EBook Gratis

APRENDIZAJE React

Free unaffiliated eBook created from
Stack Overflow contributors.

#reactjs

Tabla de contenido

Acerca de ... 1

Capítulo 1: Empezando con React ... 2

Observaciones ... 2

Versiones ... 2

Examples ... 3

Instalación o configuración ... 3

Hola componente mundial ... 4

Hola Mundo ... 6

¿Qué es ReactJS? ... 7

Hola mundo con funciones sin estado ... 8

Por ejemplo: ... 8

Crear React App ... 9

Instalación ... 9

Configuración ... 10

Alternativas ... 10

Fundamentos absolutos de la creación de componentes reutilizables ... 10

Componentes y accesorios ... 10

Capítulo 2: Actuación ... 13

Examples ... 13

Lo básico - HTML DOM vs Virtual DOM ... 13

Algoritmo de diferencia de React ... 14

consejos y trucos ... 14

Medición de rendimiento con ReactJS ... 15

Capítulo 3: Apoyos en reaccionar ... 16

Observaciones ... 16

Examples ... 16

Introducción ... 16

Accesorios por defecto ... 17

PropTypes ... 17

Pasando los puntales utilizando el operador extendido ... 19

Props.children y composición de componentes 20
Detección del tipo de componentes para niños. 21
Capítulo 4: Cómo configurar un webpack básico, reaccionar y babel. 22
Observaciones 22
Examples 23
Cómo construir una tubería para un "Hola mundo" personalizado con imágenes. 23
Capítulo 5: Cómo y por qué usar llaves en React. 28
Introducción 28
Observaciones 28
Examples 28
Ejemplo básico 28
Capítulo 6: Componentes 30
Observaciones 30
Examples 30
Componente basico 30
Componentes de anidación 31
1. Anidar sin usar niños. 32
Pros 32
Contras 32
Bien si 32
2. Anidar usando niños. 32
Pros 33
Contras 33
Bien si 33
3. Anidar utilizando puntales. 33
Pros 34
Contras 34
Bien si 34
Creando Componentes 34
Estructura basica 34
Componentes funcionales sin estado 35

Componentes de estado ... 35
Componentes de orden superior ... 36
trampas setState ... 37
Accesorios ... 39
Estados del componente - Interfaz de usuario dinámica ... 40
Variaciones de los componentes funcionales sin estado ... 41
Capítulo 7: Componentes de orden superior ... 43
Introducción ... 43
Observaciones ... 43
Examples ... 43
Componente de orden superior simple ... 43
Componente de orden superior que comprueba la autenticación ... 44
Capítulo 8: Componentes funcionales sin estado ... 46
Observaciones ... 46
Examples ... 46
Componente funcional sin estado ... 46
Capítulo 9: Comunicación Entre Componentes ... 50
Observaciones ... 50
Examples ... 50
Componentes de padres a hijos ... 50
Componentes del niño al padre ... 51
Componentes no relacionados ... 51
Capítulo 10: Comunicar Entre Componentes ... 53
Examples ... 53
Comunicación entre componentes funcionales sin estado ... 53
Capítulo 11: Configuración de React Ambiente ... 56
Examples ... 56
Componente Reactivo Simple ... 56
Instalar todas las dependencias ... 56
Configurar webpack ... 56
Configurar babel ... 57

Archivo HTML para usar el componente reaccionar 57
Transpilar y agrupar su componente 57
Capítulo 12: Estado en reaccionar 58
Examples 58
Estado basico 58
setState () 58
Usando setState() con un Objeto como updater 59
Usando setState() con una función como updater 59
Llamar a setState() con un objeto y una función de devolución de llamada 60
Antipattern común 60
Estado, Eventos y Controles Gestionados 62
Capítulo 13: Formularios y comentarios del usuario 64
Examples 64
Componentes controlados 64
Componentes no controlados 64
Capítulo 14: Instalación 66
Examples 66
Configuración simple 66
Configurando las carpetas 66
Configurando los paquetes 66
Configuración de webpack 66
Probando la configuración 67
Utilizando webpack-dev-server 68
Preparar 68
Modificando webpack.config.js 68
Capítulo 15: Instalación de React, Webpack y Typescript. 70
Observaciones 70
Examples 70
webpack.config.js 70
El cargador 70
Resolver las extensiones TS 70

tsconfig.json