodos Descripción](#page8)



7

Ciclo de Vida Métodos Descripción



**Reaccionar Los métodos del ciclo de vida general**

El equipo de desarrollo Reaccionar ofrece una serie de ganchos podemos aprovechar en cada fase del ciclo de vida. Estos ganchos método nos informan de que el componente está en el ciclo de la vida y lo que pueden y no pueden hacer.

Cada uno de los métodos de ciclo de vida, son llamados en un orden específico y en un momento determinado. Los métodos también están vinculados a diferentes partes del ciclo de vida. Éstos son los métodos desglosados ​​en orden y por su fase de ciclo de vida correspondiente[1](#page8):

**Nacimiento / Montaje**

1. Inicializar / Construcción
2. getDefaultProps () *(React.createClass)* o MyComponent.defaultProps *(Clase ES6)*
3. getInitialState () *(React.createClass)* o this.state = ... *(Constructor ES6)*
4. componentWillMount ()
5. hacer()
6. Los niños de inicialización y ciclo de vida de patada de salida
7. componentDidMount ()



**Crecimiento / Actualización**

1. componentWillReceiveProps ()
2. shouldComponentUpdate ()
3. componentWillUpdate ()
4. hacer()
5. métodos de ciclo de vida de los niños
6. componentDidUpdate ()



**Muerte / Desmontar**

1. componentWillUnmount ()
2. métodos de ciclo de vida de los niños
3. Instancia destruyó la recolección de basura



8

Ciclo de Vida Métodos Descripción



El orden de estos métodos son estrictas y llamada como se define anteriormente. La mayor parte del tiempo se dedica en la fase de crecimiento / Actualización y esos métodos son llamados muchas veces. Los métodos de nacimiento y la muerte sólo se llama una vez.

***Hasta la próxima***: [Nacimiento / de montaje en profundidad](#page10)



* *La mayoría de los métodos son los mismos si utiliza cualquiera React.createClass o utilizar clases ES6, tales como MyComponent clase extiende React.Component. Unos pocos son diferentes, principalmente en torno a cómo se produce la creación de instancias / creación. Vamos a llamar a estas diferencias a lo largo del capítulo.*



9

Nacimiento / de montaje en profundidad



**Nacimiento / de montaje en profundidad**

Un componente Reaccionar se inicia el ciclo de vida durante la aplicación ex inicial: ReactDOM.render () . Con la inicialización de la instancia del componente, que comenzará a moverse a través de la fase de nacimiento de su ciclo de vida. Antes de profundizar más en la mecánica de la fase de nacimiento, vamos a retroceder un poco y hablar de lo que esta fase se centra en.

El enfoque más obvio de la fase de nacimiento es la configuración inicial para nuestra instancia de componente. Aquí es donde se pasa en elaccesoriosque definirá la instancia. Sin embargo, durante esta fase hay muchas más piezas móviles que podemos aprovechar.

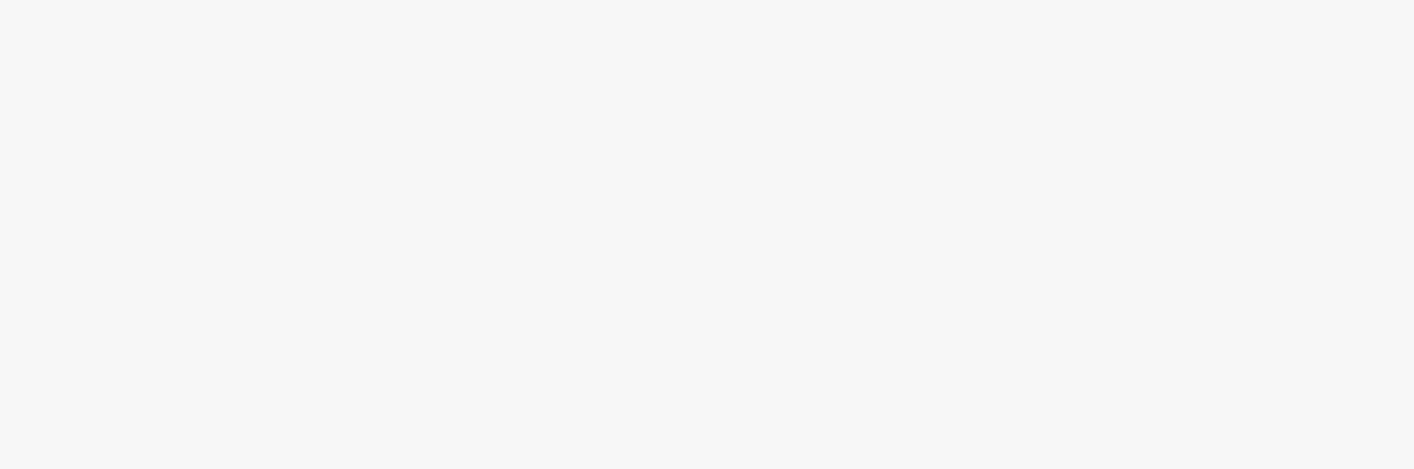


En Nacimiento configuramos el valor por defecto estadoy obtener acceso a la pantalla inicial de la interfaz de usuario. También se inicia el proceso de montaje para los niños del componente. Una vez que se montan los niños, obtenemos el primer acceso a la capa de interfaz de usuario nativa[1](#page10) (DOM, UIView, etc.). Con acceso a la interfaz de usuario nativa, podemos empezar a consultar y modificar la forma en realidad se muestra nuestro contenido. Esto también es cuando podemos empezar el proceso de integración de las bibliotecas y componentes de interfaz de usuario 3 ª parte.



**Componentes vs. Elementos**

Cuando el aprendizaje Reaccionar, muchos desarrolladores tienen un error común. A primera vista, es de suponer que una instancia montado es lo mismo que una clase de componente. Por ejemplo, si se crea un nuevo componente y luego Reaccionarhacer() al DOM:



importar Reaccionar de 'reaccionar';

importar ReactDOM de 'Reaccionar-dom';

clase MyComponent se extiende Reaccionar.Componente {

render () {

regreso <Div>Hola Mundo!</ Div>;

}

};

ReactDOM.render (<MyComponent />, Document.getElementById ( 'punto de montaje'));

La suposición inicial es que durante hacer() una instancia de la MyComponent Se crea la clase, usando algo como nueva MyComponent (). Esta instancia se pasa a rendir. Aunque esto suena razonable, la realidad de que el proceso es un poco más complicado.

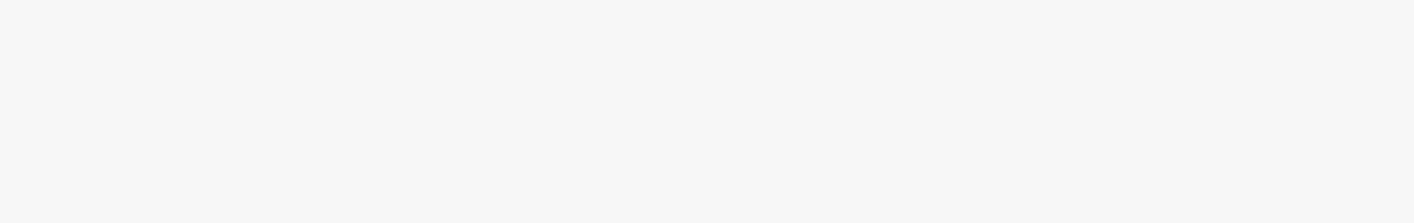


Lo que está ocurriendo en realidad es el procesador convierte la JSX <MyComponent /> línea usar React.createElement para generar la instancia. Esto generó elemento es lo que se pasa a el hacer() método:



10

Nacimiento / de montaje en profundidad



* código generado post-procesamiento JSX ReactDOM.render (

React.createElement (MyComponent, nulo), documento.getElementById ('punto de montaje'));

Un elemento Reaccionar en realidad es sólo una descripción[2](#page10) de lo que eventualmente será utilizado para generar la interfaz de usuario nativa. Este es un núcleo, perdón el interior, el elemento de la tecnología de DOM virtual en React.



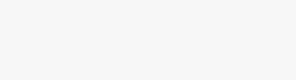
El tipo principal en React es la ReactElement. Tiene cuatro propiedades: Tipo, apoyos, claves y ref. No tiene nada métodos y sobre el prototipo.

* <https://facebook.github.io/react/docs/glossary.html#react-elements>

El elemento es una representación de objetos de peso ligero de lo que será la instancia del componente. Si tratamos de acceder al elemento pensando que es la instancia de clase vamos a tener algunos problemas, como la disponibilidad de métodos esperados.

Así que, ¿Cómo encaja esto en el ciclo de vida? Estos elementos descriptores son esenciales para la creación de la interfaz de usuario nativa y son el catalizador para el ciclo de vida.

**El primero** **hacer()**



Para más Reaccionar a los desarrolladores, las hacer()método es el más familiar. Escribimos nuestra JSX y el diseño aquí. Es el lugar donde pasamos mucho tiempo e impulsa el diseño de la aplicación. Cuando hablamos de la primerahacer() esta es una versión especial de la hacer() método que se monta toda nuestra aplicación nativa en la interfaz de usuario.

En el navegador, este es el ReactDOM.render ()método. Aquí se pasa en la raíz Elemento and Tell intervengan cuando para montar nuestro contenido. Con esta llamada, Reaccionar comienza a procesar el elemento (s) pasado y generar instancias de nuestros componentes reaccionan. El elemento se utiliza para generar la instancia de tipo y luego elaccesorios se pasan a la instancia de componente.



Este es el punto en el que entramos en el ciclo de vida de los componentes. Reaccionar utiliza elejemplo propiedad en el elemento y comienza la construcción.



***Lo siguiente:*** [Inicialización y Construcción](#page13)



* La capa de interfaz de usuario nativa es el sistema que maneja la presentación del contenido de la interfaz de usuario de pantalla. En un navegador, este es el DOM. En el dispositivo, esto sería los UIView (o comparable). Reaccionar se encarga de la traducción de los contenidos de componentes para el formato de capa nativa.

2



11

Nacimiento / de montaje en profundidad



* Dan Abramov intervino con esta terminología en una pregunta StackOverflow. <http://stackoverflow.com/a/31069757>



12

extender React.Component

Inicialización y Construcción



**Inicialización y Construcción**



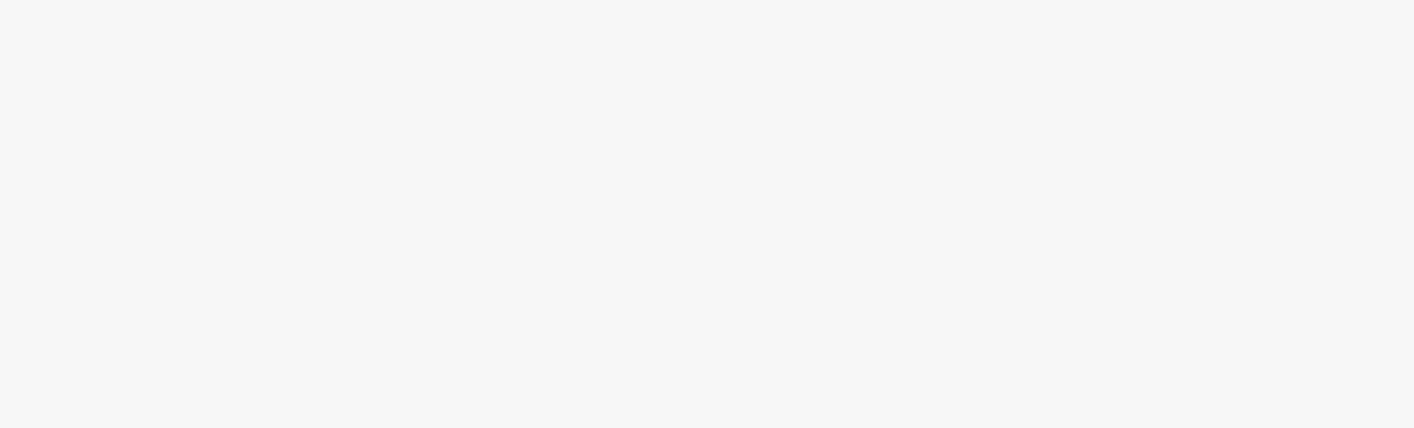
Durante la inicialización del componente del elemento, la accesorios y estadoestán definidos. Cómo estos valores se definen depende de si está utilizandoReact.createClass () o . Veamos primero aaccesorios y luego vamos a examinar estado .

**apoyos por defecto**

Como hemos mencionado anteriormente, la instancia de elemento contiene la corriente accesoriosque se están pasando a la instancia de componente. La mayoría de las veces, toda la disposiciónaccesoriosen el componente no son necesarios. Sin embargo, algunas veces es necesario tener valores para toda laaccesorios para nuestro componente para procesar correctamente.



Por ejemplo, tenemos un componente simple que hace que un nombre y edad.



importar Reaccionar de 'reaccionar';

clase predeterminada de exportación Persona se extiende Reaccionar.Componente { render () {

regreso (

<Div>{} This.props.name (edad: {} this.props.age)</ Div> );

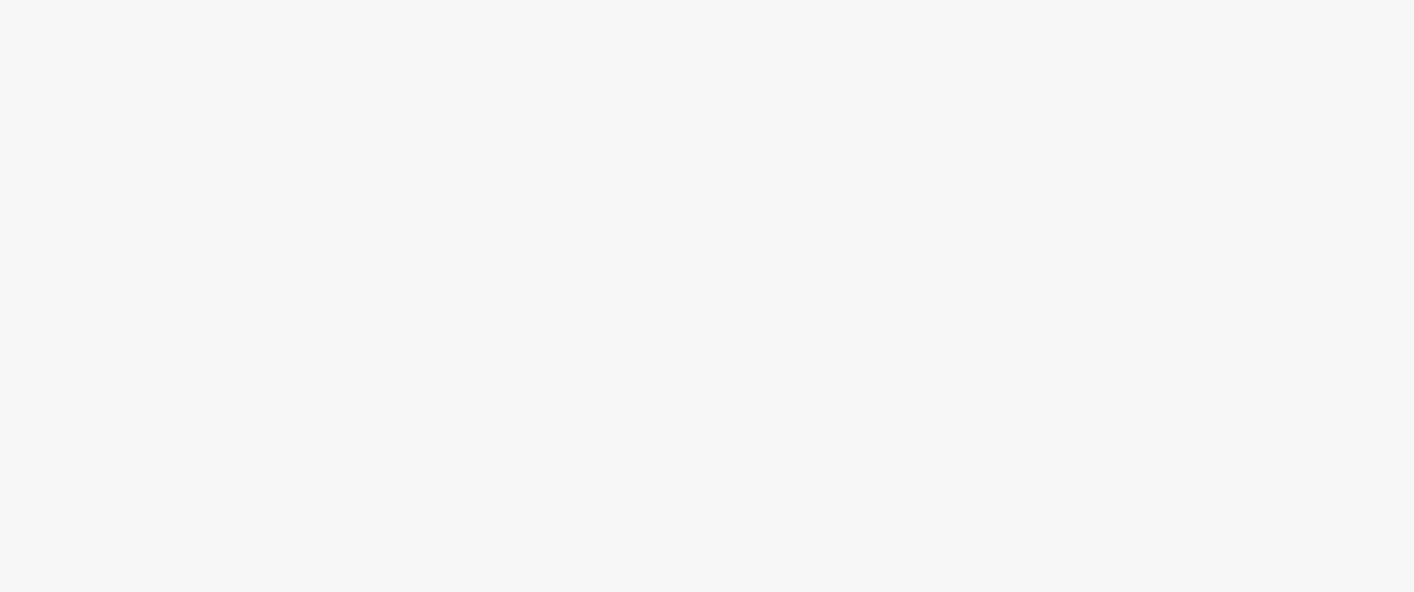
}

}

En nuestro caso, esperamos dos puntales que se pasarán en: nombre y años. Si queremos haceraños opcional y por defecto al texto 'desconocido' que podemos tomar ventaja de los apoyos Reaccionar por defecto.



**Para ES6 Clase**



importar Reaccionar de 'reaccionar';

clase Persona se extiende Reaccionar.Componente {

render () {

regreso (

<Div>{} This.props.name (edad: {} this.props.age)</ Div> );

}

}

Person.defaultProps = {edad: 'desconocido' };

por defecto de exportación Persona;

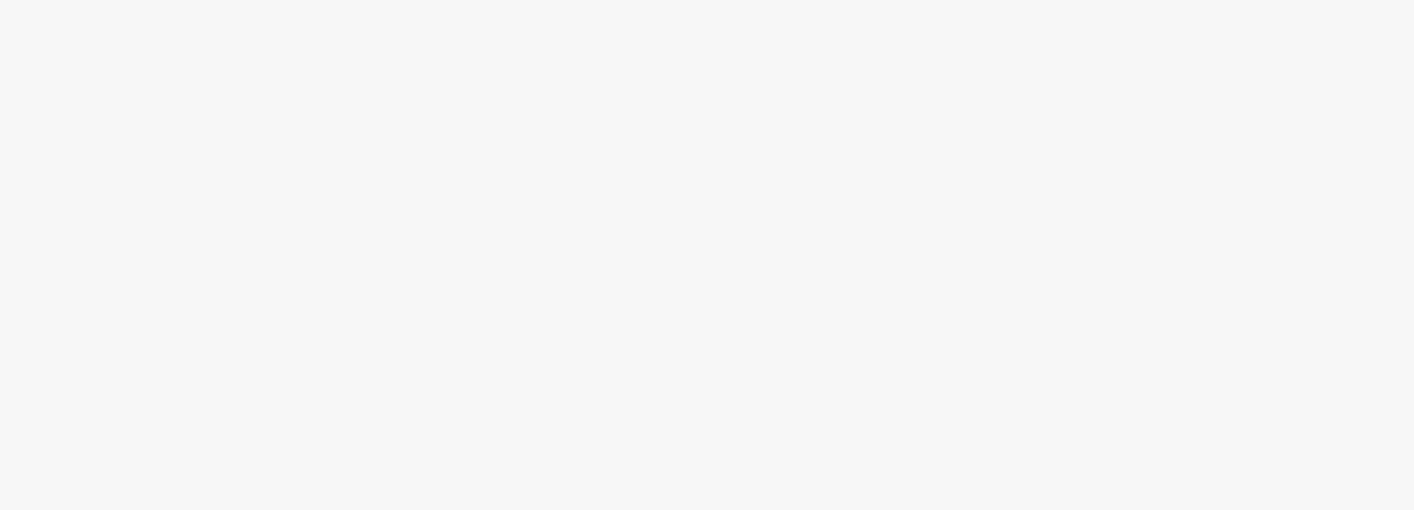


13

Inicialización y Construcción



**Para createClass (ES6 / ES5 / CoffeeScript, etc.)**



var Persona = React.createClass ({

getDefaultProps: función() {

regreso ({ años: 'desconocido' });

},

hacer: función() {

regreso (

<Div>{} This.props.name (edad: {} this.props.age)</ Div> );

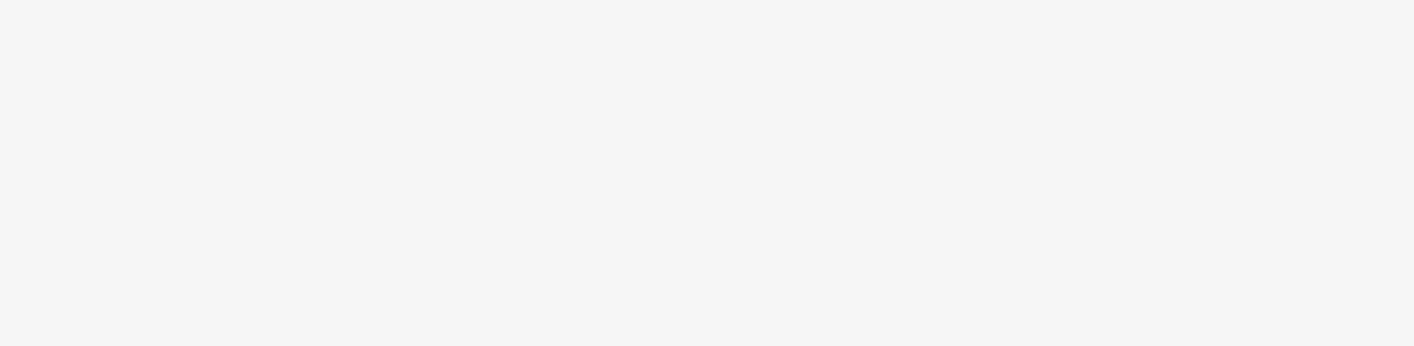
}

});

El resultado de cualquiera de los procesos es el mismo. Si creamos una nueva instancia sin establecer la edad

ex prop: <Nombre de la persona = "Bill" /> , El componente rendirá <Div> Bill (edad: desconocido) </ Div> .

Reaccionar apoyos mangos por defecto mediante la fusión de los puntales pasaron objeto y el objeto de los apoyos por defecto. Este proceso es similar a[Object.assign ()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign) o la Lodash / Underscore [\_.asignar()](https://lodash.com/docs#assign)método. El objeto apoyos por defecto es el objeto de destino y los puntales pasado es la fuente:



* Reaccionar código de la biblioteca para extraer defaultProps al constructor Si (Constructor.getDefaultProps) {

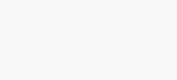
Constructor.defaultProps = Constructor.getDefaultProps ();

}

* pseudo código (como un ejemplo)

esta.props = Objeto.assign (Constructor.defaultProps, elementInstance.props);

En el fragmento de código Reaccionar, Reaccionar cheques la instancia de clase subyacente para ver si se define getDefaultProps () y utiliza esto para establecer los valores. Al utilizar clases ES6 lo único que definimos en la propia clase. Cualquier propiedad definida en elpassedProps Se aplica el valor / reemplaza a la propiedad en el objeto predeterminado apoyos.

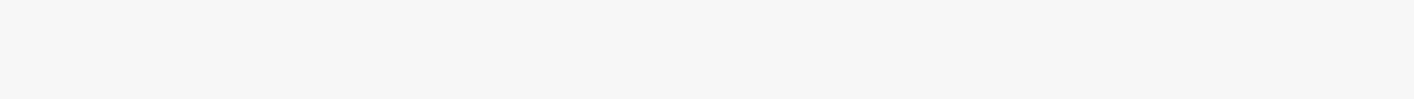


**vs nula indefinido accesorios**

Cuando el uso de accesorios por defecto, es importante entender cómo funciona el proceso de fusionar React. A menudo, estamos generando apoyos dinámicamente basándose en el estado de aplicación ([Flujo](https://facebook.github.io/flux/), [Redux](http://redux.js.org/), [MOBX](https://github.com/mobxjs/mobx), Etc.). Esto significa que a veces podemos generarnulo valores y pasan esto como la hélice.



En la asignación de apoyos por defecto, el código de fusión Reaccionar objeto ve nulo como un valor definido.



<Nombre de persona ="Chelín" edad = { nulo } />



14

Inicialización y Construcción



Porque nulo es un valor definido nuestro componente se rinde esto como <Div> Bob (edad :) </ Div> en lugar de la prestación *desconocido*. Sin embargo, si pasamos en indefinido en lugar de nulo Reaccionar trata esto como indefinido (bueno, sí, obviamente) y que haría que desconoce como se esperaba.



Tenga esto en cuenta a la hora de definir los apoyos por defecto, ya que el rastreo por una nulo valor puede ser difícil en la aplicación más grande.



**Estado inicial**

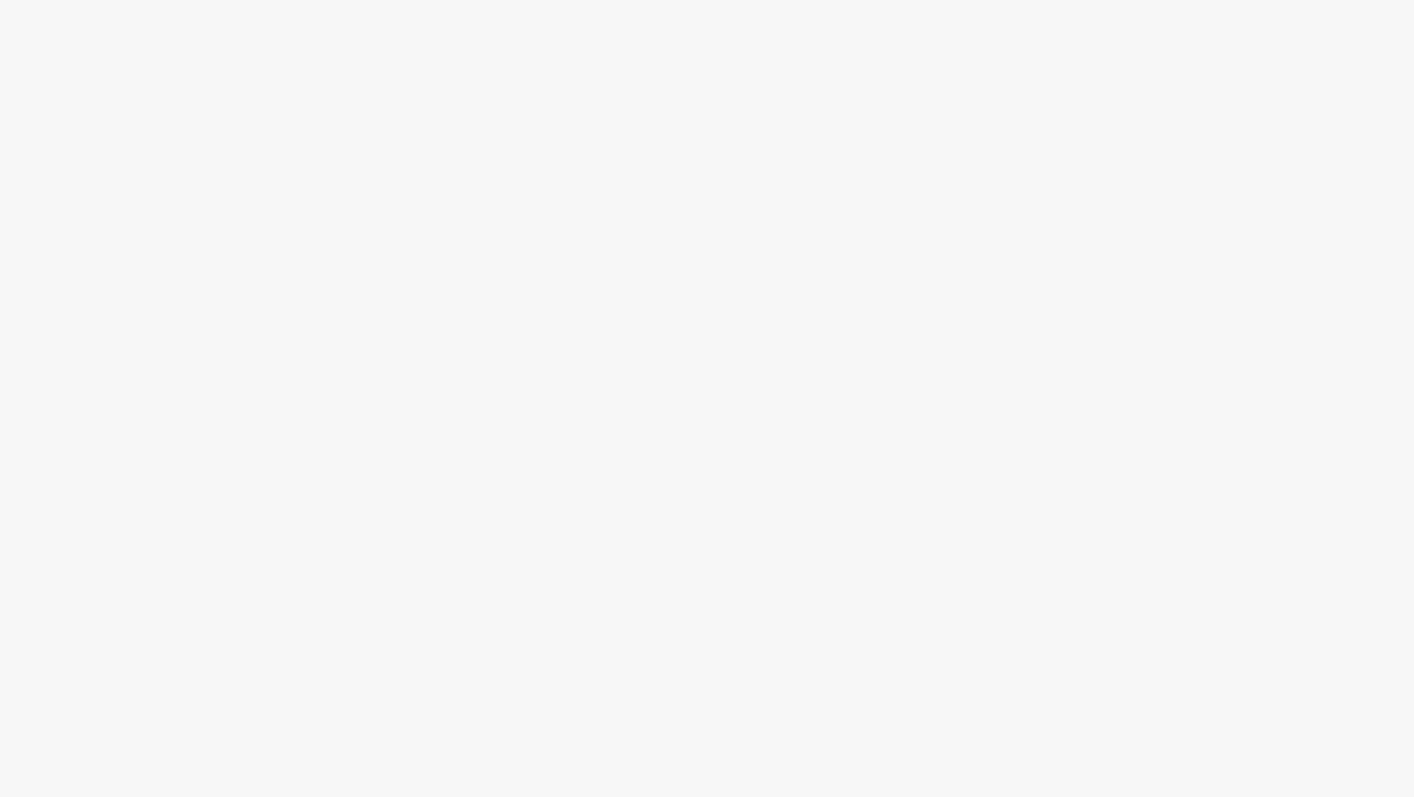
Una vez definidos los apoyos finales (pasado w / por defecto), la instancia del componente configura el inicial estado. Este proceso ocurre en la construcción de la propia instancia. A diferencia de los apoyos, el estado de componentes es un objeto interno que no está definido por los valores fuera.



Para definir el inicial estadodepende de cómo se declara su componente. Para ES6 declaramos el estado en el constructor. Al igual quedefaultProps , El estado inicial toma un objeto.



**Para ES6 Clase**



importar Reaccionar de 'reaccionar';

clase Persona se extiende Reaccionar.Componente {

constructor(apoyos) {

súper(accesorios);

esta.state = {count: 0 };

}

render () {

regreso (

<Div>{} This.props.name (edad: {} this.props.age)</ Div> );

}

}

Person.defaultProps = {edad: 'desconocido' };

por defecto de exportación Persona;

por React.createClass Componentes, existe un método de ayuda llamada getInitialState ()



que devuelve el objeto de estado. Este método se llama durante la instalación para establecer el estado de la

ejemplo.

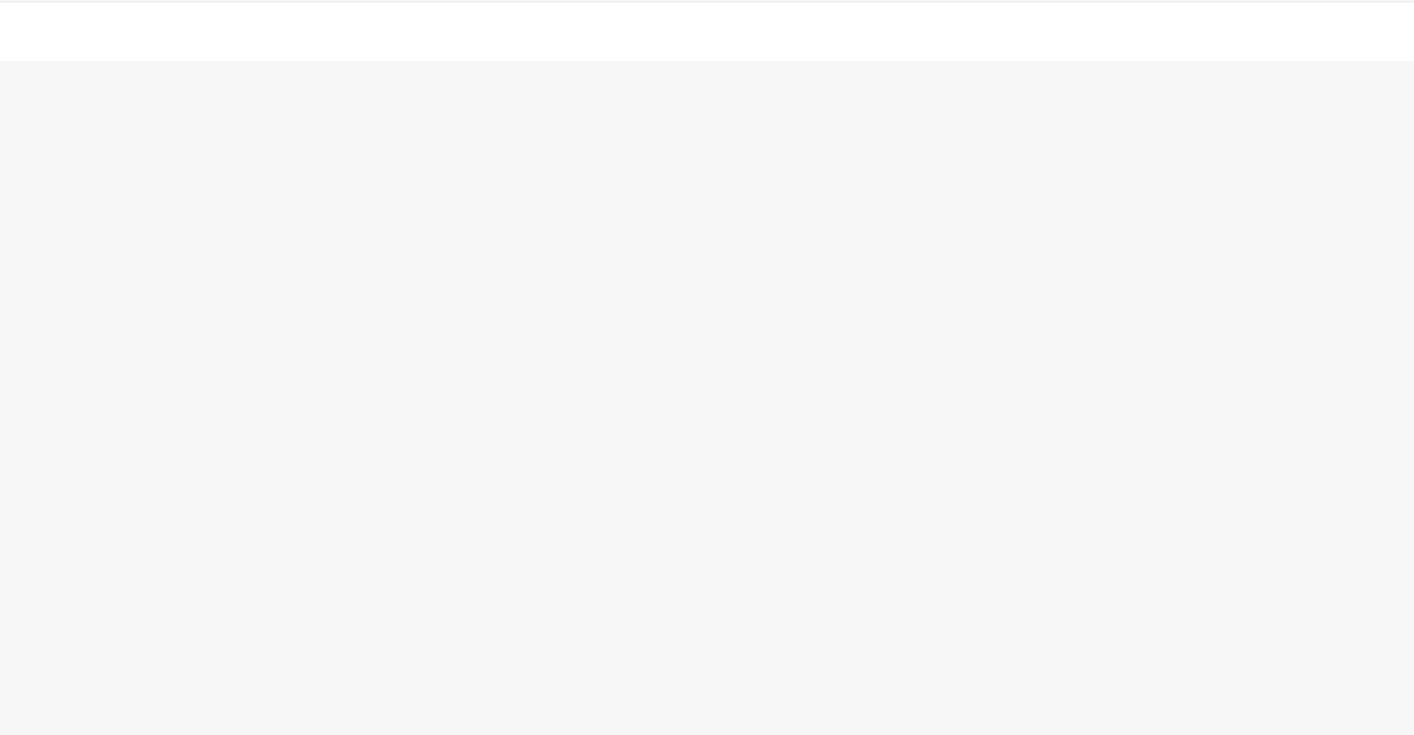
**Para createClass (ES6 / ES5 / CoffeeScript, etc.)**



15

{Ejem: null}

Inicialización y Construcción



var Persona = React.createClass ({

getDefaultProps: función() {

regreso ({ años: 'desconocido' });

},

getInitialState: función() {

regreso ({Count: 0 });

},

hacer: función() {

regreso (

<Div>{} This.props.name (edad: {} this.props.age)</ Div> );

}

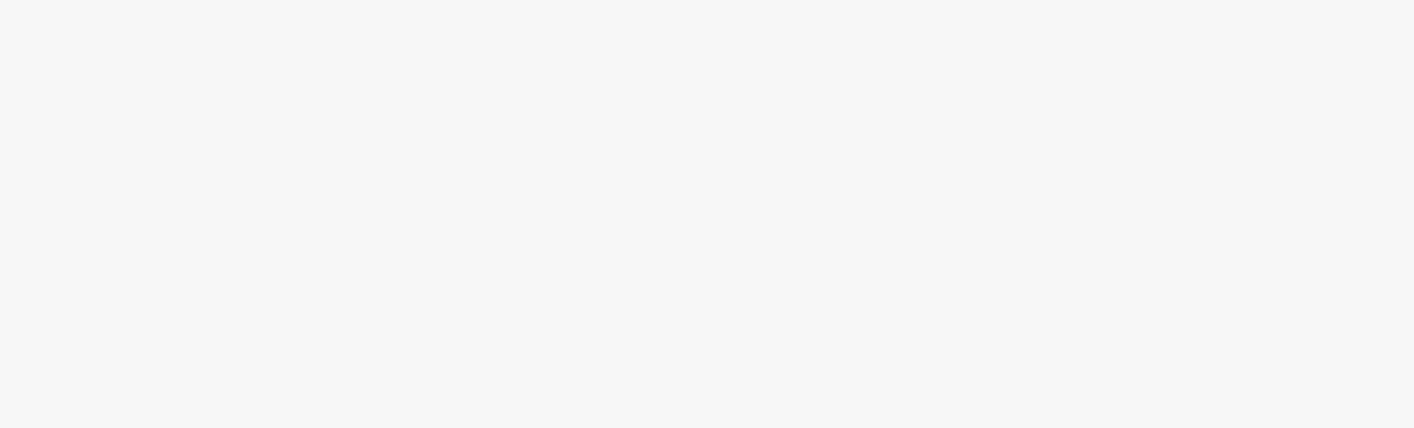
});

**incumplimientos del estado**

Es importante tener en cuenta que si no definimos un estado en el constructor / getInitialState entonces el estado será indefinido. Debido a que el estado esindefinido y no un objeto vacío ( {} ), Si intenta consultar el estado más adelante este será un problema.



En general, queremos establecer un valor predeterminado para todas las propiedades estatales. Hay algunos casos extremos donde el valor inicial de la propiedad estatal puede sernulo o indefinido. Si este estado pasa a ser única propiedad estatal, puede ser tentador para omitir el establecimiento de un estado predeterminado. Sin embargo, si nuestro código intenta acceder a la propiedad obtendrá un error.



clase Persona se extiende Reaccionar.Componente {

render () {

* Esta declaración generará un error consola.Iniciar sesión(esta.state.foo);

regreso (

<Div>{} This.props.name (edad: {} this.props.age)</ Div> );

}

}

La declaración de registro falla porque este estadoes indefinido. Cuando tratamos de accesofooobtendremos un TypeError: No se puede leer la propiedad 'foo' de null. Para solucionar esto podemos establezca la



por defecto el estado de {} o bien, tener una intención clara, configurarlo para .

***Lo siguiente:*** [Pre-montaje concomponentWillMount ()](#page17)

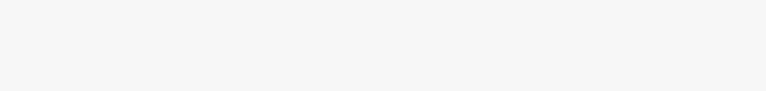


dieciséis

Pre-montaje con componentWillMount ()



**Pre-montaje con** **componentWillMount ()**



Ahora que los accesorios y el estado se establecen, por fin entramos en el terreno de los métodos de ciclo de vida. El primer método de ciclo de vida verdadera es llamadacomponentWillMount (). Este método sólo se llama una vez, que es antes de la primera representación. Dado que este método se llama anteshacer()Nuestro componente no tendrá acceso a la interfaz de usuario nativa (DOM, etc.). Asimismo, no tendremos acceso a los niñosrefs , Porque no se crean todavía.



los componentWillMount ()Es una oportunidad para nosotros para manejar la configuración, actualizar nuestro estado, y en general preparamos para el primer render. En este punto, se definen apoyos y estado inicial. Podemos consultar de forma segurathis.props y este estado, Sabiendo con certeza que son los valores actuales. Esto significa que podemos comenzar a realizar cálculos o procesos en base a los valores de prop.

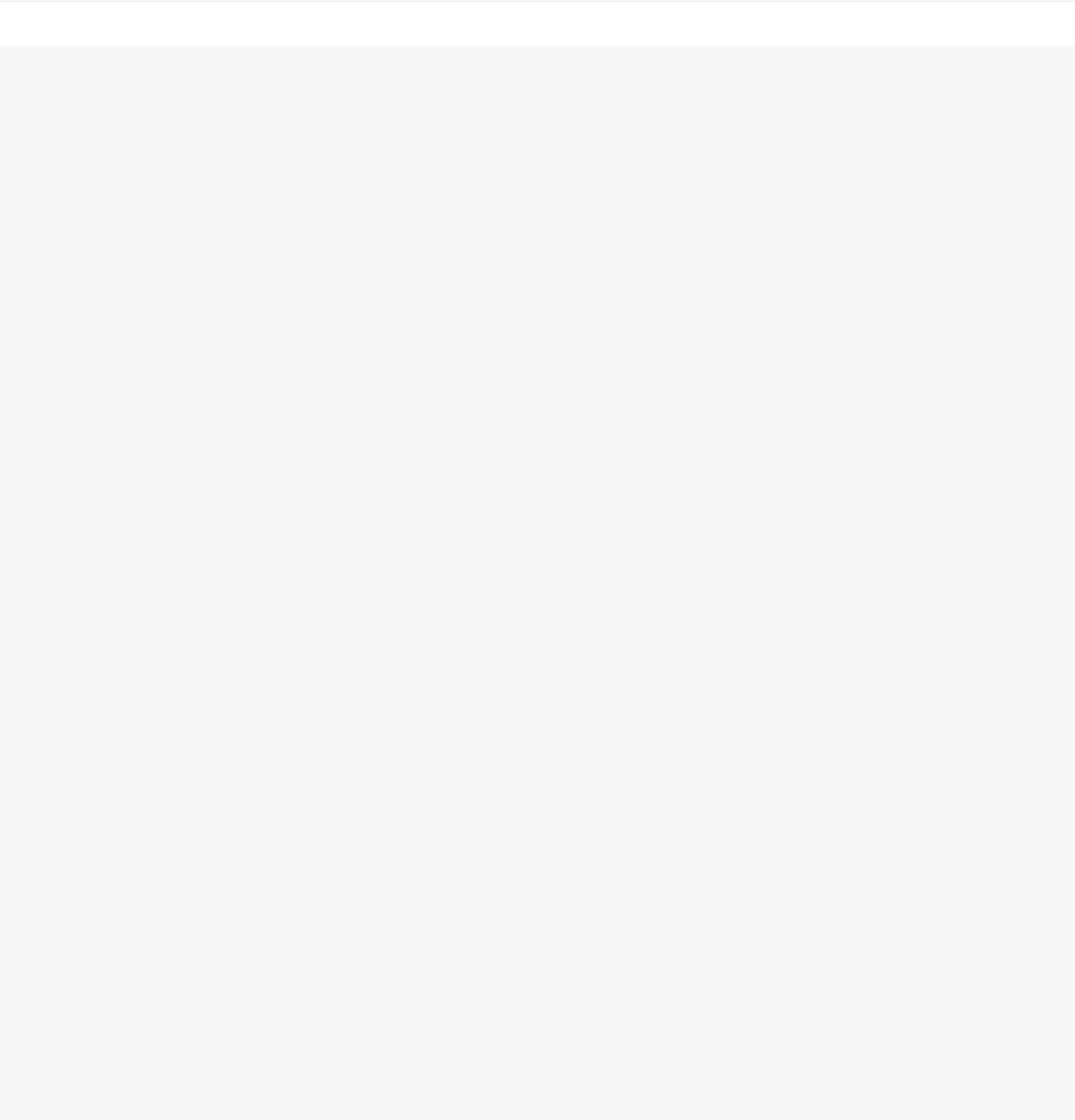


**Person.js** [**1**](#page17)



17

Pre-montaje con componentWillMount ()



importar Reaccionar de 'reaccionar';

importar classnames de '' classnames;

clase Persona se extiende Reaccionar.Componente {

constructor(apoyos) {

súper(accesorios);

esta.state = {modo: indefinido };

}

componentWillMount () {

dejar modo;

Si (this.props.age> 70) {

mode = 'antiguo';

} else if (esta.props.age < 18) {Mode = 'joven';

} más {

mode = 'medio';

}

esta.setState ({modo});

}

render () {

regreso (

<Div className ={ classnames ( 'persona', this.state.mode)}>

{} This.props.name (edad: {} this.props.age) </ Div>

);

}

}

Person.defaultProps = {edad: 'desconocido' };

por defecto de exportación Persona;

En el ejemplo anterior que llamamos this.setState ()y actualizar nuestro estado actual antes de render. Si necesitamos valores de estado en cálculos aprobadas enaccesorios , Aquí es donde debemos hacer la lógica.



Otros usos para componentWillMount ()incluye el registro de eventos globales, como una tienda de Flujo. Si su componente tiene que responder a los eventos de interfaz de usuario nativos globales, tales comoventana cambiar el tamaño o cambios del foco, este es un buen lugar para hacerlo[2](#page17).



***Lo siguiente:*** [Componentehacer()](#page20)



* En nuestro ejemplo anterior, estamos utilizando el nombres de las clases ()biblioteca, que se incluyó originalmente como un complemento de React. Sin embargo, la característica se ha quitado de reaccionar y se trasladó a su[propia biblioteca](https://github.com/JedWatson/classnames) para su uso con o sin React.

2



18

Pre-montaje con componentWillMount ()



* Es importante recordar que no existen muchos elementos de la interfaz nativa en este punto en el ciclo de vida. Eso significa que necesitamos que se adhieren a muy alto nivel / eventos globales comoventana o



documento .

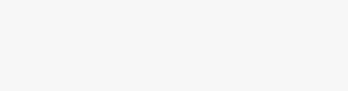


19

Componente render ()



**Componente** **hacer()**



Ahora que hemos pre-configurado nuestro componente, entramos en la primera versión de nuestro contenido. Como Reaccionar a los desarrolladores, lashacer()método es el más familiar. Creamos elementos (generalmente a través de JSX) y devolverlos. Nos acceder al componentethis.props y este estado y dejar que estos valores se derivan cómo el contenido debe ser generada. Cuando accedemos este estado , Cualquier cambio que hicimos durante componentWillMount () se aplican plenamente.

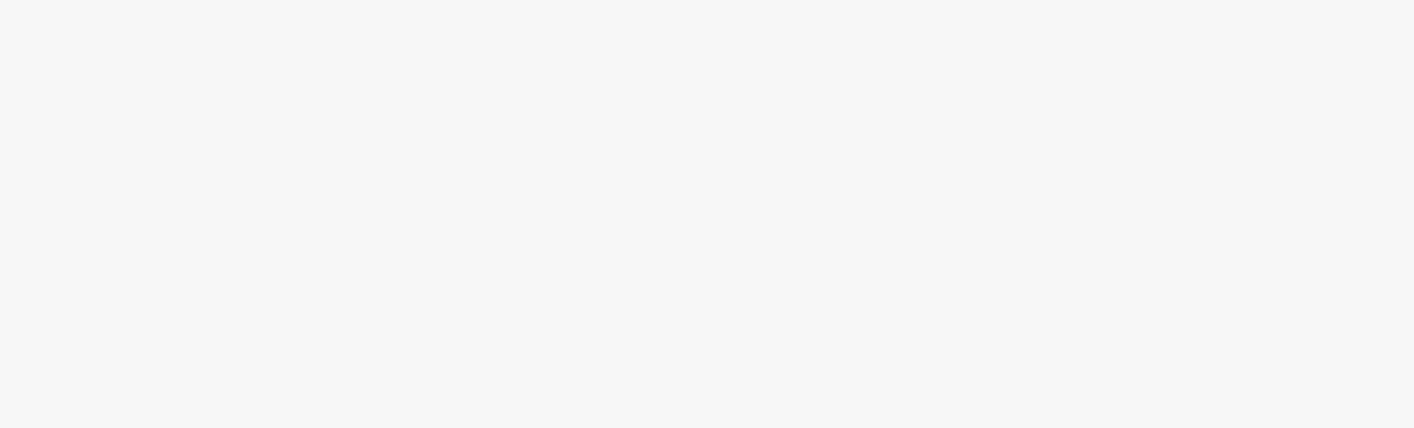


A diferencia de cualquier otro método en el Ciclo de Vida, hacer()es el método que existe a través de múltiples fases del ciclo de vida. Se produce aquí en nacimiento y es donde pasamos mucho tiempo en el crecimiento.

En ambos casos, tenemos el principio básico de mantener hacer()un método puro. Qué significa eso? Eso significa que no hay que llamarsetstate (), Consultar la interfaz de usuario nativa o cualquier otra cosa que pueda mutar el estado actual de la aplicación. La razón por la que hacemos es si este tipo de interacción enhacer(), Entonces será otra patada de salida Representar texto. Que una vez más, los disparadores

hacer() que a su vez hace lo mismo ... infinitamente.

El modo de desarrollo Reaccionar[1](#page20) es generalmente grande en la captura de estos tipos de errores y se gritarte si las hacen. Por ejemplo, si hemos hecho algo tonto como este



render () {

// MALO: No haga esto!

esta.setState ({foo: 'bar' });

regreso (

<Div className ={ classnames ( 'persona', this.state.mode)}>

{} This.props.name (edad: {} this.props.age) </ Div>

);

}

Reaccionar que ingrese a la siguiente declaración:



Advertencia: setstate (...): No se puede actualizar durante una transición de estado existente (tal como dentro de hacer ). Render métodos deben ser una función pura de utilería y estado.



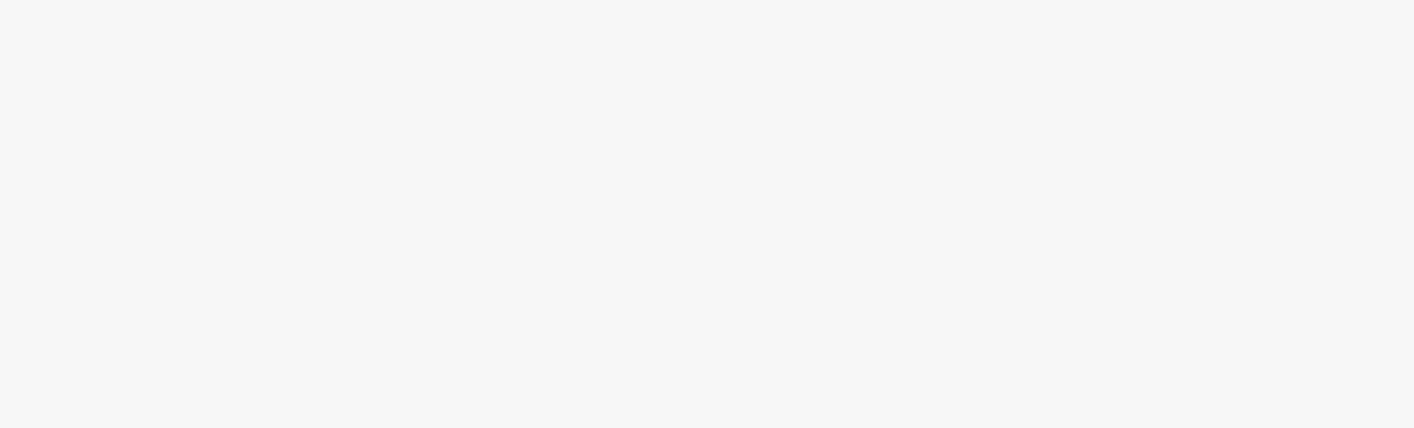
**Acceso a la interfaz de usuario nativa en** **hacer()** **es a menudo fatal**

Reaccionar también le avisará si se intenta acceder a los elementos de interfaz de usuario nativos en el paso de procesamiento.



20

Componente render ()



render () {

// MALO: No hacer esto, ya sea!

dejar nodo = ReactDOM.findDOMNode (esta);

regreso (

<Div className ={ classnames ( 'persona', this.state.mode)}>

{} This.props.name (edad: {} this.props.age) </ Div>

);

}



VM943: 45 Advertencia: La persona está accediendo getDOMNode o findDOMNode el interior de su render (). render () debe ser una pura función de utilería y estado. Nunca debe tener acceso a algo que requiere datos antiguos de los últimos render, tales como árbitros. Mueva esta lógica a componentDidMount y componentDidUpdate lugar.

En el ejemplo anterior, puede parecer seguro, ya que se acaba de Consulta del nodo. Pero, como los estados de alerta, podríamos consultar datos potencialmente viejos. Pero en nuestro caso, durante la fase de nacimiento, esto sería un error fatal.



Uncaught Violación invariante: findComponentRoot (..., 0.0): No se pudo encontrar elemento. Esto probablemente significa el DOM se mutó de forma inesperada (por ejemplo, por el navegador), generalmente debido a un olvido <tbody> utilizando las tablas, las etiquetas de anidación como <form>, <p>, ORa, o utilizando elementos no SVG en un <svg> padres. Trate de inspeccionar los nodos secundarios del elemento con Reaccionar IDPersona .



Este es uno de esos casos en los que reaccionan de error no apunta claramente a la causa del problema. En nuestro caso no modificamos el DOM, por lo que se siente como un error claro y puede inducir a error. Este tipo de error puede causar Reaccionar a los desarrolladores una gran cantidad de dolor desde el principio. Debido a que instintivamente buscamos un lugar en el que estamos cambiando la interfaz de usuario nativa.

La razón por la que sale este error se debe a que durante el primer Representar los elementos de interfaz de usuario nativos que está intentando acceder no existe todavía. Estamos pidiendo esencialmente reaccionan a encontrar un nodo DOM que no existe. Generalmente, cuandoReactDOMno puede encontrar el nodo, esto se debe a algo o alguien mutaron el DOM. Así, Reaccionar cae de nuevo a la causa más común.

Como se puede ver, tener una comprensión del ciclo de vida puede ayudar a solucionar y prevenir estos problemas a menudo no-intuitivas.

***Hasta la próxima:*** [Administración de componentes de montaje y niños](#page23)



* [Estas advertencias salen del modo de desarrollo en React. También puede utilizar elReaccionar](https://github.com/facebook/react-devtools) [Herramientas de desarrollo para ayudar a depurar y explorar Reaccionar componentes.](https://github.com/facebook/react-devtools)



21

Componente render ()



22

componentWillMount ()

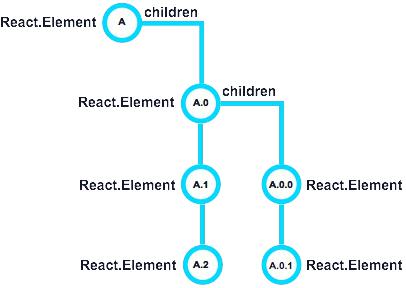
el camino hacia abajo.

La gestión de la Infancia y de componentes de montaje



**Administración de componentes de montaje y niños**

Ahora que hemos completado el primer paso render, nuestra hacer()método devuelve un solo Reaccionar Element. Este elemento puede tener elementos secundarios. Esos niños también pueden tener hijos, y así sucesivamente.



Con la posibilidad de un árbol de profundidad n de elementos, cada uno de los elementos tienen que ir a través de su propio proceso de todo el ciclo vital. Al igual que el padre del elemento, Reaccionar crea una nueva instancia para cada niño. Pasan a través de la construcción, puntales defecto, estado inicial,

y hacer() . Si el niño tiene niños, el proceso comienza de nuevo ... todo

Uno de los conceptos más poderosos en reaccionamos es la capacidad de componer fácilmente diseño complejo a través de anidación de los niños. Se animó a seguir sus componentes como 'muda' como sea posible. La idea es tener solamente contenedores[1](#page23) Administración de componentes funcionalidad de nivel superior.

Debido a que esta es la manera preferida de desarrollo, esto significa que tendremos una gran cantidad de componentes más pequeños que también tienen su propio ciclo de vida. Tenga esto en cuenta a medida que continuamos a través del ciclo de vida, ya que cada componente puede seguir el mismo patrón.

***Hasta la próxima:*** [Post-montaje concomponentDidMount ()](#page25)



* Ver [Componentes de presentación y contenedores](https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0#.pnmirdrso) por Dan Abramov para más detalles



23

La gestión de la Infancia y de componentes de montaje

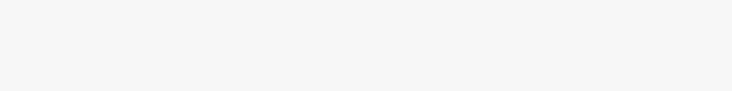


24

Post-montaje con componentDidMount ()



**Post-montaje con** **componentDidMount ()**

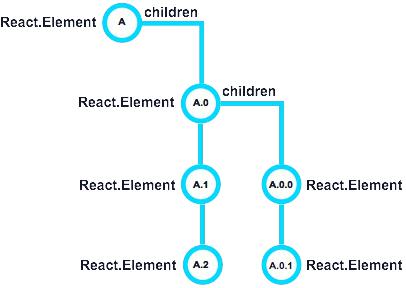


El último paso en la fase de ciclo de vida / Monte nacimiento es nuestro puesto de montaje en el acceso a través componentDidMount () . Este método se llama una vez Elementos todos nuestros hijos y nuestros instancias de componentes se montan sobre el Native UI. Cuando se llama a este método ahora tenemos acceso a la interfaz de usuario nativa (DOM, UIView, etc.), el acceso a nuestros hijosrefs y la capacidad de desencadenar potencialmente un nuevo pase de render.

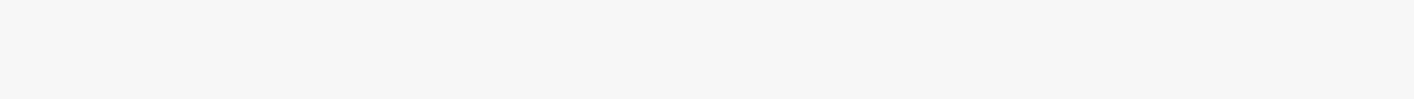


**Para la comprensión de llamadas**

Similar a componentWillMount () , componentDidMount ()Sólo se llama una vez. A diferencia de los otros métodos de montaje / nacimiento, en el que comienzan en la parte superior y trabaje hacia abajo,componentDidMount ()funciona de abajo hacia arriba. Vamos a considerar lo siguiente / árbol Componente elemento nuevo:

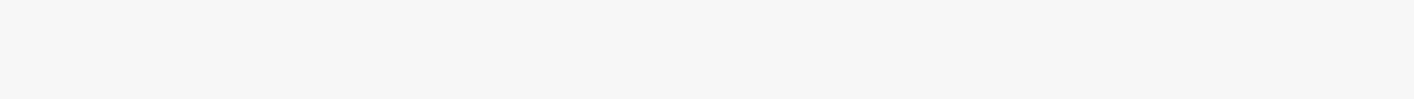


Cuando comenzamos la fase de nacimiento, procesamos hacer() en este orden:



A -> A.0 -> A.0.0 -> A.0.1 -> A.1 -> A.2.

Con componentDidMount () empezamos por el final y trabajamos nuestro camino de regreso.



A.2 -> A.1 -> A.0.1 -> A.0.0 -> A.0 -> A

Al caminar hacia atrás, sabemos que cada niño ha montado y ejecutar su propio componentDidMount () . Esto garantiza que el padre puede acceder a los elementos de la interfaz nativa para

en sí y sus hijos.



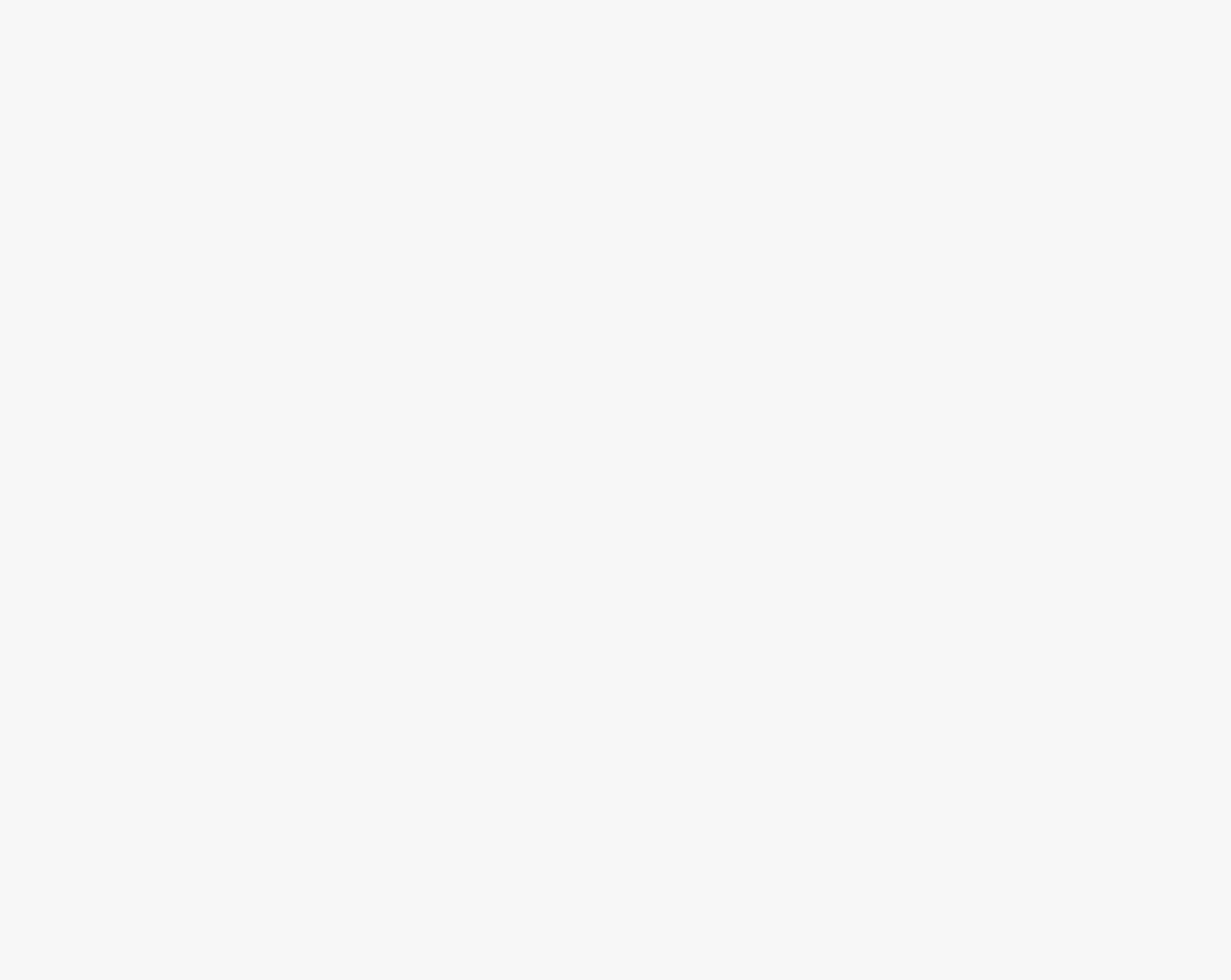
25

Post-montaje con componentDidMount ()



Vamos a considerar los siguientes tres componentes y su orden de llamada.

**GrandChild.js**



/ \*\*

* Nieto
* Se registra el componentDidMount () y tiene un método público denominado valor. \* /

importar Reaccionar de 'reaccionar';

importar ReactDOM de 'Reaccionar-dom';

clase predeterminada de exportación Nieto se extiende Reaccionar.Componente {

componentDidMount () {

consola.Iniciar sesión('Nieto hizo montaje.);

}

valor () {

regreso ReactDOM.findDOMNode (esta.refs.input) .value;

}

render () {

regreso (

<Div>

Nieto

<Input ref ="entrada" type ="texto" defaultValue ="Foo" /> </ Div>

);

}

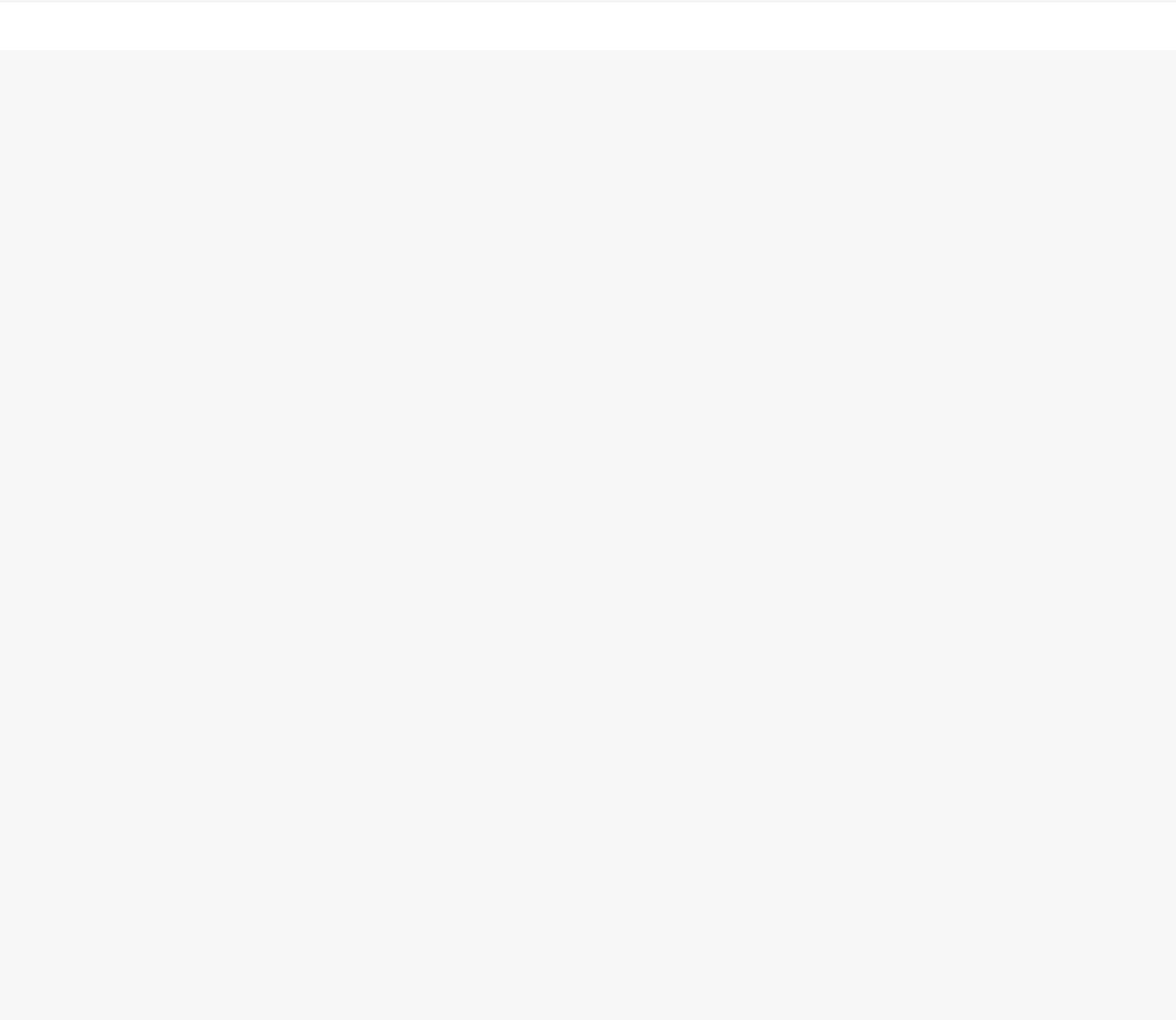
}

**Child.js**



26

Post-montaje con componentDidMount ()



/ \*

* Niño
* Se registra el componentDidMount () y tiene un método público denominado valor,
* que devuelve el valor nieto.

\* /

importar Reaccionar de 'reaccionar';

importar Nieto de './Nieto';

clase predeterminada de exportación Niño se extiende Reaccionar.Componente {

componentDidMount () {

consola.Iniciar sesión('Niño hizo montura.');

}

valor () {

devolver este.refs.grandChild.value ();

}

render () {

regreso (

<Div>

Niño

<Nieto ref ="nieto" />

</ Div>

);

}

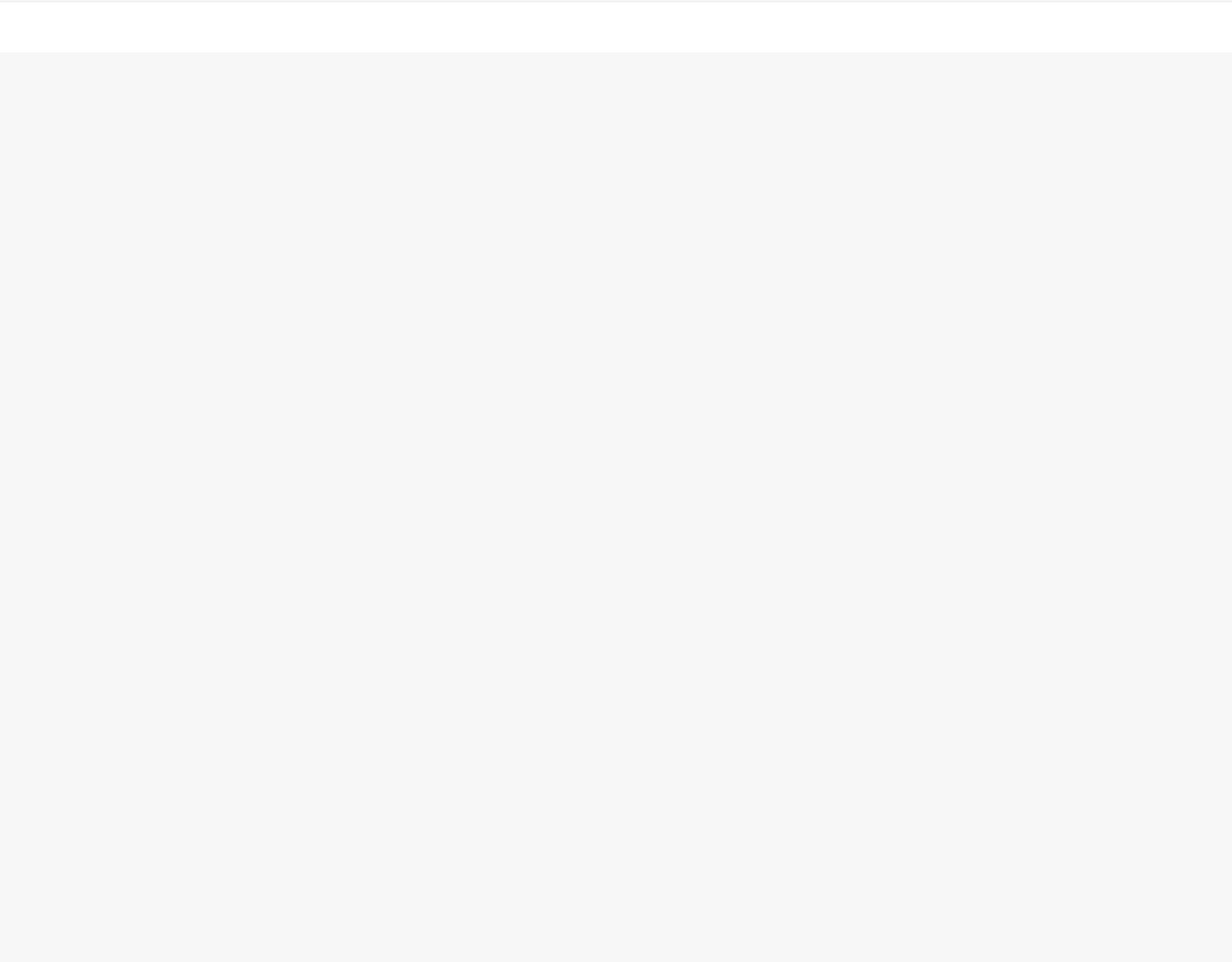
}

**Parent.js**



27

Post-montaje con componentDidMount ()



/ \*

* Padre
* Se registra el componentDidMount () y luego registra el valor hijo ()
* método.

\* /

importar Reaccionar de 'reaccionar';

importar Niño de './Niño';

clase predeterminada de exportación Padre se extiende Reaccionar.Componente {

componentDidMount () {

consola.Iniciar sesión('Padre hizo montura.');

consola.Iniciar sesión('Valor Niño:', esta.refs.child.value ());

}

render () {

regreso (

<Div>

Padre

<Ref = Niño"niño" />

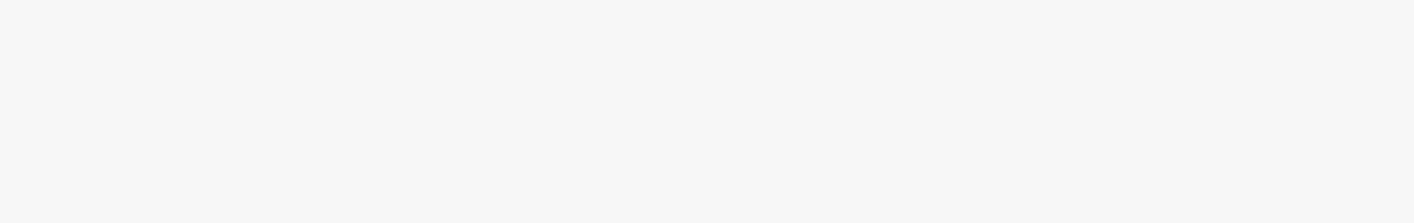
</ Div>

);

}

}

Cuando montamos <Padre /> en nuestra aplicación obtenemos el siguiente en la consola del navegador:



Nieto hizo montura.

Niño hizo montura.

Padres hizo montura.

Valor niño: foo

Como se puede ver, de los nietos componentDidMount ()fue llamado en primer lugar, seguido de niño y luego los padres. Debido a que estamos ahora montado en el DOM y nuestros hijos creados, el padre puede acceder a surefs y el nieto puede tener acceso a sus propios nodos DOM.



**tareas útiles**

los componentDidMount ()método puede ser un levantador pesado útiles para nuestros componentes. Una de las tareas más comunes es la interacción con la interfaz de usuario nativa. diferente acomponentWillMount () o hacer() ahora podemos interactuar plenamente con la pila nativa.

Por ejemplo, es posible que tenga que realizar cambios en nuestro estado actual basado en la forma en que el nativo de interfaz de usuario diseñada nuestro contenido. Es posible que tengamos que averiguar la anchura actual / altura de nuestros hijos o nuestra propia instancia. Esto es especialmente útil en el navegador en el diseño CSS maneja una gran cantidad de



28

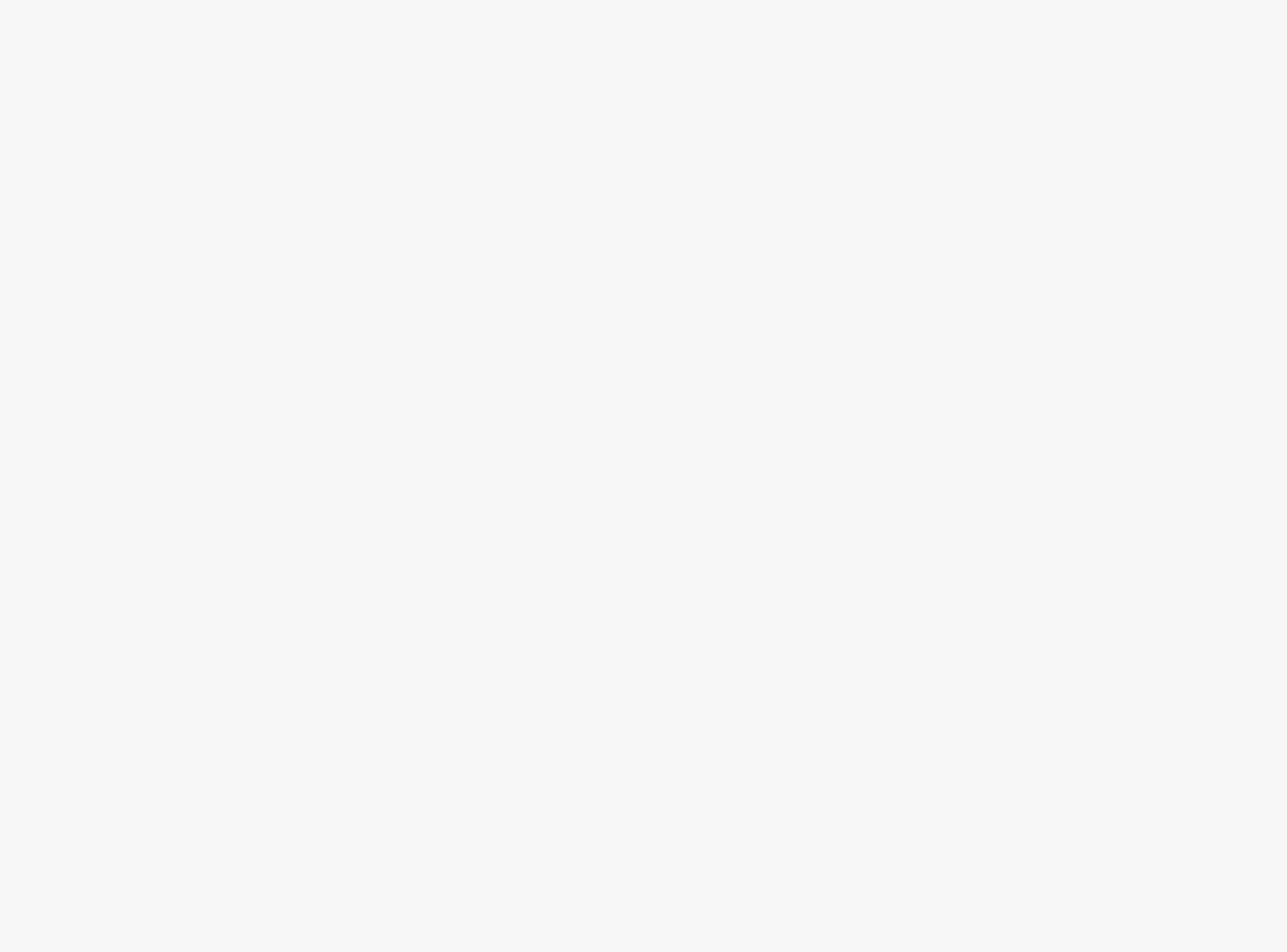
Post-montaje con componentDidMount ()



nuestros cálculos DOM.

Otra tarea útil es la creación de interfaces de usuario tercera parte. Por ejemplo, si se quiere utilizar una biblioteca como[C3.js](http://c3js.org/) o el [Selector de rango fecha](http://www.daterangepicker.com/), Aquí es donde nos inicializar las bibliotecas de interfaz de usuario.

**Chart.js**



importar Reaccionar de 'reaccionar';

importar ReactDOM de 'Reaccionar-dom';

importar c3 de 'C3';

clase predeterminada de exportación Gráfico se extiende Reaccionar.Componente {

componentDidMount () {

esta.chart = c3.generate ({

bindto: ReactDOM.findDOMNode (esta.refs.chart),

Datos: {

columnas: [

['Datos1', 30, 200, 100, 400, 150, 250],

['Data2', 50, 20, 10, 40, 15, 25]

]

}

});

}

render () {

regreso (

<Div ref ="gráfico"> </ Div>

);

}

}

En el ejemplo anterior, el apalancamiento componentDidMount () para generar nuestra tabla, se unen a la DOM utilizando refs y luego pasar en los datos.



Al integrar las bibliotecas 3 ª parte, a menudo necesitamos para unirse a eventos, como el usuario interactúa con el Plan General. Aquí es donde nos gustaría establecer nuestros oyentes de inicialización después de la biblioteca. También podemos añadir detectores más globales aquí, si no queríamos para configurar los oyentes en elcomponentWillMount () llamada.

**Iniciar otro Representar** [**1**](#page25)

Hay algunas situaciones únicas en las que podemos tener una segunda rinden inmediatamente después del nacimiento / Monte. Esto no es una situación común y por lo general ocurre cuando tenemos que cambiar nuestro estado actual, basado en la interfaz de usuario de diseño nativo. Esta fila dinámica se podría calcular



29

Post-montaje con componentDidMount ()



altura o ancho de las columnas en una tabla de datos. Se podría tener que volver a posicionar los niños del componente basado en la forma en que están dimensionados primera vez.

Si necesita este tipo de funcionalidad, usted tiene la posibilidad de llamar this.setState () o forceUpdate () en tus componentDidMount () . Si cambia de estado o forzar una actualización (más en esta función más adelante), el componente comenzará otra Representar y escriba el [Crecimiento / fase de actualización](#page31). Porque componentDidMount () se llama una sola vez, no tenemos a preocuparse por este método causando un bucle infinito. Sin embargo, este proceso puede conducir a problemas en el futuro si no se toman el tiempo para caminar por todas las ramificaciones potenciales de múltiples rinde.



***Hasta la próxima:*** [Crecimiento / actualización de fase en Profundidad](#page31)



* Render múltiples pases se abre la puerta a serios problemas de rendimiento. Proceder con extrema precaución!



30

Crecimiento / actualización en profundidad



**Crecimiento / actualización en profundidad**

Una vez que nuestro componente se monta en la fase de nacimiento, estamos preparado y listo para la fase de crecimiento / actualización. La fase de crecimiento es donde un componente pasa la mayor parte de su tiempo. Aquí tenemos actualizaciones de datos, actúan sobre las acciones del usuario o del sistema y proporcionar la experiencia en general para nuestra aplicación.

La fase de crecimiento se dispara en tres formas diferentes: el cambio de accesorios , Cambiando de estado o llamando forceUpdate () . Los cambios que se realizan afectan la manera en la fase de actualización es manejado. Vamos a discutir cada uno de estos cambios en la profundidad a medida que caminamos a través de todo el proceso de crecimiento.



En esta sección, nos sumergiremos en los diferentes métodos. Vamos a examinar el orden de los métodos llamados y cómo afectan al proceso global. También discutiremos qué tareas se manejan mejor en cada método y discutir la optimización de aplicaciones.

**A partir Actualizar: Cambio de Apoyos**

Como se mencionó anteriormente, tenemos tres formas de iniciar la fase de crecimiento / actualización. La primera es cuando el componente deaccesoriosactualizar. Esto ocurre cuando el elemento raíz (por ejemplo:



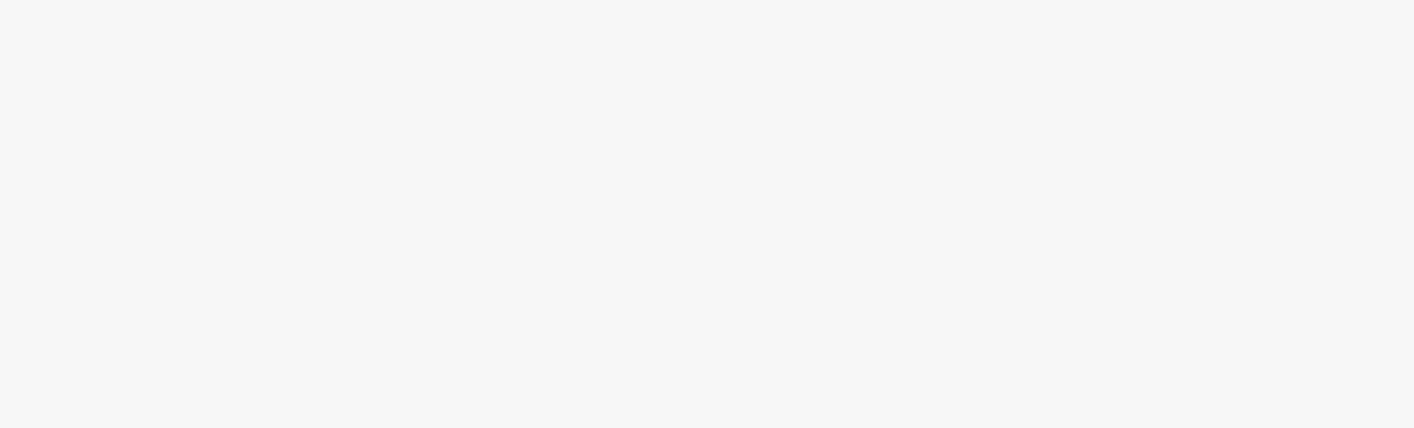
ReactDOM.render (<MyComponent datos = {} dataVaule />, ...) tiene el accesorios valor cambiado



o el padre del componente de apuntalar cambios.



Desde la perspectiva de un ejemplo de componentes (por ejemplo, <Nombre de la persona = "Bill" />) El pasado en los apoyos son inmutables. En otras palabras, la instancia de persona no puede actualizar el nombre del valor internamente. De hecho, si se intenta obtendrá un error en React.



render () {

* MALO: No hagas esto! esta.props.name = 'Tim'; regreso (

<Div className ={ classnames ( 'persona', this.state.mode)}> {} This.props.name (edad: {} this.props.age)

</ Div>

);

}



TypeError: No se puede asignar a propiedad de sólo lectura 'nombre' del objeto '# <objeto>'

Debido a que los apoyos son inmutables por el propio componente, el padre debe proporcionar los nuevos valores. Cuando los nuevos apoyos se pasan a través de la raíz o el padre, esto comienza la fase de actualización.

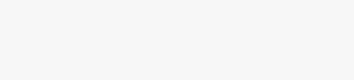


31

Crecimiento / actualización en profundidad



**A partir de actualización:** **setstate ()**



Similar a cambiar accesorios , Cuando un componente cambia su valor de estado[1](#page31) vía this.setState () , Esto también provoca una nueva fase de actualización. Para muchos de los desarrolladores Reaccionar, las primera gran (y para ser honesta en curso) reto es gestionar el estado de los componentes. Estado mismo puede ser un tema controvertido en la comunidad. Muchos desarrolladores de evitar a toda costa el estado. Otros sistemas, tales como[MOBX](http://mobxjs.github.io/mobx/), Son, en esencia, tratando de reemplazarlo. Muchos usos de estado pueden caer en diferentes anti-patrones, tales como la transferencia deaccesorios dentro estado [2](#page31).



Fundamentalmente, el estado puede ser un tema complicado y confuso. ¿Cuándo usamos Estado? ¿Qué datos deben o no deben ser almacenados en el estado? Deberíamos incluso utilizar el estado en absoluto? Para ser honesto, este es un tema que todavía estamos tratando de lidiar con nosotros mismos.

Teniendo esto en mente, todavía es importante entender cómo funciona el estado en reaccionar. Vamos a seguir para analizar en profundidad el estado y cómo los mecánicos trabajan. Vamos a tratar de compartir las mejores prácticas que hemos encontrado, pero en general lo que es bueno hoy probablemente será malo mañana.

**La asincronía de Estado**

Antes de pasar a la última forma de iniciar una actualización, deberíamos hablar un poco acerca de cómo se gestiona el estado en la parte interna de React. Cuando los desarrolladores por primera vez elsetstate () se da por supuesto que cuando se llama este estadolos valores que se aplican en el set estarán disponibles. Esto no es verdad. lossetstate () método debe tratarse como un proceso asíncrono [3](#page31). Entonces, ¿cómosetstate () ¿trabajo?

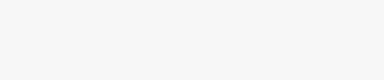


Cuando llamamos setstate ()esto se considera un cambio de estado parcial. No estamos Flushing / sustitución de todo el estado, simplemente actualización de la parte (s) de la misma. Reaccionar utiliza un sistema de colas[4](#page31)para aplicar el cambio de estado parcial. Debido a que podemos establecer el estado varias veces en una cadena de método, una cola de cambio está construido para gestionar todos los diversos cambios. Una vez que se añade el cambio de estado de la cola, Reaccionar se asegura se añade el componente a la cola sucia. Esta cola sucia seguimiento de las instancias de componentes que han cambiado. En esencia, esto es lo que dice Reaccionar qué componentes necesita para entrar en la fase de actualización más tarde.

Cuando se trabaja con el estado, es muy importante tener esto en cuenta. Un error común es fijar estado en un método y más tarde en la misma cadena método sincrónico tratar de acceder al valor de estado. Esto a veces puede causar errores difíciles, especialmente si expone los valores de estado a través de los métodos públicos de su componente, tales comovalor(). Vamos a hablar más tarde de cuandoeste estado está finalizado.



**A partir de actualización:** **forceUpdate**



32

Crecimiento / actualización en profundidad



Hay otra manera de comenzar una fase de actualización. Hay un método especial en un componente llamadoforceUpdate (). Esto hace exactamente lo que piensa, que obliga al componente en una fase de actualización. losforceUpdate () método tiene algunas ramificaciones específicas acerca de cómo se procesan los métodos de ciclo de vida y vamos a discutir esto en profundidad más adelante.

Hasta la próxima: [actualización y](#page34) [componentWillReceiveProps ()](#page34)



* Con el estado de componentes, consideramos que esta funcionalidad interna. En teoría, podemos acceder e incluso editar estado fuera de la estancia, pero esto es un anti-patrón. Accediendo el estado de un componente desde fuera inyecta una gran cantidad de fragilidad en el sistema (encauzamiento de la dependencia, el cambio de los valores internos, etc.). Sólo una instancia de componente debesetstate () en sí mismo.
* A pesar de que se mueve accesorios a estado [se considera un anti-patrón](https://facebook.github.io/react/tips/props-in-getInitialState-as-anti-pattern.html) hay algunos casos de uso. El más común es tener unavalor por defecto puntal que se convierte en el interior valor



en el estado. Vemos este patrón con la mayoría de los elementos del formulario en React; aunque existe un fuerte movimiento para alejarse de esto y trabajar con solamente[componentes controlados](https://facebook.github.io/react/docs/forms.html#controlled-components).

* Los comentarios de código Reaccionar recomiendan que" ... Debe tratar *este estado*como inmutables. No hay ninguna garantía de que*este estado se actualizará de inmediato, por lo que el acceso a este estado después de llamar a [la setstate] método puede devolver el valor antiguo ".*



[4](#page31) Reaccionar método interno enqueueSetState () para ser exacto.

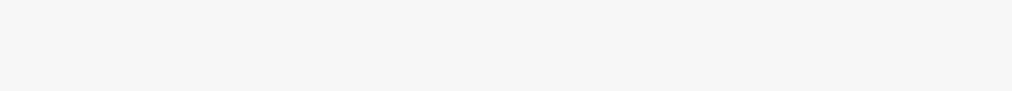


33

Actualización y componentWillReceiveProps ()



**actualización y**

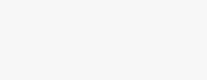


**componentWillReceiveProps ()**

Ahora que hemos discutido a partir de una actualización, vamos a bucear en los métodos de ciclo de actualización de la vida. El primer método disponible para nosotros escomponentWillReceiveProps (). Este método se llama cuandoaccesoriosse pasan a la instancia de componente. Vamos a profundizar un poco más en lo que esto significa.



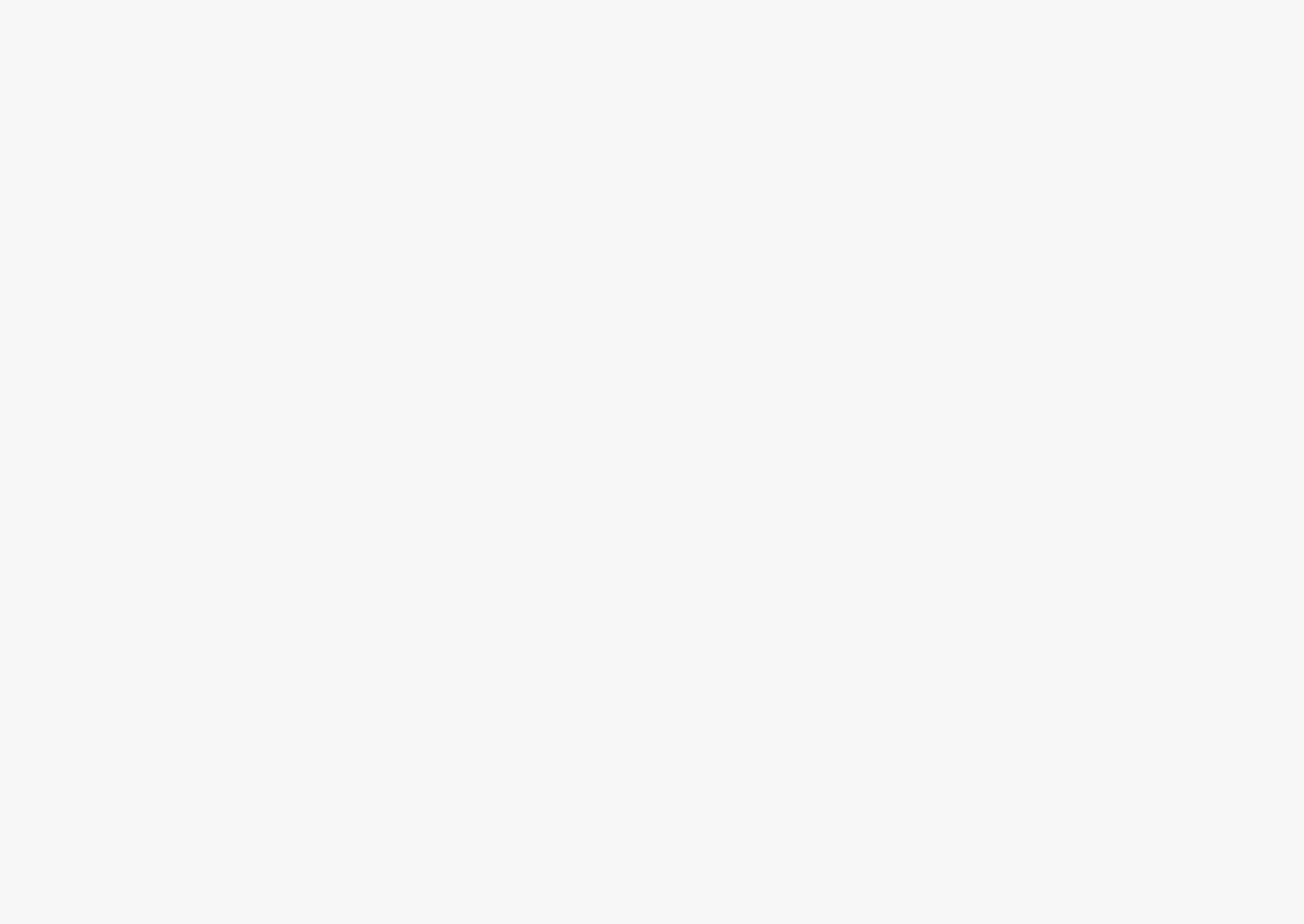
**Paso** **accesorios**



El ejemplo más obvio es cuando el nuevo accesoriosse pasan a un componente. Por ejemplo, tenemos un Formulario de componentes y un componente persona. El componente Formulario tiene una sola<Input /> que permite al usuario cambiar el nombre escribiendo en la entrada. La entrada es unido a la onChangeevento y establece el estado en el formulario. El valor de estado se pasa a continuación a la persona como un componenteapuntalar .



***form.js***



importar Reaccionar de 'reaccionar';

importar Persona de './Persona';

clase predeterminada de exportación Formar se extiende Reaccionar.Componente { constructor(apoyos) {

súper(accesorios);

esta.estado = {Nombre: '' };

esta.handleChange = esta.handleChange.bind (esta);

}

handleChange (evento) {

esta.setState ({nombre: event.currentTarget.value});

}

render () {

regreso (

<Div>

<Input type = "text" onchange = {} this.handleChange /> <nombre de persona = {} this.state.name />

</ Div>

);

}

}



34

Actualización y componentWillReceiveProps ()



En cualquier momento el usuario escribe en el <Input />esto comienza una actualización para el componente persona. El primer método se llama en el componente es



componentWillReceiveProps (nextProps) pasando en el nuevo apuntalar valor. Esto nos permite



comparar el entrante accesorios en contra de nuestra actual accesoriosy tomar decisiones lógicas basadas en el valor. Podemos hacer que nuestros apoyos actuales llamandothis.props y el nuevo valor es el nextProps argumento pasado al método.



**Estado actualizar**

Entonces, ¿por qué es necesario componentWillReceiveProps? Este es el primer gancho que nos permite mirar en la próxima actualización. Aquí podríamos extraer los accesorios nuevos y actualizar nuestro estado interno. Si tenemos un estado que es un cálculo de múltiples apoyos, podemos aplicar con seguridad la lógica aquí y almacenar el resultado utilizandothis.setState () .



Use esto como una oportunidad de reaccionar a una transición hélice antes de render () es llamado por la actualización del estado usando this.setState (). Los viejos puntales se puede acceder a través de this.props. Llamando this.setState () dentro de esta función no se disparará un render adicional.

* <https://facebook.github.io/react/docs/component-specs.html#updating-componentwillreceiveprops>

**Apoyos no pueden cambiar**

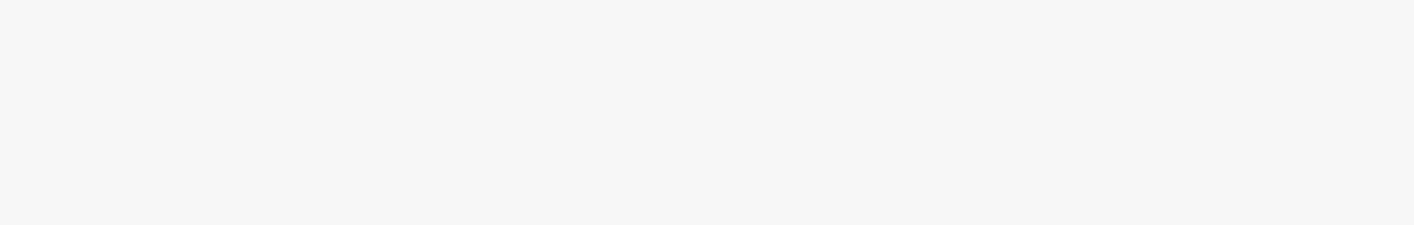
Una palabra de precaución con componentWillReceiveProps (). El hecho de que este método se llama, no significa que el valor de los apoyos ha cambiado.



Para entender por qué, tenemos que pensar en lo que podría haber ocurrido. Los datos podrían haber cambiado entre la primera y hacer que las dos actualizaciones posteriores ... Reaccionar no tiene manera de saber que los datos no han cambiado. Por lo tanto, Reaccionar necesidades para llamarcomponentWillReceiveProps , Ya que el componente necesita ser notificado de la nueva apoyos (incluso si los nuevos apoyos resultan ser los mismos que los viejos apoyos).

-- Ver [(A => B)! => (B => A)](https://facebook.github.io/react/blog/2016/01/08/A-implies-B-does-not-imply-B-implies-A.html)

El tema central con accesorios y componentWillReceiveProps ()Es así como JavaScript proporciona estructuras de datos mutables. Digamos que tenemos un apoyo llamadadatos y datos es una matriz.



// pseudo código

esta.setState ({datos: [1, 2, 3]});

<datos MyComponent ={ this.state.data} />



35

Actualización y componentWillReceiveProps ()



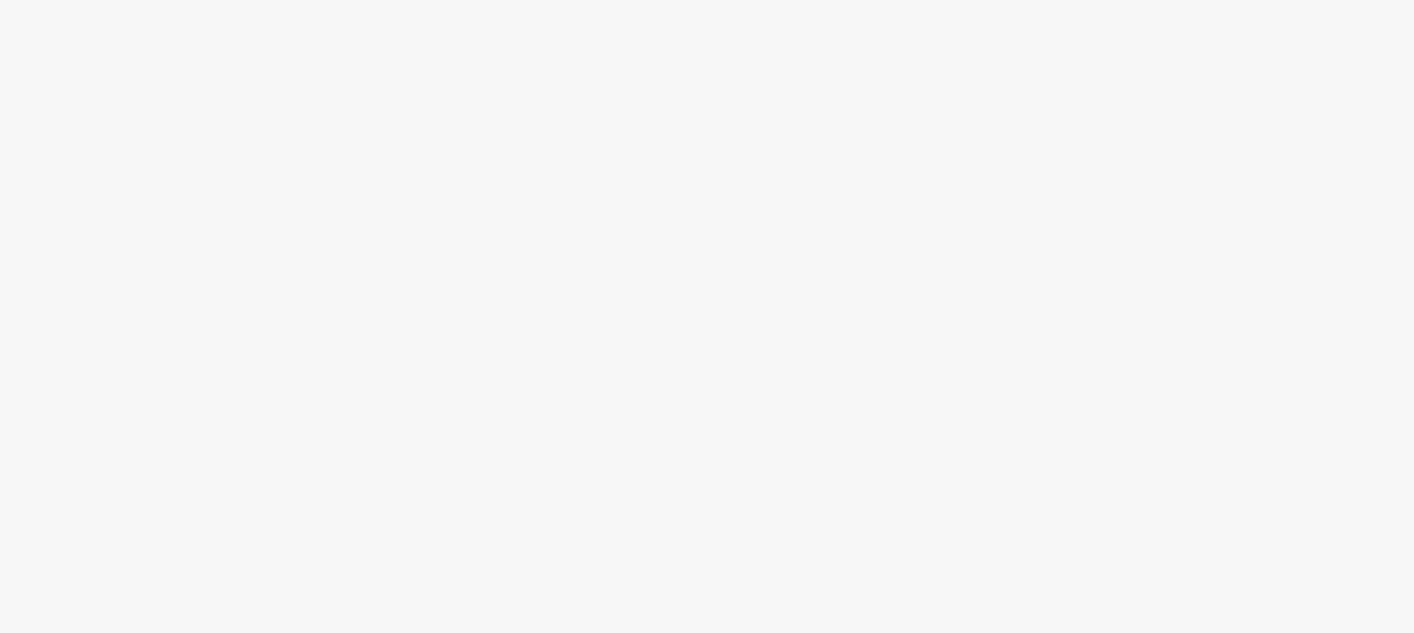
Si en algún lugar de nuestra aplicación, un proceso actualiza la matriz de datos a través de empujar(), Esto cambia el contenido de la matriz de datos. Sin embargo, la matriz en sí es la misma instancia. Debido a que es la misma instancia, Reaccionar no se puede determinar fácilmente si los datos internos ha cambiado. Por lo tanto, para evitar un montón de problemas, o tener que hacer comparaciones profundas, Reaccionar empujará los mismos apoyos hacia abajo.



Con esto se dice, componentWillReceiveProps ()nos permite comprobar y ver si los nuevos apoyos están llegando y podemos tomar decisiones basadas en los datos. Sólo tenemos que asegurarnos de que[Nunca asuma que los apoyos son diferentes en este método. Asegúrese de leer la gran post(A => B)](https://facebook.github.io/react/blog/2016/01/08/A-implies-B-does-not-imply-B-implies-A.html) [! => (B => A) por Jim Sproch.](https://facebook.github.io/react/blog/2016/01/08/A-implies-B-does-not-imply-B-implies-A.html)

**Saltarse este método**

A diferencia de los otros métodos en la fase de montaje, no todos los métodos de actualización de fase son llamados cada vez. Por ejemplo, vamos a omitircomponentWillReceiveProps ()Si la actualización se activa con sólo un cambio de estado. Volviendo a nuestro ejemplo anterior form.js:



* ...

handleChange (evento) {

esta.setState ({nombre: event.currentTarget.value});

}

render () {

regreso (

<Div>

<Input type = "text" onchange = {} this.handleChange /> <nombre de persona = {} this.state.name />

</ Div>

);

}

// ...

Cuando el usuario escribe en el <Input /> que desencadenan una setstate ()método. Esto dará lugar a una fase de actualización en nuestro Formulario de componentes y el componente persona. Para nuestro Formulario de componentes, que no reciben nuevos apoyos, por locomponentWillReceiveProps () se evitará.

***Hasta la próxima:*** [Uso de `shouldComponentUpdate ()](#page37)

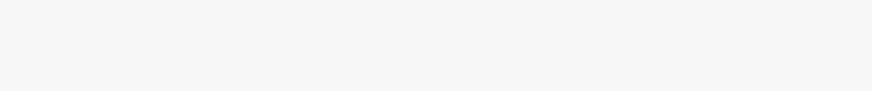


36

Usando shouldComponentUpdate ()



**Utilizando** **shouldComponentUpdate ()**



El siguiente método en el ciclo de actualización es la vida shouldComponentUpdate (). Este método permite que el componente para salir del ciclo de actualización de la vida si no hay ninguna razón para aplicar un nuevo render. Fuera de la caja, elshouldComponentUpdate () es un no-op que devuelve cierto. Esto significa que cada vez que comenzamos una actualización de un componente, vamos a volver a hacer.



Si recuerdan, hace Reaccionar [No profundamente comparar accesorios](https://facebook.github.io/react/blog/2016/01/08/A-implies-B-does-not-imply-B-implies-A.html) por defecto. Cuandoaccesorios o estadose actualiza Reaccionar supone que tenemos que volver a hacer el contenido. Sin embargo, si elaccesorios o estado no han cambiado, debemos realmente ser re-renderizado?



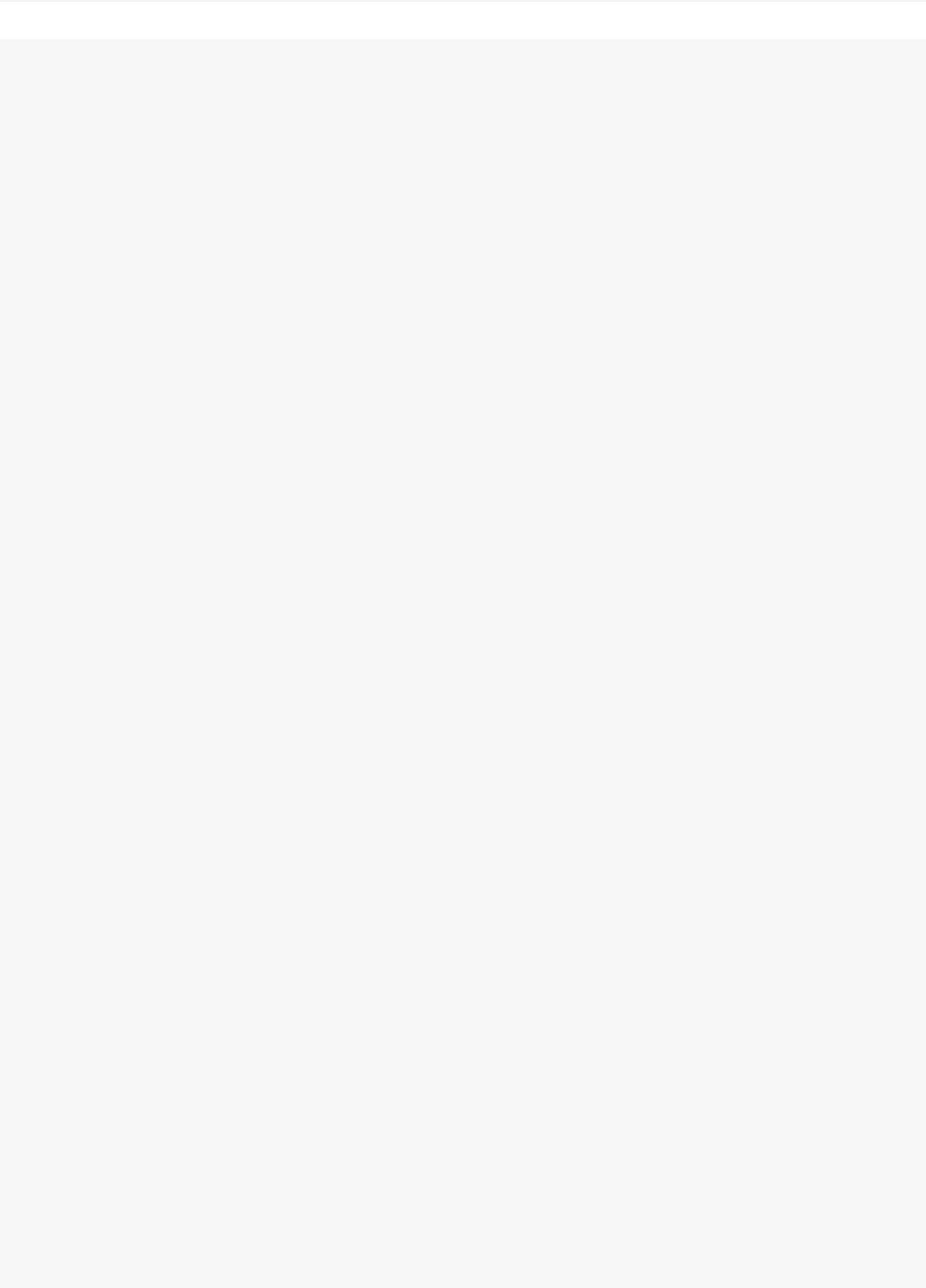
**Prevención haga innecesario**

los shouldComponentUpdate ()método es el primer método de optimización del ciclo de la vida real que podemos aprovechar en React. Podemos mirar a nuestro actual y el nuevoaccesorios y estado y tomar una decisión si se debe seguir adelante. [de reaccionar PureRenderMixin](https://facebook.github.io/react/docs/pure-render-mixin.html) hace exactamente esto. Se comprueba los puntales y el estado actual, lo compara con los próximos puntales y estado y luego vuelvecierto si son diferentes, o falso si son los mismos.



37

Usando shouldComponentUpdate ()



/ \*\*

* Realiza la igualdad por iteración a través de las teclas en un objeto y volver falsa
* cuando cualquier tecla tiene valores que no son estrictamente iguales entre los argumentos.
* Devuelve verdadero cuando los valores de todas las claves son estrictamente iguales.

\* /

función shallowEqual(objA: mezclado, objB: mixto): booleano { Si (ObjA === objB) {

regreso cierto;

}

Si (tipo de objA! == 'objeto' || objA === nulo || tipo de objB! == 'objeto' || objB === nulo) {

regreso falso;

}

var keysA = Objeto.keys (objA);

var keysB = Objeto.keys (objB);

Si (KeysA.length! == keysB.length) {

regreso falso;

}

// Prueba de teclas de A diferentes de B.

var bHasOwnProperty = hasOwnProperty.bind (objB); para (var i = 0; i <keysA.length; i ++) {

Si (! BHasOwnProperty (keysA [i]) || objA [keysA [i]] == objB [keysA [i]]) { regreso falso;

}

}

regreso cierto;

}

función shallowCompare(ejemplo, nextProps, del estado siguiente) { regreso (

! shallowEqual (instance.props, nextProps) ||

! ShallowEqual (instance.state, del estado siguiente)

);

}

var ReactComponentWithPureRenderMixin = { shouldComponentUpdate: función(nextProps, del estado siguiente) {

regreso shallowCompare (esta, NextProps, del estado siguiente); },

};

El código anterior se extrae de la Reaccionar complemento / fuente[1](#page37). El mixin define el



shouldComponentUpdate (nextProps, del estado siguiente) y compara el ejemplo es accesorios en contra el nextProp y el estado en contra de estado siguiente .



38

Usando shouldComponentUpdate ()

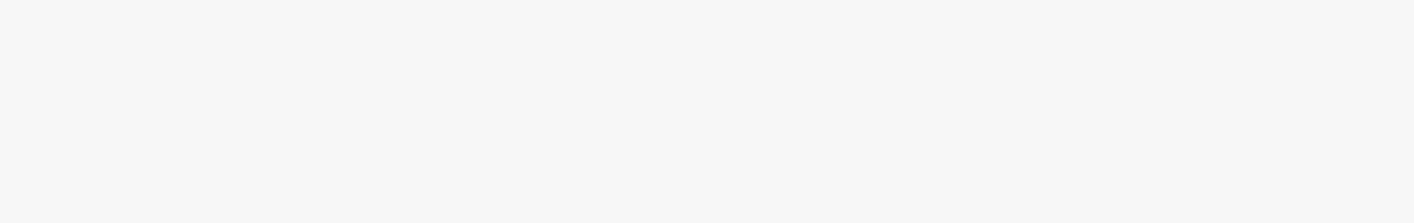


**métodos mutabilidad y puros**

Es importante tener en cuenta que la shallowCompare método utiliza simplemente ===para comprobar cada instancia. Es por esto que el equipo Reaccionar llama al mixin puro, porque no va a comprobar correctamente con los datos mutables.



Vamos a pensar de nuevo a nuestra datos Ejemplo de array puntales donde utilizamos empujar() añadir una nueva pieza de datos en la matriz.



// pseudo código

esta.setState ({datos: [1, 2, 3]});

<datos MyComponent ={ this.state.data} />

los shallowCompare verá la corriente props.data como la misma instancia que el



nextProps.data ( props.data === nextProps.data) Y por lo tanto no hacer una actualización. Ya que



la mutación de la datos Array, nuestro código no se considera que sea puro.



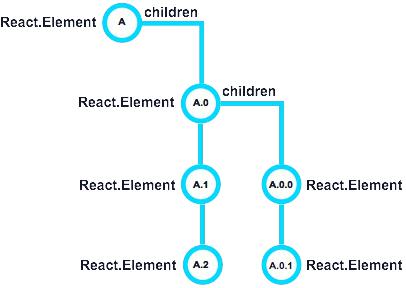
Es por esto que los sistemas como [Redux](http://redux.js.org/) requiere métodos puros para reductores. Si es necesario cambiar los datos anidada que tiene que clonar los objetos y asegurarse de que una nueva instancia siempre se devuelve. Esto permiteshallowCompare () para ver el cambio y actualizar el componente.

Otras maneras de manejar esto es usar un sistema de datos inmutable, tales como [Immutable.js](https://facebook.github.io/immutable-js/). Estas estructuras de datos impiden que los desarrolladores de los datos de forma accidental mutantes. Mediante la aplicación de estructuras de datos inmutables, podemos aprovecharshouldComponentUpdate () y haga que sea verificar que nuestra accesorios y estado han cambiado[2](#page37).



**Detener hace en la fuente**

Si recuerdan nuestra estructura anidada de componentes:



39

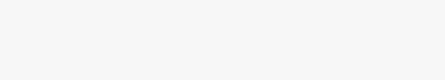
Usando shouldComponentUpdate ()



Por defecto, si una actualización se dispara en A, entonces todos los otros niños también pasan por sus actualizaciones. Esto puede provocar fácilmente un problema de rendimiento, porque ahora tenemos muchos componentes que van a través de cada paso del proceso.

Mediante la adición de controles lógicos en shouldComponentUpdate ()en A podemos evitar que todos sus niños de re-representación. Esto puede mejorar significativamente el rendimiento general. Pero hay que tener en cuenta, si se impide un pase a los apoyos a los niños que puede evitar que se requieren hace que se produzcan.

**Saltar adelante con** **forceUpdate ()**



Me gusta componentWillReceiveProps () , Podemos omitir shouldComponentUpdate () llamando

forceUpdate () en el componente. Esto establece una bandera en el componente cuando se añade a la cola sucia. Cuando marcado,shouldComponentUpdate ()se ignora. Debido a que estamos forzando una actualización estamos afirmando algo ha cambiado y el componente debe volver a hacer.

Ya que forceUpdate ()es un método de fuerza bruta, que siempre debe utilizarse con precaución y consideración cuidadosa. Usted puede conseguir fácilmente en un bucle sin fin hacer que si se mantiene la activaciónforceUpdate una y otra vez. Solución de problemas render bucles infinitos puede ser muy difícil. Asi que, cuando trate de alcanzar forceUpdate mantener todo esto en mente.

***Lo siguiente:*** [AprovechandocomponentWillUpdate ()](#page41)



* capturado de [reaccionar 15.0.1](https://github.com/facebook/react/blob/15-stable/src/addons/shallowCompare.js)
* Si utiliza Immutable.js, hay una [biblioteca ImmutableRenderMixin](https://github.com/jurassix/react-immutable-render-mixin) que proporciona tanto superficial del objeto comparar y comparación de datos inmutable.



40

El aprovechamiento de componentWillUpdate ()



**Aprovechando** **componentWillUpdate ()**



Una vez que hemos determinado que sí tenemos que volver a hacer en nuestra fase de actualización, el componentWillUpdate () sera llamado. El método se hace pasar en dos argumentos: nextProps



y estado siguiente . El método componentWillUpdate () es [Similar a](#page17) [componentWillMount ()](#page17) ,

y muchas de las mismas consideraciones y tareas son las mismas. La diferencia es quecomponentWillUpdate () se llama cada vez que un re-render se requiere, por ejemplo, cuando

this.setState () se llama. diferente a componentWillMount () en la que accedemos a la siguiente accesorios y estado .



Al igual que componentWillMount () , Este método se llama antes hacer(). Debido a que no han hecho aún, el acceso de nuestro componente de la interfaz de usuario nativa (DOM, etc.) reflejará la antigua interfaz de usuario prestado. diferente acomponentWillMount () , Podemos acceder refspero en general esto no es recomendable porque los árbitros pronto estarán fuera de fecha. Hay casos de uso para acceder a la interfaz de usuario nativa aquí, como iniciar animaciones.



los componentWillUpdate ()Es una oportunidad para nosotros para manejar los cambios de configuración y prepararse para la próxima render. Si queremos acceder a los viejos puntales o estado, que podemos llamarthis.props o este estado . entonces podemos compararlos con los nuevos valores y hacer cambios / cálculos

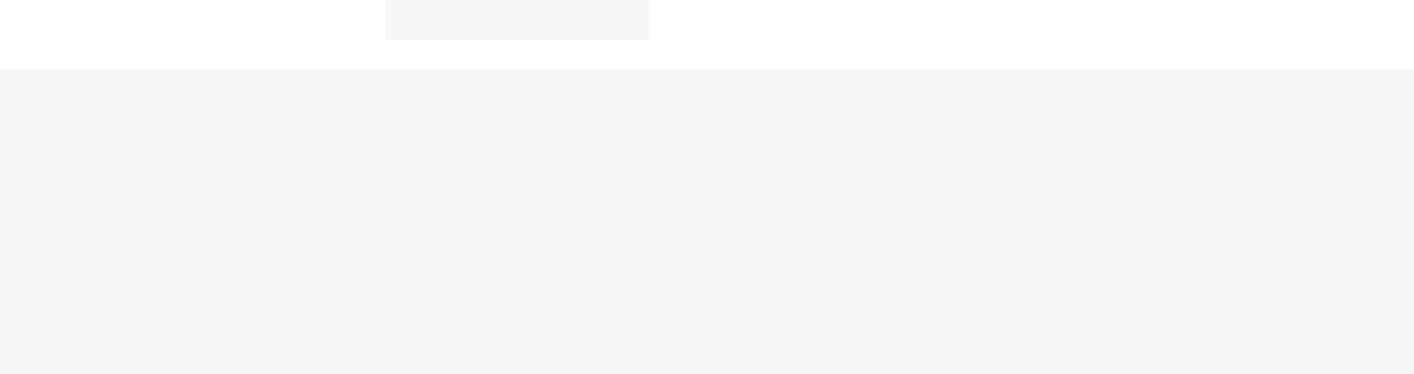


según sea necesario.

diferente a componentWillMount () , No hay que llamar this.setState ()aquí. La razón por la que no lo llamamosthis.setState () es que el método desencadena otro componentWillUpdate (). Si nos desencadenar un cambio de estado en elcomponentWillUpdate () vamos a terminar en un bucle infinito [1](#page41).



Algunos de los usos más comunes para componentWillUpdate () es establecer una variable en función de los cambios de estado (no usar this.setState () ), el envío de eventos o de partida animaciones [2](#page41).



* el envío de una acción basada en el cambio de estado componentWillUpdate (nextProps, del estado siguiente) {

Si (NextState.open == cierto && esta.state.open == falso) { esta.props.onWillOpen ();

}

}

***Hasta la próxima:*** [Re-representación y actualizaciones niños](#page43)



* En la versión anterior de esta sección por error dice que se puede llamar de forma segura

setstate () en este método. Nuestro supuesto en ese momento era que una bandera sucia estaba rastreando el estado actual de la pasada de render, pero este no es el caso. Técnicamente es posible llamar



41

El aprovechamiento de componentWillUpdate ()



setstate () detrás de una condicional (tal como cuando un cambios prop / estado) pero es **no** recomendada y debe ser considerado un no ir. Agradecimientos especiales a[Robin Venneman](https://github.com/robinvenneman) para la captura de este error y llamar a nuestra atención!



* Un ejemplo de desencadenar transiciones CSS en componentWillUpdate () y otras discusiones en torno a los usos de este método es más en este [respuesta StackOverflow](http://stackoverflow.com/a/33075514/815544).



42

Re-representación y actualizaciones niños

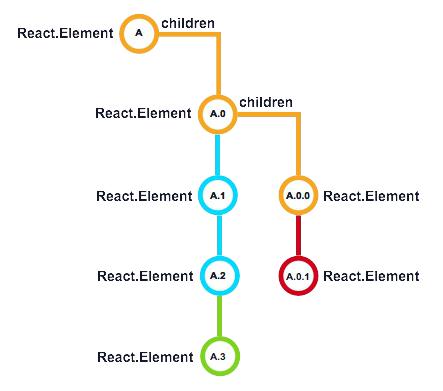


**Re-representación y actualizaciones niños**

Una vez más volvemos a hacer(). Ahora que nuestraaccesorios y estado son todos actualizado[1](#page43) nosotros [puede aplicarlos a nuestro contenido y niños. Al igual que el primer render,todos las mismas reglas y](#page20) [las condiciones se aplican.](#page20)



A diferencia de nuestro primer render, Reaccionar realiza una gestión diferente cuando se trata de los elementos generados. La principal diferencia es alrededor de los elementos de fase de inicialización e hijos de los componentes.



Reaccionar compara la actual estructura de árbol de elementos volvió de la hacer()método. Reaccionar utiliza las claves generadas (o teclas asignadas) para que coincida con cada elemento a una instancia de componente. Reaccionar determina si tenemos nuevos casos (A.3), la eliminación de los casos (A.0.1) o actualizando las instancias existentes (A, A.0, A.0.0).

Si las claves son los mismos, entonces pasará la Reaccionar accesoriosa la instancia existente, dando inicio a su ciclo de actualización de la vida. Si hemos añadido nuevos componentes o claves cambiado, Reaccionar va a crear nuevas instancias de los datos del elemento. Estos nuevos componentes entran entonces en la fase de nacimiento / montaje.



***Hasta la próxima:*** [Post-Procesamiento concomponentDidUpdate ()](#page45)



43

Re-representación y actualizaciones niños



* Como se mencionó anteriormente, la naturaleza asincronía del estado está ahora plenamente aplicado y se puede acceder de forma segura.

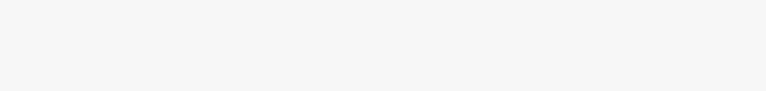


44

Post-Procesamiento con componentDidUpdate ()



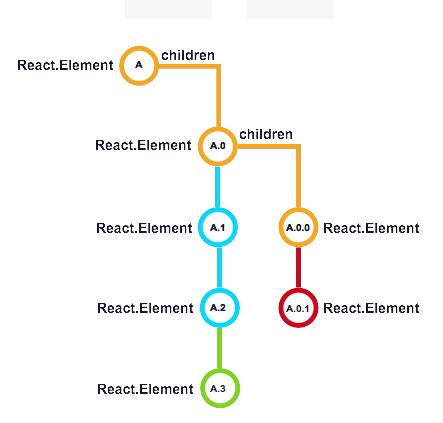
**Post-Procesamiento con** **componentDidUpdate ()**



Continuando con la tendencia de los métodos correspondientes, el componentDidUpdate () es la versión de actualización [componentDidMount ()](#page25) . Una vez más, podemos acceder a la pila de interfaz de usuario nativa, interactuar con nuestrarefs y, si es necesario iniciar otra re-render / actualización [1](#page45).



Cuando componentDidUpdate () se llama, se pasan dos argumentos: prevProps y prevState . Esta es la inversa de componentWillUpdate () . Los valores pasados ​​son lo que el los valores eran, y this.props y este estado son los valores actuales.



Al igual que componentDidMount () , el componentDidUpdate ()se llama después de que todos los niños se actualizan. Sólo para refrescar la memoria, A.0.0 tendrácomponentDidUpdate () llamado primero, y luego A.0, y finalmente A.

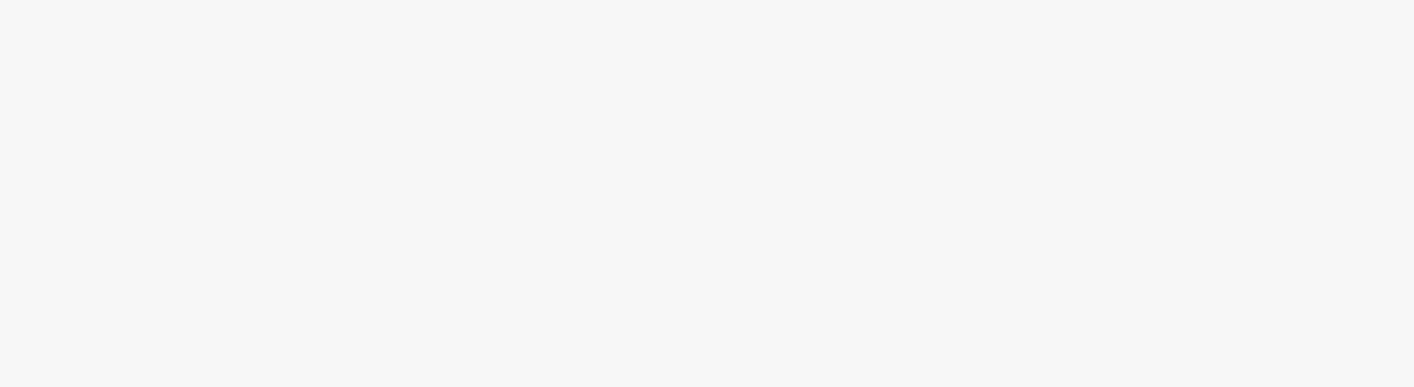
**Tareas comunes**

Los usos más comunes de componentDidUpdate ()es la gestión de los elementos de interfaz de usuario 3er partido y la interacción con la interfaz de usuario nativa. Al utilizar las bibliotecas tercio del partido, como nuestro ejemplo, Carta, necesitamos actualizar la biblioteca de interfaz de usuario con datos nuevos.



45

Post-Procesamiento con componentDidUpdate ()



componentDidUpdate (prevProps, prevState) {

* Sólo actualizar la carta si los datos han cambiado Si (PrevProps.data! == esta.props.data) {

esta.chart = c3.load ({ datos: esta.props.data

});

}

}

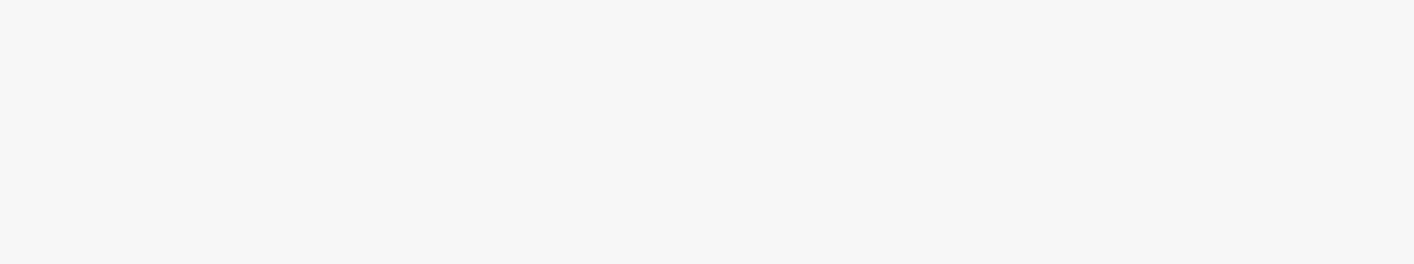
Aquí tenemos acceso a nuestra instancia Gráfico y actualizarlo cuando los datos han cambiado[2](#page45).

**Otra Representar?**

También podemos consultar la interfaz de usuario nativa y obtener el tamaño, estilo CSS, etc. Esto nos puede requerir actualizar nuestro estado interno o accesoriospara nuestros hijos. Si este es el caso podemos llamarthis.setState () o forceUpdate () aquí, pero esto abre una gran cantidad de problemas potenciales porque obliga a un nuevo pase de render.



Una de las peores cosas que hacer es hacer un sin marcar setstate () :



componentDidUpdate (prevProps, prevState) {

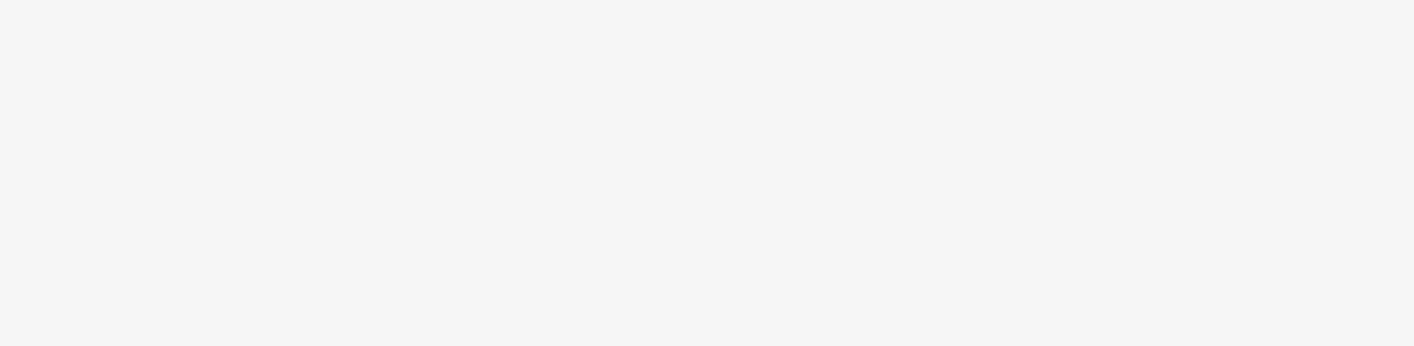
// malo: no hace esto !!!

dejar height = ReactDOM.findDOMNode (esta) .offsetHeight; esta.setState ({internalHeight: altura});

}

Por defecto, nuestra shouldComponentUpdate ()Devuelve verdadero, así que si utilizamos el código anterior caeríamos en un bucle infinito render. Nos gustaría hacer, entonces llamamos hizo que establece la actualización de estado, lo que provocó otra render.

Si tiene que hacer algo como esto, entonces se puede aplicar un cheque en componentDidUpdate () y / o añadir otras comprobaciones para determinar cuando un re-tamaño realmente ocurrió.



componentDidUpdate (prevProps, prevState) {

// Una solución posible ...

dejar height = ReactDOM.findDOMNode (esta) .offsetHeight; Si (esta.state.height! == altura) {

esta.setState ({internalHeight: altura});

}

}

En general, esto no es un requisito común y re-representación tiene impacto en el rendimiento de su módulo y aplicaciones. Tenga esto en cuenta si usted encuentra que tiene que añadir un segundo pase en rendircomponentDidUpdate () .



46

Post-Procesamiento con componentDidUpdate ()



***Hasta la próxima:*** [De muerte / Desmontando en profundidad](#page48)



* Este es un comportamiento arriesgado y puede entrar fácilmente en un bucle infinito. ¡Proceda con precaución!

[2](#page45) En este ejemplo se supone que los datos es puro y no mutado.



47

Muerte / desmontar en profundidad

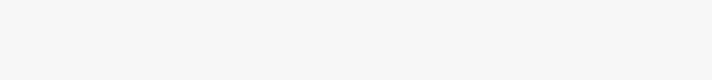


**Muerte / desmontar en profundidad**

Después de nuestro componente ha pasado tiempo en la fase de actualización, que finalmente entrar en la fase de muerte. Durante esta fase de nuestro componente se desmonta de la pila de interfaz de usuario nativa y está marcado para recolección de basura.

Nos entrar en esta fase cuando nuestros cambios de interfaz de usuario y el árbol de elementos ya no tiene una clave que coincida con nuestro componente. Este podría ser el cambio de diseño o programación cambiar teclas (forzando una nueva instancia de componente que se creará). Una vez que esto ocurre, Reaccionar miradas en la instancia que se está eliminado y sus hijos.

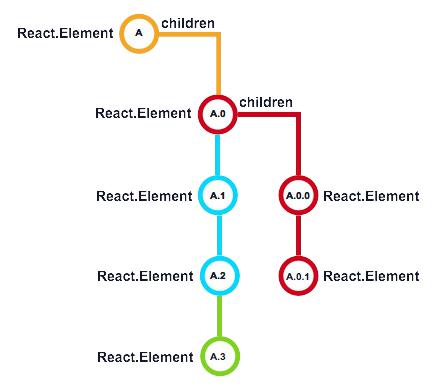
**Utilizando** **componentWillUnmount ()**



Al igual que el resto de nuestras fases del ciclo de vida, la fase de muerte / Desmontar tiene un gancho método para nosotros. Este método nos permite hacer algo de limpieza antes de que se retiran de la pila de interfaz de usuario. Normalmente queremos revertir cualquier configuración que hicimos en cualquieracomponentWillMount () o

componentDidMount () .

Por ejemplo, nos gustaría anular el registro de eventos / System / Library mundial, destruir elementos de la interfaz de usuario de la biblioteca de 3 ª Parte, etc. Si no nos tomamos el tiempo para eliminar los eventos que podemos crear pérdidas de memoria en nuestro sistema o salir de malas referencias por ahí .



48

Muerte / desmontar en profundidad



Reaccionan comienza con el elemento que se eliminan, por ejemplo A.0 y llamadas componentWillUnmount () en eso. Entonces Reaccionar va al primer hijo (A.0.0) y hace lo mismo, trabajando su camino hacia abajo hasta el último niño. Una vez que se han realizado todas las llamadas, Reaccionar eliminará los componentes de la interfaz de usuario y prepararlos para la recolección de basura.

***Hasta la próxima:*** [La Crónica del Ciclo de Vida](#page50)



49

La Crónica del Ciclo de Vida



**La Crónica del Ciclo de Vida**

Ahora hemos trabajado a través de las tres fases del ciclo de vida React: [Nacimiento / Montaje](#page10), [Crecimiento / Actualización](#page31) y finalmente [Muerte / Desmontar](#page48). Al tener estas fases y las correspondientes Reaccionar métodos nos proporciona un camino claro para el desarrollo de componentes. Estas fases también nos permitirá comenzar a optimizar nuestros componentes y toda nuestra aplicación.

Para revisar, los métodos y el orden son llamados:

**Nacimiento / Montaje**

1. Inicializar / Construcción
2. getDefaultProps () *(React.createClass)* o MyComponent.defaultProps *(Clase ES6)*
3. getInitialState () *(React.createClass)* o this.state = ... *(Constructor ES6)*
4. componentWillMount ()
5. hacer()
6. Los niños de inicialización y ciclo de vida de patada de salida
7. componentDidMount ()



**Crecimiento / Actualización**

1. componentWillReceiveProps ()
2. shouldComponentUpdate ()
3. componentWillUpdate ()
4. hacer()
5. métodos de ciclo de vida de los niños
6. componentDidUpdate ()



**Muerte / Un-Montaje**

1. componentWillUnmount ()
2. métodos de ciclo de vida de los niños
3. Instancia destruyó la recolección de basura



**Diagrama de flujo del ciclo de vida y** **setstate ()** **la seguridad**

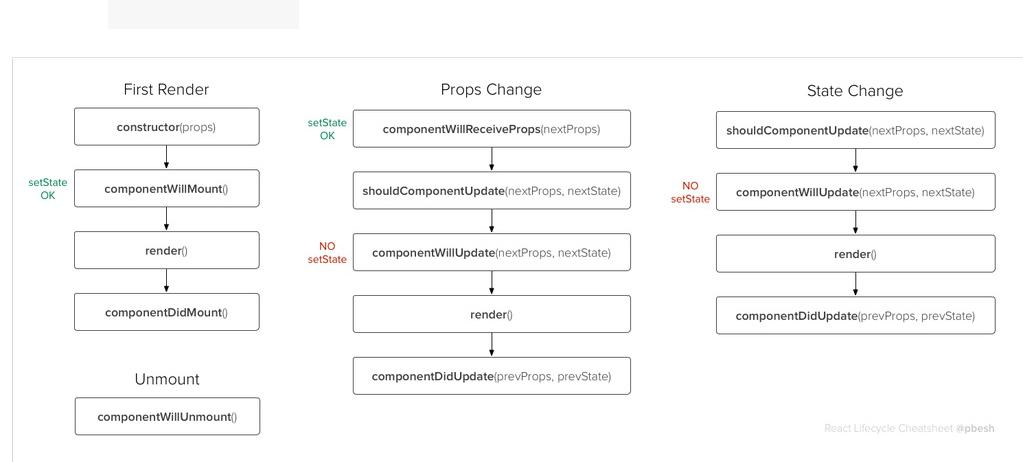


50

La Crónica del Ciclo de Vida



Además, este diagrama de flujo por [Peter Beshai](https://twitter.com/pbesh) descompone los diferentes métodos y pide también cuando this.setState () es seguro y no seguro llamar a:



51

Componente Evolución y Composición



**Componente Evolución y Composición**

la reutilización de los componentes y del composability son algunos de los principios básicos de desarrollo React. A medida que la escala de nuestras aplicaciones, el tiempo de desarrollo puede reducirse drásticamente como resultado de este proceso. Sin embargo, la creación de componentes reutilizables requiere de una planificación y comprensión para soportar múltiples casos de uso.

La comprensión de la intención del componente es el primer paso para su reutilización. A veces, sabemos que un componente se puede utilizar de muchas maneras diferentes desde el principio. En esas situaciones, podemos planear para los distintos escenarios de inmediato. En otras situaciones, las intenciones de componente cambiarán durante la vida útil de la aplicación. La comprensión de cómo evolucione un componente es tan importante como la comprensión de cómo crear reutilización.

**El proceso de Arquitectura de aplicaciones**

Echemos un momento rápido y discutir el proceso de arquitectura de la aplicación. A menudo oímos acerca de los sistemas con arquitectura excesiva. Esto ocurre a menudo cuando tratamos de planificar todos los escenarios posibles que le puede ocurrir a través de la vida de una aplicación. Para tratar de apoyar a todos los posibles usos es un mandado tontos. Cuando tratamos de construir estos sistemas añadimos una complejidad innecesaria y, a menudo más difícil hacer que el desarrollo, en lugar de más fácil.

Al mismo tiempo, no queremos construir un sistema que no ofrece ninguna flexibilidad en absoluto. Puede ser más rápido que acaba de construirlo sin pensar en el futuro, pero la adición de nuevas características puede ser tan lento más adelante. Tratando de encontrar el equilibrio adecuado es la parte más dura de la arquitectura de aplicaciones. Queremos crear una aplicación flexible que permite el crecimiento, pero no queremos perder el tiempo en todas las posibilidades.

El otro reto con arquitectura de la aplicación está tratando de entender nuestras necesidades. Con el desarrollo, a menudo tenemos que construir algo para comprender verdaderamente la misma. Esto significa que nuestra arquitectura de la aplicación es un proceso vivo. Cambia con el tiempo debido a tener una mejor comprensión de lo que se requiere. Componentes refactorización es fundamental para el éxito de un proyecto y hace más fácil la adición de nuevas características.

Debido a este proceso, sentimos que es importante caminar a través de la evolución de un componente. Vamos a empezar con un enfoque ingenuo para la construcción de una lista de componentes y luego caminar a través de diferentes refactorizaciones para apoyar la reutilización. Más que probable, sabríamos desde el principio que una lista debe ser reutilizable. Pero, caminando a través del proceso de evolución puede ayudar a profundizar nuestra comprensión de cómo habilitar la reutilización.

***Hasta la próxima***: [La evolución de un componente List](#page54)



52

Componente Evolución y Composición



53

La evolución de un componente List



**La evolución de un componente List**

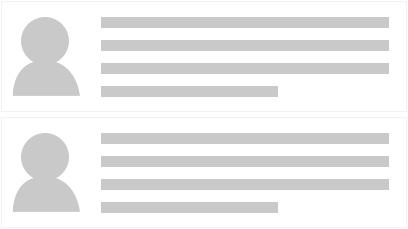
Las listas están por todas partes en las aplicaciones de hoy en día. La lista es crucial para interfaces de usuario de los medios sociales, como Facebook, Twitter, Reddit, Instagram, etc. La tendencia actual de aplicación de demostración Todos son todos acerca de cómo mostrar una lista de elementos. El humilde HTML desplegable muestra una lista de opciones seleccionables. Es tan común, la mayoría de nosotros damos por sentado listas.

Cuando comenzamos a construir nuestra aplicación, ¿cómo debemos abordar la creación de componentes reutilizables? Vamos a caminar a través de una posible progresión de una función de lista.

**El primer pase**

Por lo general, el primer enfoque es construir un componente Reaccionar que hace que la interfaz de usuario a las necesidades de diseño y de datos específicos. Para nuestro ejemplo, estamos construyendo una lista de perfiles de clientes. La primera ronda de diseño requiere que el perfil de tener un avatar / imagen y texto descriptivo.

**IU de carcasa # 1**

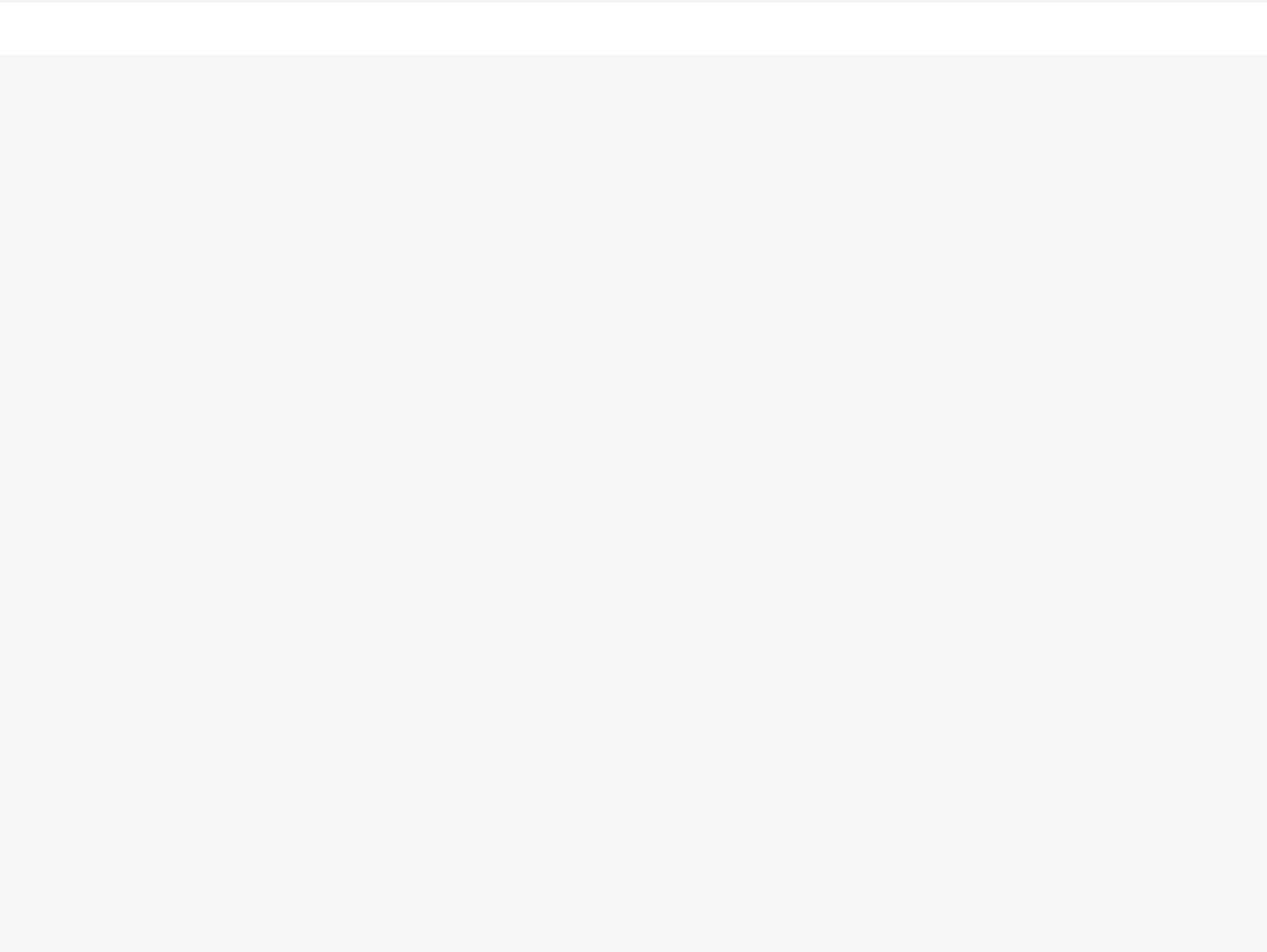


El primer paso sería la creación de un componente que tiene una matriz de objetos, que tiene una trayectoria de la imagen y el texto de descripción. Nuestro bucle voluntad de componentes sobre esta matriz y renderizar cada elemento, utilizando<Li> artículos.



54

La evolución de un componente List



importar Reaccionar de 'reaccionar';

clase Lista se extiende Reaccionar.Componente {

renderProfiles () {

devolver este.props.profile.map ((perfil) => {

regreso (

<Li>

<Img href ={ profile.imagePath} align ="izquierda" width ="30" height ="30" /> <Div className ="Descripción del perfil">

{ Descripción del perfil } </ Div>

</ Li>

);

});

}

render () {

regreso (<Ul className ="Perfil-list">{()} this.renderProfiles</ Ul>);

}

}

List.defaultProps = {perfil: []};

Lista predeterminada de exportación;

entonces podríamos aplicar un estilo a la <Ul> , <Li> y <Div>elementos para satisfacer nuestras necesidades de diseño. Este componente es una forma sencilla de hacer nuestra contenido. Que satisface nuestras necesidades de diseño, pero no es reutilizable.



**cambio de los requisitos**

Al igual que con cualquier proyecto, las necesidades cambian. Para nuestro ejemplo, los usuarios quieren ahora al listado más detalles acerca de cada cliente. El equipo de diseño viene con un nuevo diseño y ahora tenemos que soportar campos opcionales.

**IU de carcasa # 2**



55

La evolución de un componente List



Con este nuevo diseño, ahora tenemos que hacer nuestro primer bit de refactorización de componentes. Para apoyar a los nuevos campos de detalles opcionales, tenemos que añadir lógica a nuestro perfil de renderizado. Una buena práctica de desarrollo es mantener a nuestros Reaccionar componentes como compartimentada como sea posible. Esto permite que múltiples beneficios.

En primer lugar, ayuda a reducir la carga cognitiva. Tener más pequeños componentes individuales enfocadas, significa que son más fáciles de leer y entender la intención. Una experiencia común que todos hemos tenido como desarrolladores está volviendo a nuestro propio código de seis o más meses después. Debido a lo escribimos, hay que entenderlo, pero a menudo se necesita un poco de tiempo para ponernos de nuevo en modo de pensar de lo que el código está resolviendo. Si el componente tiene cientos de líneas de la lógica, que tomará mucho más tiempo para asimilar lo que se pretende. Aún más difícil (y consume mucho tiempo) está haciendo esto con el trabajo de otro desarrollador.

Una de las características hermosas de Reaccionar es que podemos (y debemos) romper nuestros Componentes

en pequeños trozos pequeños. Debido a que es tan fácil en reaccionar, esto ayuda a hacer que nuestro código

Más fácil de entender. Al mismo tiempo, esto lleva a la segunda ventaja: la reutilización más rápida.

Si rompemos un componente a una sola tarea, tales como la prestación de un único perfil, ahora tenemos la posibilidad de volver a utilizarlo. Es posible que en la aplicación a otra parte tenemos que mostrar un perfil. Con nuestra implementación actual, esto no es fácil de hacer. Esto se debe a la prestación de los detalles del perfil es manejado internamente por el componente List. Vamos a romper los detalles del perfil hacia un nuevo componente y refactorizar nuestra lista un poco.

**Creación de un componente Perfil**

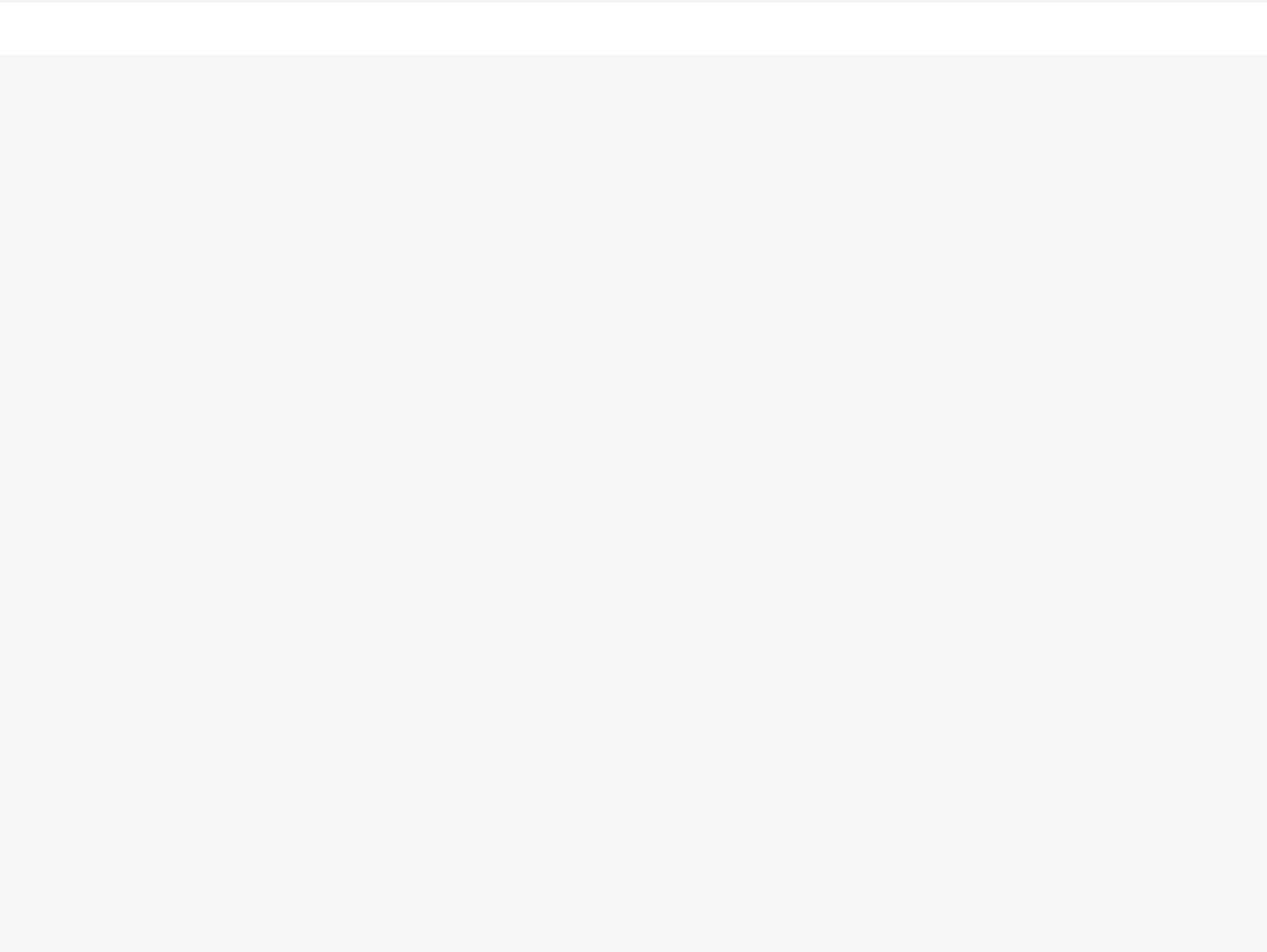
El primer paso es mover el código de render de la lista en su propio componente.

**Profile.js**



56

La evolución de un componente List



importar Reaccionar de 'reaccionar';

clase predeterminada de exportación Perfil se extiende Reaccionar.Componente { renderDetails (de llave, etiqueta) {

Si (esta.props [tecla]) {

regreso (<Div className ="detalle">{Label} {this.props [clave]}</ Div>);

}

}

render () {

regreso (

<Li>

<Img href ={ this.props.imagePath} align ="izquierda" width ="30" height ="30" /> <Div className ="Descripción del perfil">

{} This.props.description </ Div>

{this.renderDetails ( 'email', 'Email:')}

{this.renderDetails ( 'Twitter', 'Twitter:')} {this.renderDetails ( 'teléfono', 'Teléfono:')}

</ Li>

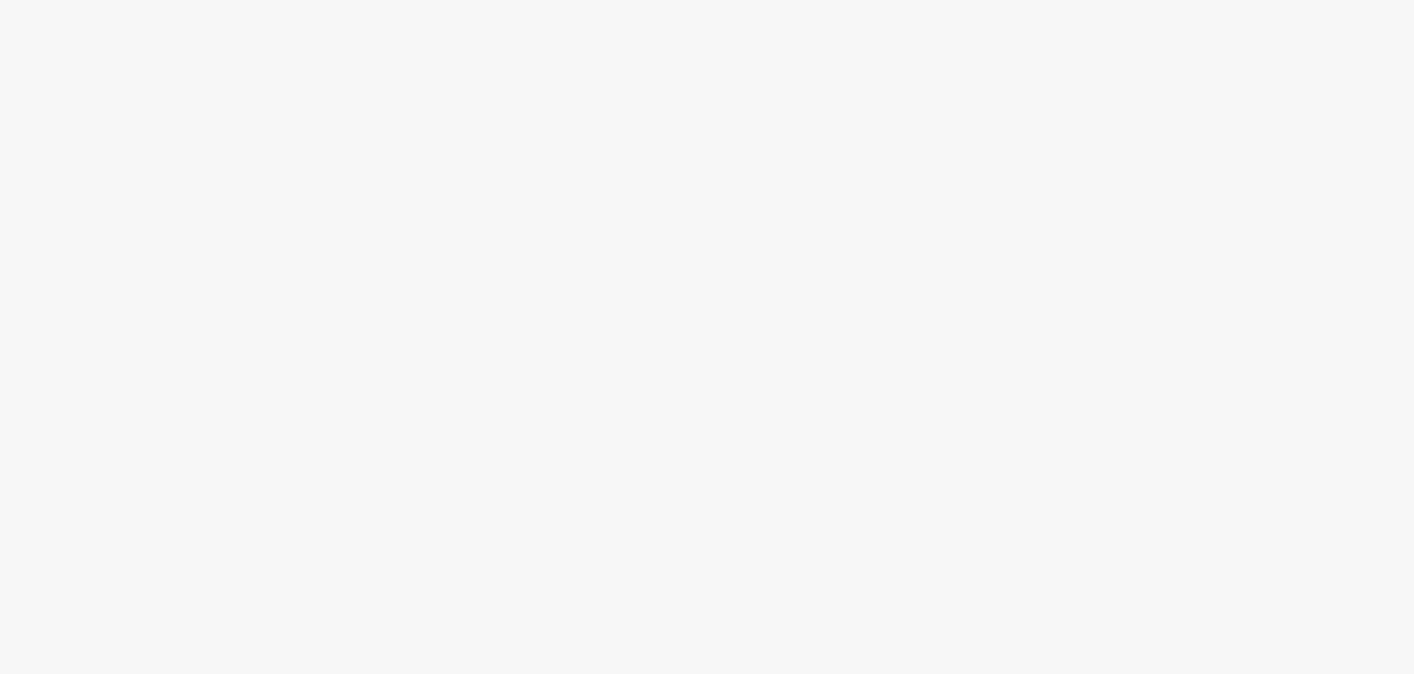
);

}

}

Aquí hemos roto los detalles opcionales representación en un nuevo componente llamado Perfil . El trabajo de perfil es hacer que el diseño de la base y luego hacer nuestra opcional detalles, dependiendo de si están o no definidos[1](#page54). entonces podemos actualizar nuestro código de lista:

**List.js**



importar Reaccionar de 'reaccionar';

importar Perfil de './Perfil';

clase Lista se extiende Reaccionar.Componente {

render () {

regreso (

<Ul>

{This.props.profile.map ((perfil) => <Perfil {...} perfil /> )} </ Ul>

);

}

}

List.defaultProps = {perfil: []};

Lista predeterminada de exportación;



57

La evolución de un componente List



Ahora nuestra lista de los mapas de los datos del perfil y lo envía a la Perfil Componente para la representación. [Al aislar la prestación del perfil de un solo componente que tenemos una clara separación de](https://en.wikipedia.org/wiki/Separation_of_concerns) [preocupaciones (SoC). No sólo tenemos la ventaja de SoC, también hacemos cada Componente mucho](https://en.wikipedia.org/wiki/Separation_of_concerns) Más fácil de entender. Cuando tenemos que volver a este código de seis meses más tarde, será mucho más rápido a quedar atrapada una copia de seguridad.

***Hasta la próxima***: [Rendering contenido diferente](#page59)



* Siguiendo este patrón podríamos ir aún más lejos si así se desea. Podríamos dividir cada perfil detalle en su propio componente. Sin embargo, que tal vez ir demasiado lejos en el agujero del conejo granularidad. Una vez más, el exceso de la arquitectura es una pendiente resbaladiza y tener que hacer un juicio es parte del proceso.



58

Representar el contenido diferente



**Representar el contenido diferente**

Moviendo nuestra representación de la interfaz de usuario de cada perfil a un componente, nos hemos separado de diseño y visualización de contenido. La lista es responsable de diseño y gestión de datos. El perfil es el encargado de mostrar la interfaz de usuario para cada elemento individual. Debido a este primer paso, podemos avanzar un paso más y hacer nuestra lista aún más flexible.

**Lista de ampliación de funciones**

Continuando con nuestro ejemplo al cliente, imaginemos que nuestra lista de perfiles ha comenzado a evolucionar aún más. Hemos añadido soporte de paginación, la gestión de la selección, clasificación, filtrado, etc. Ahora, nuestros usuarios solicitan la posibilidad de gestionar un tipo diferente de contenido. Ahora quieren para gestionar los mensajes.

Estos mensajes tienen algunos elementos de interfaz de usuario similares a nuestro perfil: imágenes, descripciones y detalles. Sin embargo, la disposición y el contenido varían drásticamente. Todavía tenemos toda la parte de la funcionalidad de la Lista; paginación, el filtrado, etc. La pregunta es, ¿cómo se realiza el tratamiento?

**Representación de artículos**

Un simple, pero no es ideal, el enfoque sería añadir un interruptor en nuestra lista de mapamétodo. El interruptor comprueba el tipo de datos y luego opta por utilizar el Componente perfil o el componente Post. Sin embargo, este enfoque añade una muy mala[olor código](https://en.wikipedia.org/wiki/Code_smell). Al igual que en nuestro primer borrador de la lista, que satisface nuestras necesidades inmediatas, pero lo que sucede cuando necesitamos una lista de mensajes? O List Viewer? Pronto nuestra lista tiene una gran cantidad de interruptores.



Una mejor manera de resolver esto es a través de la configuración. Podemos exponer un puntal en el componente List que se encarga de la prestación de cada elemento. Hay dos maneras de hacer esto: mediante introducción de una función o mediante introducción de una clase de componente.

**Procesador de Elemento Función**

El primer enfoque examinaremos está pasando en una función que se encarga de renderizar cada elemento individual en la lista. El primer paso es actualizar nuestra lista de componentes para requerir unaitemRenderer puntal que es una función y cambiar nuestros perfiles apuntalar a los elementos.

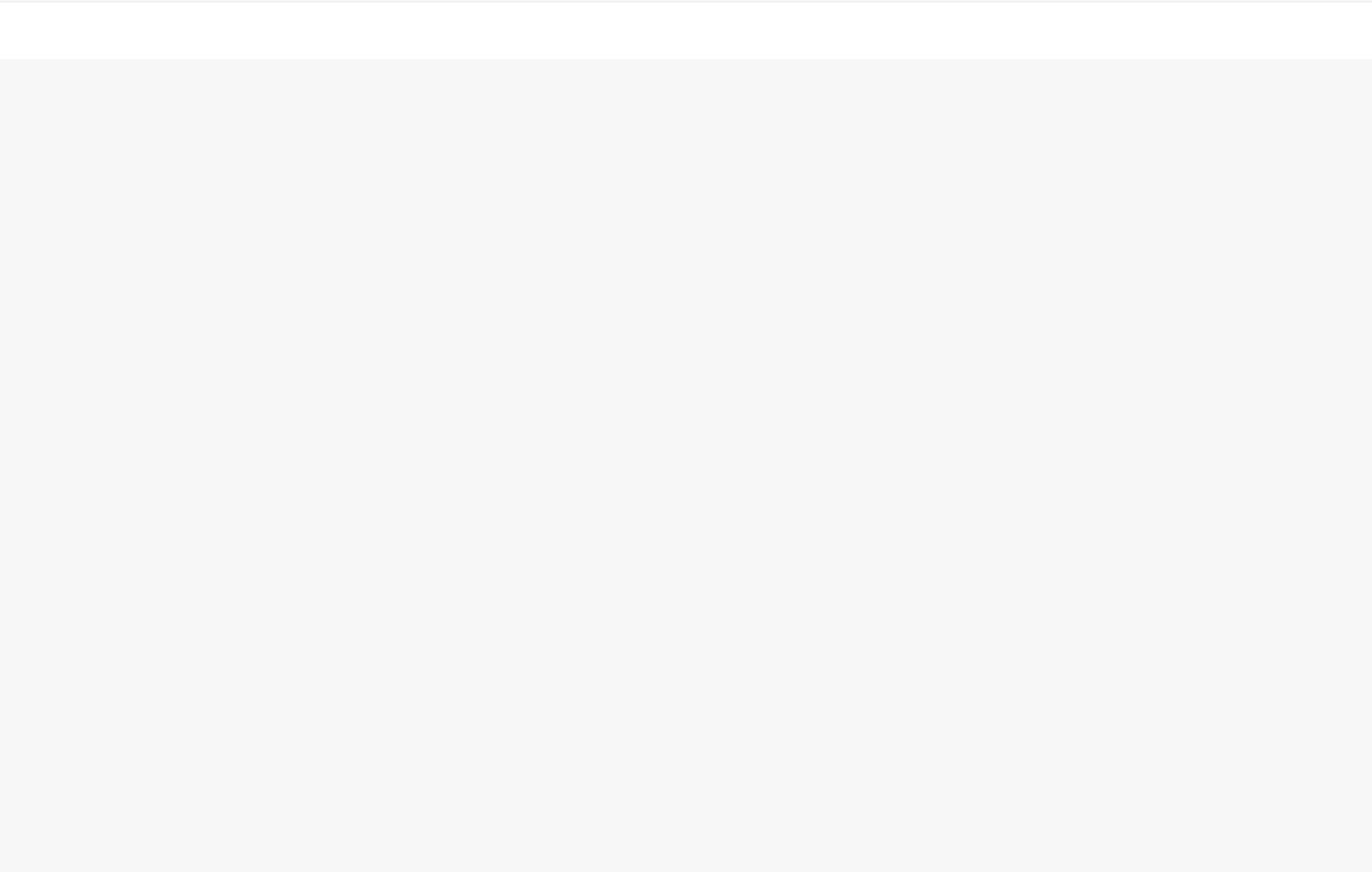


**List.js**



59

Representar el contenido diferente



importar Reaccionar de 'reaccionar';

clase Lista se extiende Reaccionar.Componente {

render () {

regreso (

<Ul>

{This.props.items.map ((artículo, índice) => this.props.itemRenderer (artículo, índice)

)}

</ Ul>

);

}

}

List.propTypes = {

artículos: React.PropTypes.array,

itemRenderer: React.PropTypes.func.isRequired };

List.defaultProps = {objetos: []};

por defecto de exportación Lista;

Hemos añadido una propTypes configuración para requerir la itemRendererprop, que debe ser una función. También hemos añadido un elementosapuntalar , Que sustituye perfiles. En nuestrohacer()que ahora llamamos el paso en función de los datos de instancia elemento y el índice. Vamos a hablar más acerca de por qué necesitamos pasaríndiceen un poco. En nuestro padre o de componentes de aplicaciones que ahora hacemos lo siguiente:

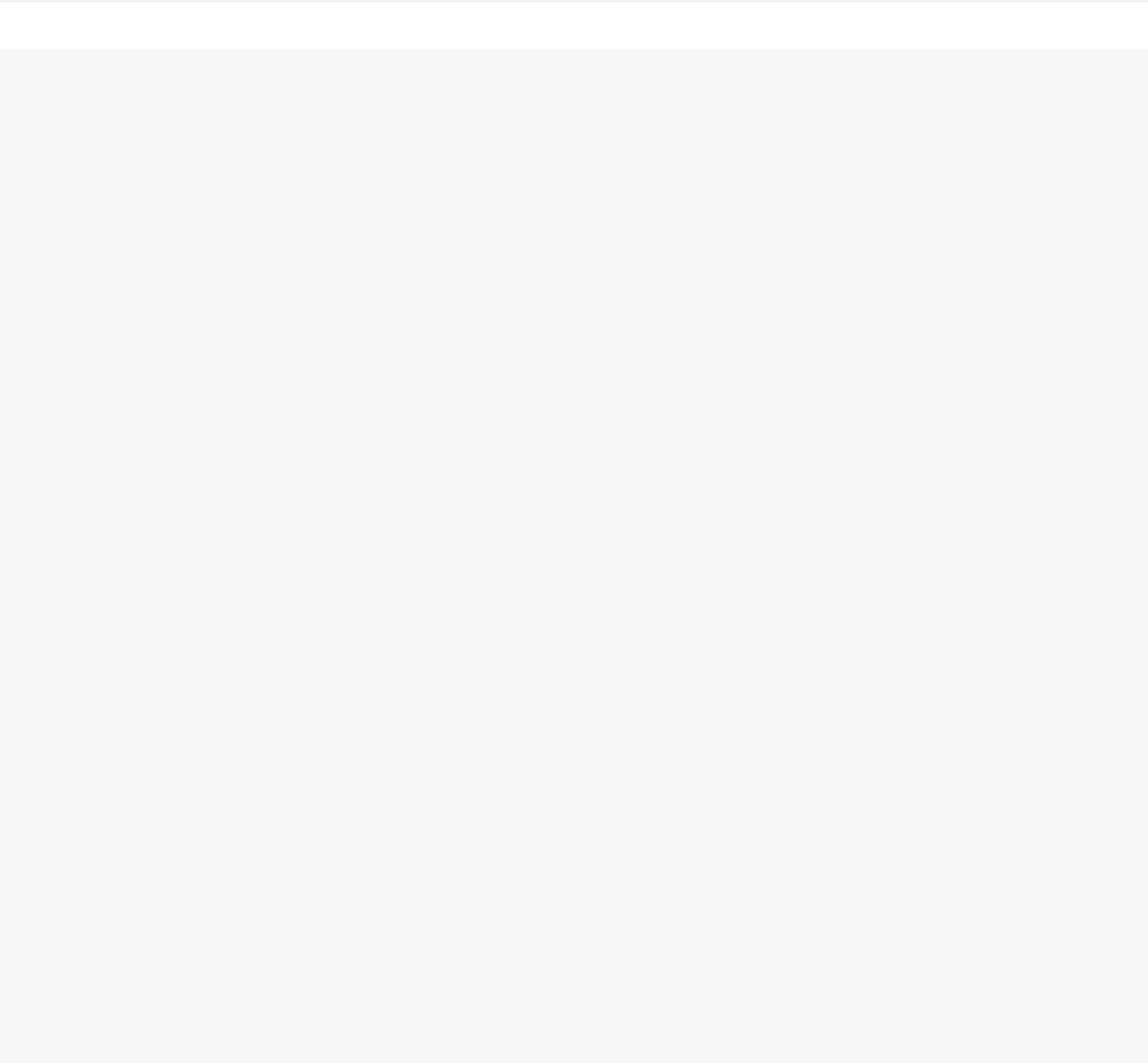


**index.js**



60

Representar el contenido diferente



importar Reaccionar de 'reaccionar';

importar ReactDOM de 'Reaccionar-dom';

importar Lista de './components/List';

importar Perfil de './components/Profile';

importar Mensajes de './components/Posts;

dejar profileData = [...] // pseudo código, esto tiene todos nuestros datos de perfil dejar postsData = [...] // pseudo código, esto tiene todo nuestro datos post

clase Aplicación se extiende Reaccionar.Componente {

renderProfile (perfil, clave) {

regreso (<Perfil {...} perfil clave = {clave} />);

}

renderPosts (postes, clave) {

retorno (<Mensajes {...} puesto clave = {clave} />);

}

render () {

regreso (

<Div>

<Los elementos de lista = {} profileData itemRenderer = {} this.renderProfile /> <Elementos de la lista = {} postsData itemRenderer = {} this.renderPosts />

</ Div>

);

}

}

ReactDOM.render (<App />, document.getElementById ( 'punto de montaje'));

En index.jsprestamos a cabo dos componentes List diferentes. Por primera lista, pasamos en nuestros datos de perfil y nuestrarenderProfileMétodo de referencia. Al igual que cualquier Reaccionar acción (comoal hacer clic ) Pasamos la referencia al método y en realidad no llamamos al método. Para el En segundo lugar, se pasa en los datos de mensajes y la renderPosts Método de referencia.



Cuando las listas de render, el mapa método llama a cualquiera renderProfile () o renderPosts () con cada elemento de datos y el índice actual.



**Reaccionar claves y conjuntos de componentes**

La razón por la que pasa el índice es que necesitamos para generar una clave única para cada elemento de la lista. Cuando la descarga de renderizado a un método, ya no obtenemos incorporado en la capacidad para generar las claves para nosotros reaccionar.

Reaccionar teclas de componente se utilizan para Componente Reconciliación:



61

Representar el contenido diferente



La reconciliación es el proceso por el cual actualiza el DOM Reaccionar con cada nuevo pase rendir ...

... La situación se complica cuando los niños se barajan alrededor (como en los resultados de búsqueda) o si se añaden nuevos componentes en la parte delantera de la lista (como en arroyos). En estos casos en los que la identidad y el estado de cada niño deben ser mantenidos a través de rendir pases, se puede identificar de forma única a cada niño mediante la asignación de una clave

Cuando Reaccionar concilie esos niños con llave, se asegurará de que cualquier niño con clave será reordenada (en lugar de clobbered) o destruido (en lugar de volver a utilizar).

* [Reaccionar Reconciliación Niño](https://facebook.github.io/react/docs/multiple-components.html#child-reconciliation)

Si no establecemos una clave cuando se genera dinámicamente los niños (a través de nuestra itemRenderer método)

obtendríamos la siguiente advertencia:



Advertencia: Cada niño en una matriz o iterador debe dotarse de un instrumento único "clave". Compruebe el método de renderLista. Ver<https://fb.me/react-warning-keys>para más información.



La solución más rápida es pasar en el índice de los datos, pero esto puede no ser la solución ideal. El problema con este enfoque es que se genera una clave basada en el orden material. Sería mejor utilizar un únicocarné de identidadeso está definida en el conjunto de datos. Otra opción es generar un código hash o algún otro identificador único que refleje el contenido del elemento de datos.



Al tener un identificador basado en el contenido de datos en lugar de orden, podemos ayudar a optimizar la prestación de componentes. Cuando mostramos listas parciales, como el filtrado o la clasificación, si la llave se basa en el contenido y no el fin, Reaccionar sabe que no tiene que generar un nuevo elemento de los datos. Simplemente tiene que cambiar el orden de los elementos.

**Procesador de elementos de componentes**

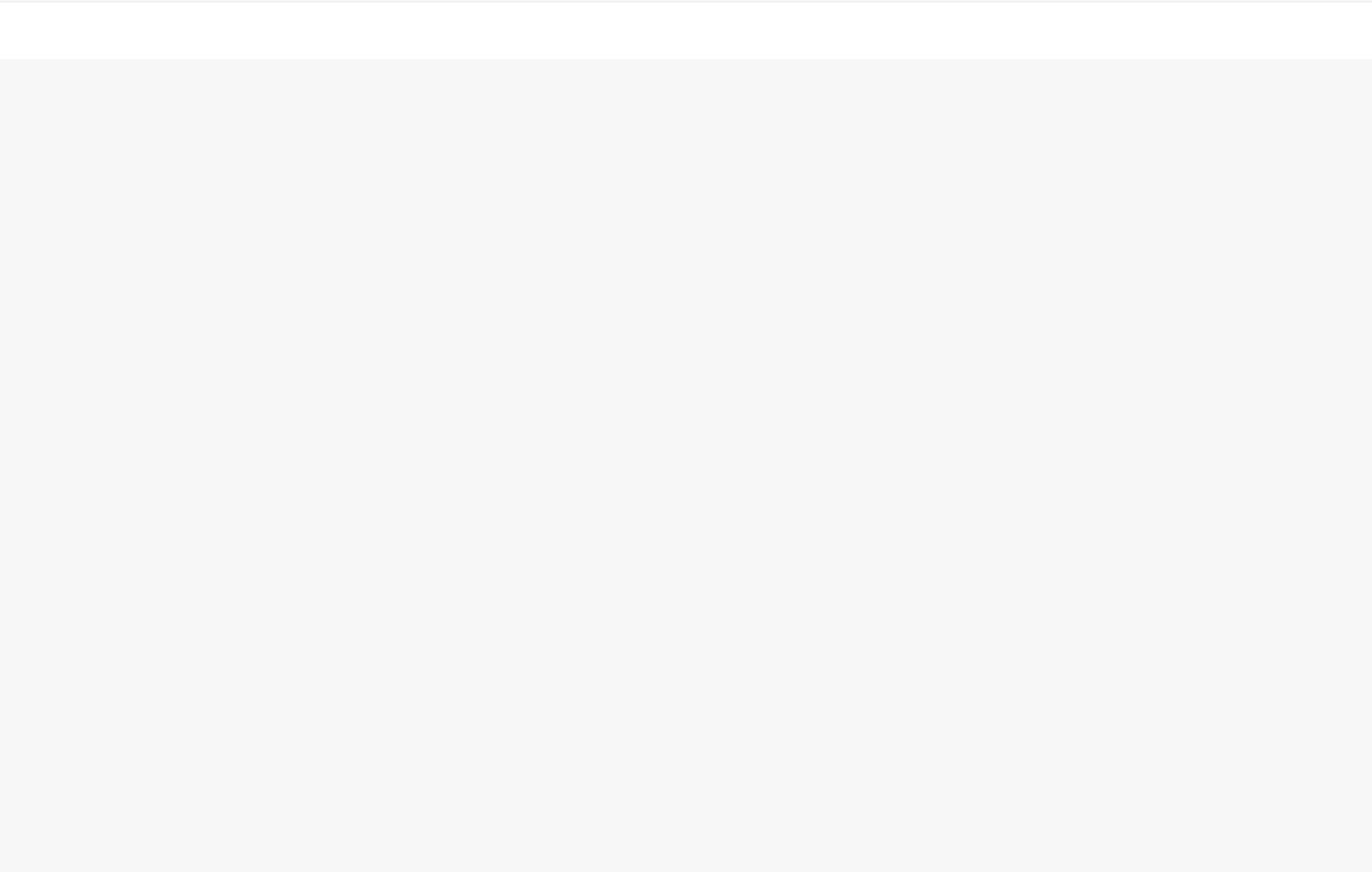
Otra opción para el manejo de procesadores dinámicos, es utilizar una referencia de clase de componente. Este proceso es similar a la que pasa en una función. En lugar de la descarga de la prestación con el valor de retorno de un método creamos un Reaccionar Elemento del componente y pasan en la configuración.

**List.js**



62

Representar el contenido diferente



importar Reaccionar de 'reaccionar';

importar Perfil de './Perfil';

clase Lista se extiende Reaccionar.Componente {

render () {

regreso (

<Ul>

{This.props.profile.map ((perfil, index) => {

dejar que newProps = Object.assign ({clave: index}, perfil);

volver React.createElement (this.props.itemRenderer, newProps); })}

</ Ul>

);

}

}

List.propTypes = {itemRenderer: React.PropTypes.func}; List.defaultProps = {perfil: [], itemRenderer: Perfil};por defecto de exportación Lista;

En esta versión de la lista de componentes, creamos un nuevo elemento utilizando la Reaccionar this.props.itemRenderer como el tipo de clase de componente. Generamos una newProps objeto que se suma el llave a los datos de perfil y pasar esta al elemento como su accesorios .



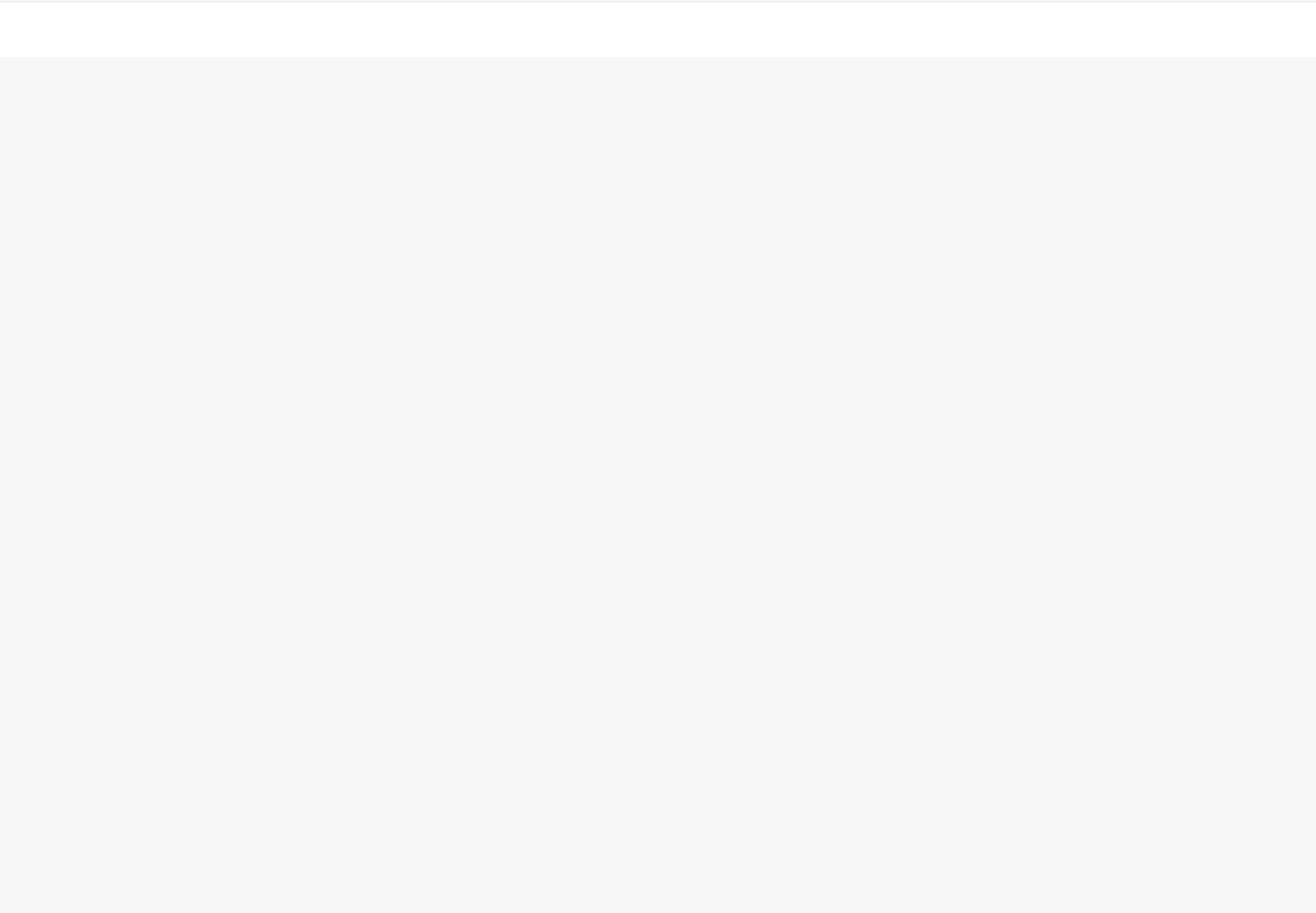
Debido a que definimos un procesador de elementos por defecto de Perfil en el defaultProps podemos actualizar propTypes para hacer itemRenderer un parámetro opcional. Para utilizar esta versión de la lista de nuestra index.js ahora se ve así:

**index.js**



63

Representar el contenido diferente



importar Reaccionar de 'reaccionar';

importar ReactDOM de 'Reaccionar-dom';

importar Lista de './components/List';

importar Perfil de './components/Profile';

importar Enviar de './components/Post';

dejar profileData = [...] // pseudo código, esto tiene todos nuestros datos de perfil dejar postsData = [...] // pseudo código, esto tiene todo nuestro datos post

clase Aplicación se extiende Reaccionar.Componente {

render () {

regreso (

<Div>

<lista de artículos = {} profileData />

<Los elementos de lista = {} postsData itemRenderer = {Mensaje} /> </ div>

);

}

}

ReactDOM.render (<App />, document.getElementById ( 'punto de montaje'));

Ya que tenemos un procesador de elementos por defecto (el perfil de componentes), la primera versión de la lista sólo necesita los datos del perfil. En la segunda versión, cambiamos el tipo de procesador pasando en nuestro componente y pasando los datos del artículo.

Cuando nuestra lista hace que los datos que crea ahora un elemento de la Reaccionar itemRenderervalor y pases en el elemento de datos actual. A[DevelopmentArc](http://developmentarc.com), Hemos encontrado usando una clase Reaccionar es un enfoque mucho más limpio para el desarrollo de elementos de la interfaz reemplazables.



***Hasta la próxima***: [Componentes de orden superior](#page65)



64

Componentes de orden superior



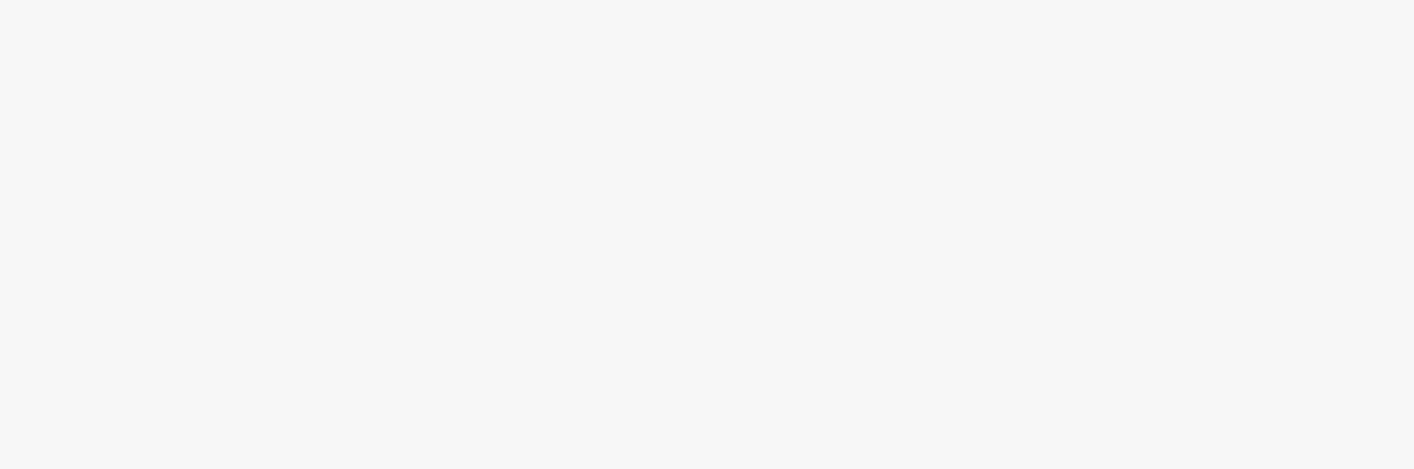
**Componentes de orden superior**

El último patrón de composición de componentes que examinaremos en esta sección es superior componentes de orden (HOC). Como[Dan Abramov discute](https://medium.com/@dan_abramov/mixins-are-dead-long-live-higher-order-components-94a0d2f9e750#.b74nxbqew), Componentes de orden superior se propuso por primera vez [Sebastián Markbåge](https://gist.github.com/sebmarkbage/ef0bf1f338a7182b6775) en una esencia. La idea central de HOC es definir una función, que se pasa uno o más componentes. Esta función genera y devuelve un nuevo componente, que es una envoltura alrededor de la pasado en el componente (s).

La necesidad de HOC se produjo con Reaccionar a pasar a apoyar las clases ES6 y la falta de apoyo mixin con la nueva sintaxis JavaScript clase. Para hacer frente a este cambio, un nuevo patrón era necesario definir para reemplazar mixins. Típicamente, mixins ADD / anulan funcionalidad alrededor de la[Ciclo de Vida de componentes](#page6) y permitir el intercambio de código reutilizable de una manera elegante. Sin mixin apoyo en la ES6, se requiere que el patrón HOC.

**Un ejemplo de grupo forma**

Para nuestro ejemplo HOC, vamos a crear una función para envolver un componente de un grupo de formulario personalizado con un opcional <Label>campo. El objetivo de la HOC es que nos permitirá crear dos salidas, con y sin una etiqueta:



<! - Con una etiqueta ->

<Div class ="Forma-grupo">

<Etiqueta de clase ="Forma de etiqueta" para ="nombre de pila">Nombre de pila:</ Label> <input type ="texto" name ="nombre de pila" />

</ Div>

<! - Sin una etiqueta ->

<Div class ="Forma-grupo">

<Input type ="texto" name ="apellido" />

</ Div>

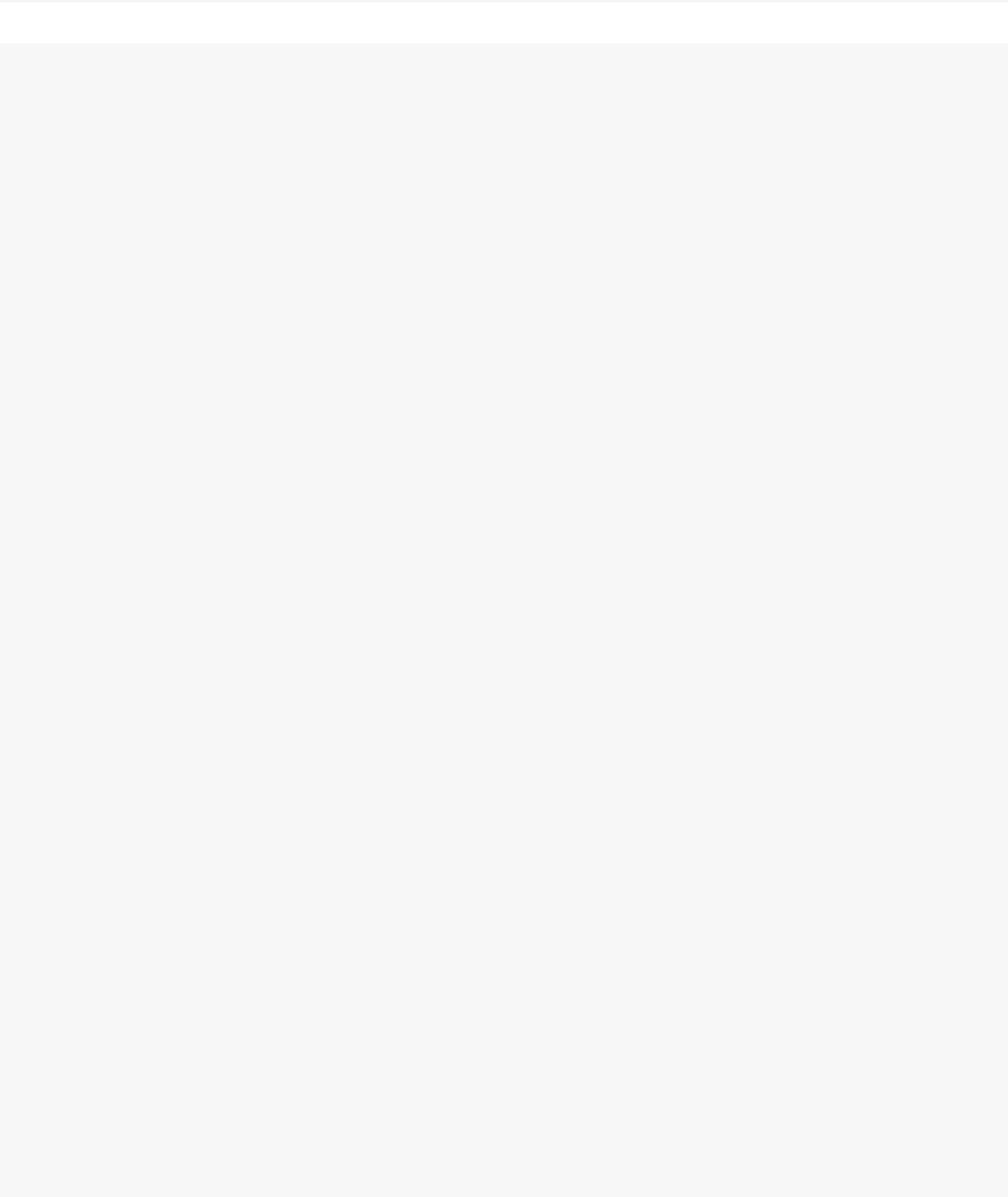
Debido a que esto podría convertirse en una tarea común, podemos usar el patrón HOC para generar nuestra envoltura de forma grupo y dejar que decida si debe inyectarse la etiqueta o no.

**formGroup.js**



sesenta y cinco

Componentes de orden superior



importar Reaccionar de 'reaccionar';

importar {} IsString de 'Lodash';

función formGroup(Componente, config) {

const FormGroup = React.createClass ({

\_\_renderLabel () {

* comprobar si el valor pasado es una cadena usando Lodash # isString Si (IsString (esta.props.label)) {

regreso(

<Label className ="Forma de etiqueta" htmlFor ={ this.props.name}>

{} This.props.label </ Label>

);

}

},

\_\_renderElement () {

* Tenemos que ver si pasamos por un componente o un elemento
* tal como Perfil vs. <input type = "text" />

Si (React.isValidElement (Componente)) regreso React.cloneElement (Componente, esta.pag ROPS);

regreso(<Componente {...} this.props />);

},

render () {

regreso(

<Div className = "forma-grupo">

{.\_\_ este renderLabel ()} {esta .\_\_ renderElement ()}

</ Div>

);

}

});

retorno (<FormGroup {...} config />);

}

formGroup predeterminado de exportación;

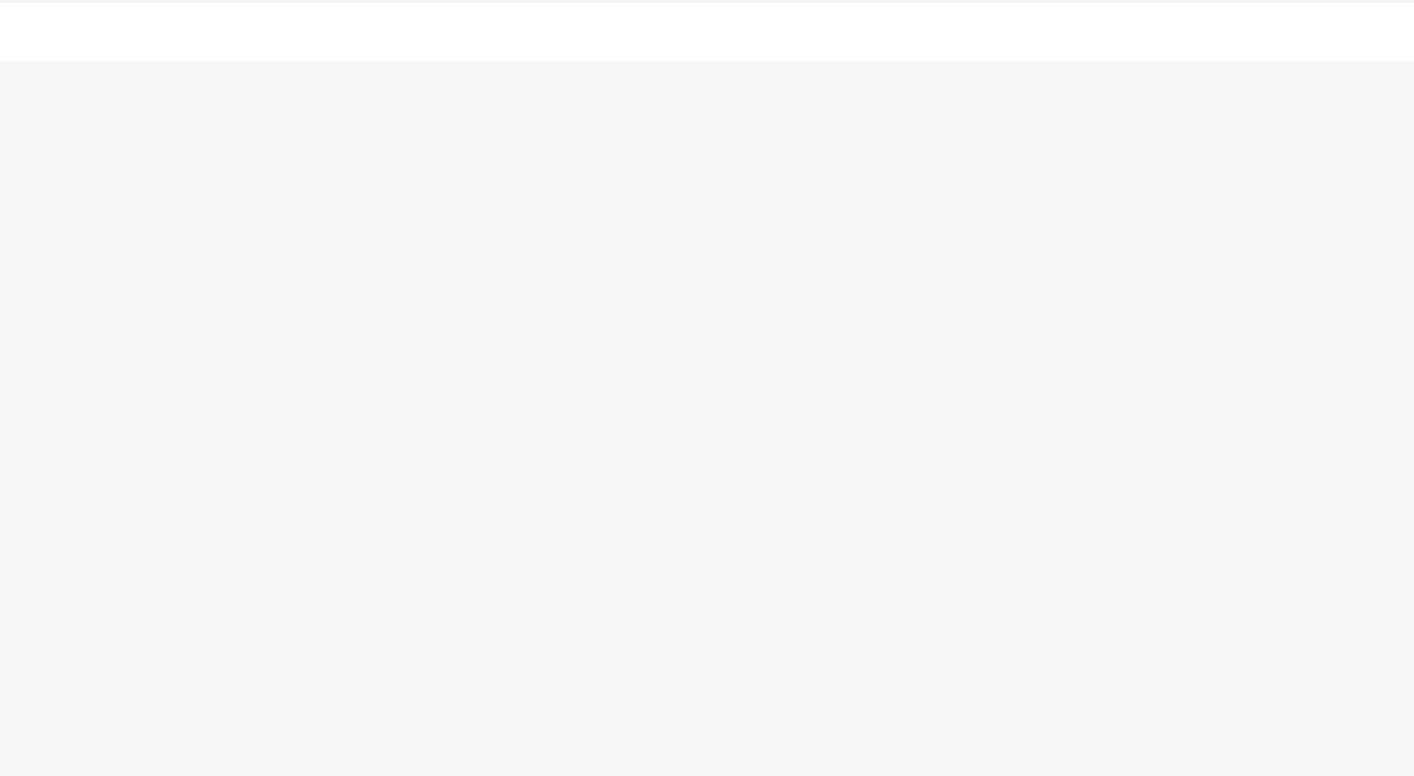
Para utilizar esta HOC podemos hacer lo siguiente:

**index.js**



66

Componentes de orden superior



importar Reaccionar de 'reaccionar';

importar ReactDOM de 'Reaccionar-dom';

importar formGroup de './higherOrderComponents/formGroup';

dejar MyComponent = React.createClass ({

render () {

regreso (

<Div>

{FormGroup (<input type = "text" />, {label: 'Nombre:', nombre: 'primerNombre'}

)}

</ Div>

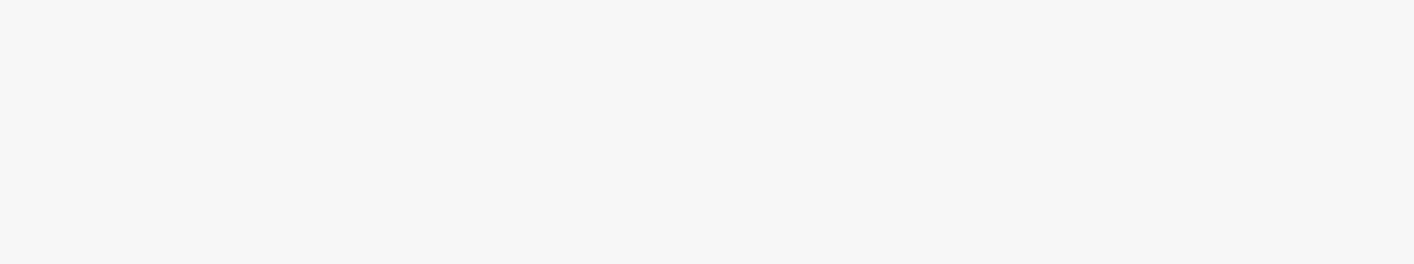
);

}

});

ReactDOM.render (<MyComponent />, document.getElementById ( 'punto de montaje'));

Vamos a examinar el código de seguridad. La primera cosa que hacemos para la HOC es crear una función llamadaformGroup que toma dos argumentos: Componente y config .



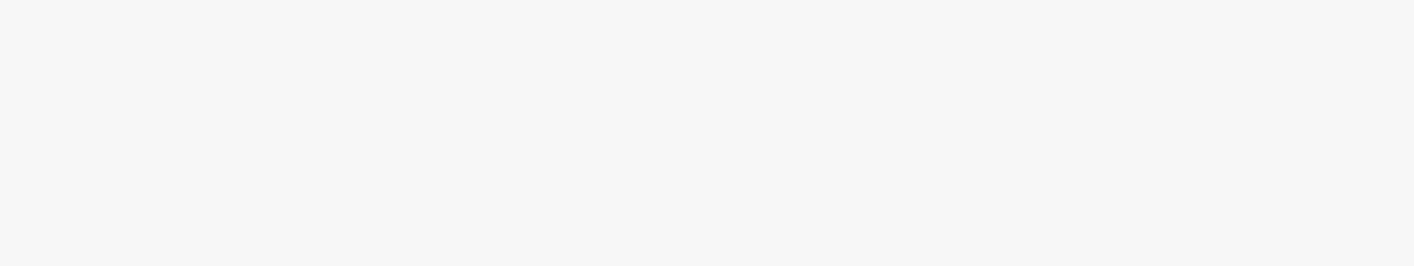
función formGroup(Componente, config) {

...

}

por defecto de exportación formGroup;

El componente será la instancia queremos envolver en nuestro grupo formulario. En la función, se crea un nuevo componente Reaccionar y luego devolver una instancia utilizando el elementoconfig como accesorios.



const FormGroup = React.createClass ({

...

});

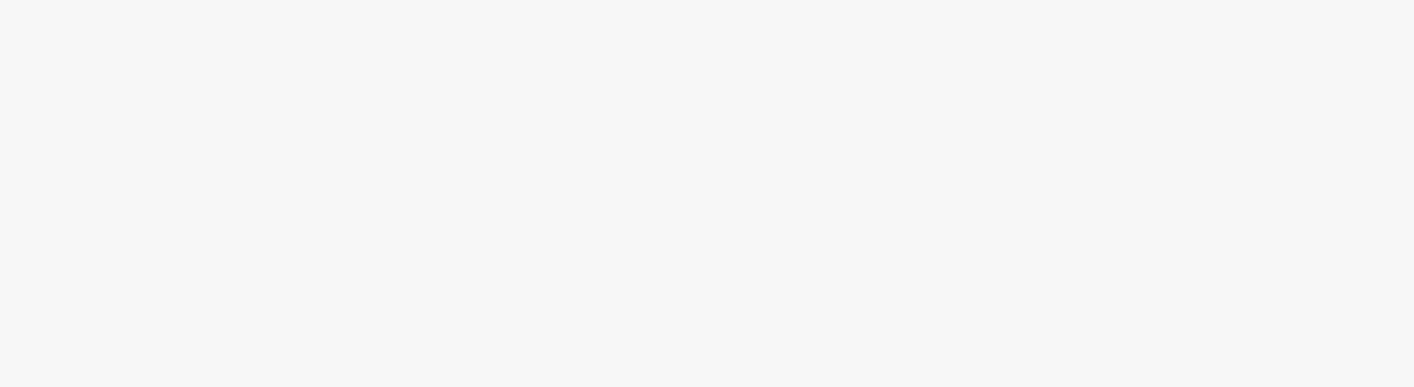
regreso(<FormGroup {...} config />);

Nos aprovechamos de la [operador de difusión ES6](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator) para pasar en nuestro configobjeto como el apoyos para el elemento JSX generado. En nuestrohacer() Método creamos el grupo de formularios <Div> y luego hacer que nuestra etiqueta opcional y el contenido de componentes.



67

Componentes de orden superior



render () {

regreso(

<Div className ="Forma-grupo">

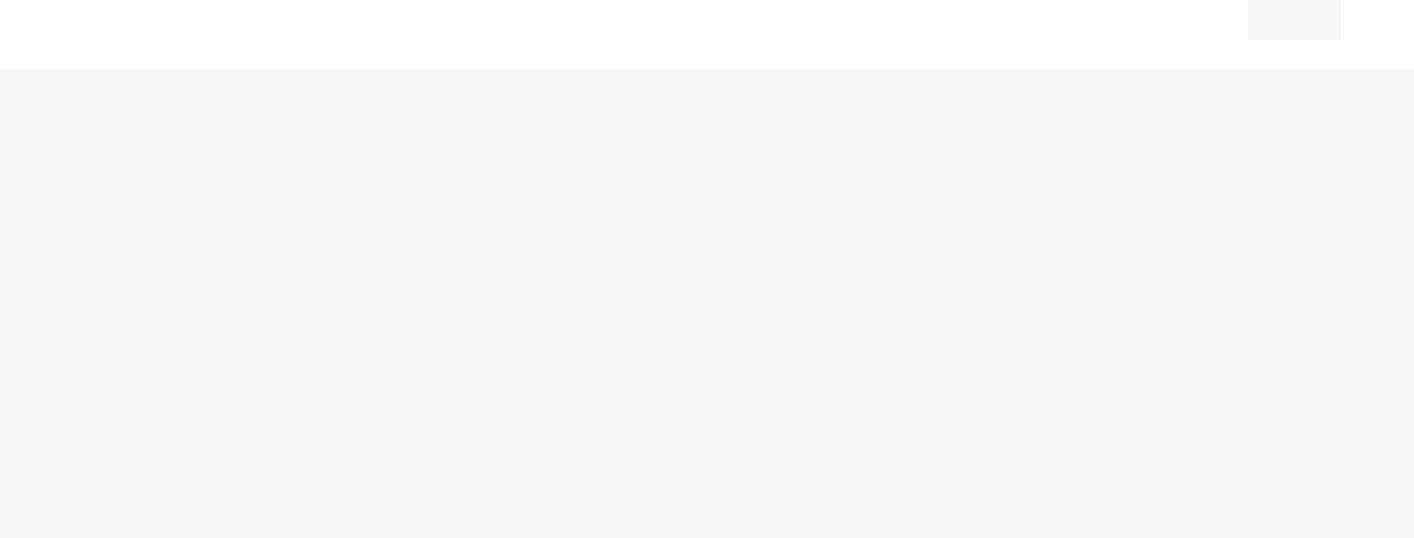
{.\_\_ este renderLabel ()} {esta .\_\_ renderElement ()}

</ Div>

);

}

En nuestro \_\_renderLabel () método[1](#page65) usamos la [Lodash isString](https://lodash.com/docs#isString) método para comprobar si el valor de la etiqueta es una cadena. Si es así, que prestamos nuestra etiqueta de elemento DOM, de lo contrario volvemosnulo .



\_\_renderLabel () {

* comprobar si el valor pasado es una cadena usando Lodash # isString Si (IsString (esta.props.label)) {

regreso(

<Label className ="Forma de etiqueta" htmlFor ={ this.props.name}> {} This.props.label

</ Label>

);

}

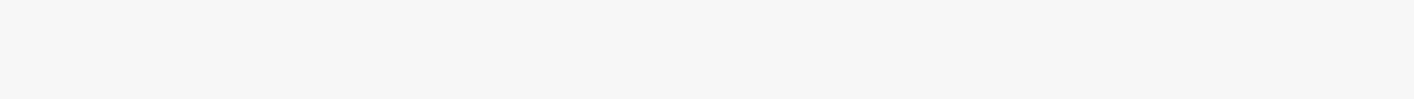
},

Porque nulo no hace a la interfaz de usuario nativa en Reaccionar, así es como hacemos la <Label> opcional basado en el valor pasado.



Por último, hemos tenido que añadir una comprobación para determinar qué tipo se pasó a nuestra función HOC para el componente. Este es un control importante porque queremos apoyar tanto Reaccionar componentes y elementos.

En nuestro index.js estamos pasando en:

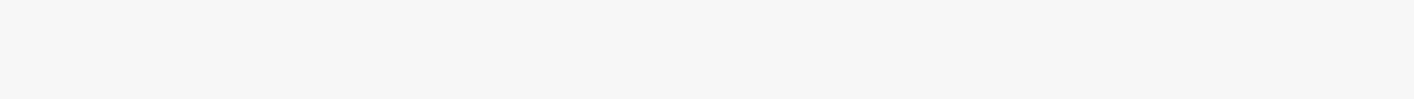


formGroup (<Input type ="texto" />, {Label: 'Nombre:', nombre: 'primerNombre'})

Debido a que estamos utilizando para generar nuestra JSX <Input /> HOC recibirá un Elemento.



Sin embargo, si utilizamos nuestro componente perfil, es posible que no desee utilizar JSX:



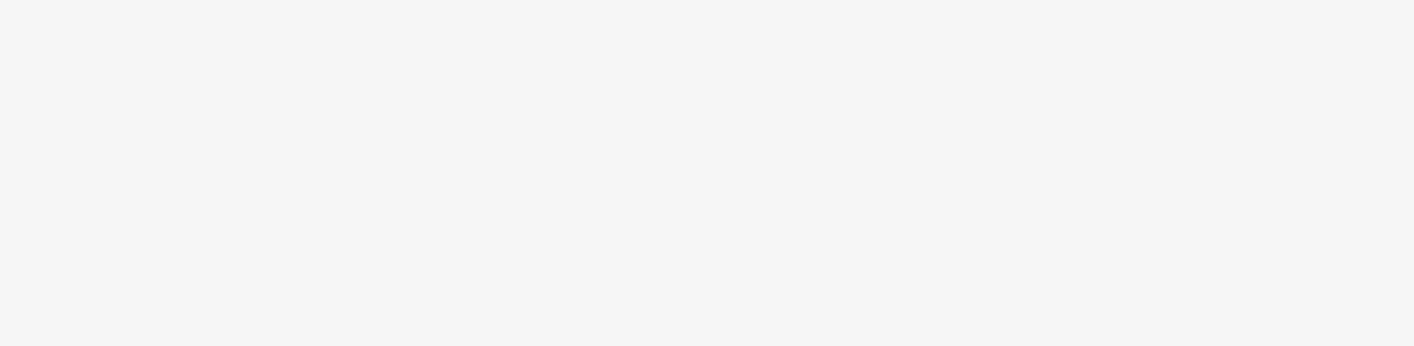
formGroup (Perfil, {label: 'Nombre de pila:', nombre: 'nombre de pila' })

Para apoyar ambas opciones y transmitir la accesorios , Se utiliza el \_\_renderElement () método para manejar la generación de inspección y de salida:



68

Componentes de orden superior



\_\_renderElement () {

* Tenemos que ver si pasamos por un componente o un elemento
* tal como Perfil vs. <input type = "text" />

Si (React.isValidElement (Componente)) regreso React.cloneElement (Componente, esta.accesorios );

regreso( <Componente {...} this.props />);

},

Si la instancia de componente es un elemento, que [clonar el elemento](https://facebook.github.io/react/docs/top-level-api.html#react.cloneelement) y transmitir los nuevos accesorios.

De lo contrario, se genera un nuevo elemento utilizando JSX y la aprobada en Reaccionamos de componentes.

Este ejemplo HOC es sólo la punta del iceberg cuando se trata de componentes de envoltura de auto-generación. Usando este modelo, podemos aprovechar el[métodos de ciclo de vida de los componentes](#page6), Podemos tomar decisiones más complejas basadas en los datos, que se pueden registrar para tiendas u otros eventos, y muchas otras combinaciones posibles.

[Para ejemplos más en profundidad es muy recomendable la lectura de Dan Abramov *Son mixins*](https://medium.com/@dan_abramov/mixins-are-dead-long-live-higher-order-components-94a0d2f9e750#.9y0gg1ix5) [*Muerto. Composición Long Live*y @ franlplant de](https://medium.com/@dan_abramov/mixins-are-dead-long-live-higher-order-components-94a0d2f9e750#.9y0gg1ix5)[*Reaccionar Superior componentes de orden de profundidad*](https://medium.com/@franleplant/react-higher-order-components-in-depth-cf9032ee6c3e#.d38rbnsu8)



* En estos ejemplos estamos anteponiendo nuestros métodos con \_\_para reflejar que estos son métodos de componentes internos. Esto es completamente opcional y es sólo nuestra sintaxis estilo preferido.



69

Sobre los autores



**Sobre los autores**

**James Polanco**

[@jamespolanco](https://twitter.com/jamespolanco)

En su papel como co-fundador y CTO de DevelopmentArc®, James se esfuerza por tomar poderosas ideas de negocios e integrarlos en experiencias tecnológicas elegantes para ayudar a los clientes a alcanzar sus objetivos estratégicos generales. Desde 1996, James ha ayudado a planificar, guía y crear soluciones interactivas y basadas en web para empresas como Adobe, Toyota Motor Sports, BlueKai, VMWare, Macromedia y DHAP digital. James también es un orador internacional, presentando en las implementaciones y los procesos tecnológicos y un autor publicado en el tema de desarrollo de productos y el equipo de la compañía flujos de trabajo completos.

**Aaron Pedersen**

[@aaronpedersen](https://twitter.com/aaronpedersen)

Como co-fundador y CEO de DevelopmentArc®, una empresa boutique de firma de desarrollo y los padres de Pedanco, la pasión de Aaron Pedersen está en ayudar a las empresas a optimizar proceso de toma de equipos trabajan de manera más efectiva a través de soluciones tecnológicas innovadoras. Un autor publicado, experto orador, y codiciados consultor y entrenador, Aaron trabaja con una amplia gama de empresas, desde empresas de Fortune 500 y compañías de hospitalidad multi-cadena de marcas emergentes y nuevas empresas de semillas redondas como Toyota Motor Sports, DHAP Digital , Adobe, KitchenNetwork y FitStar.



70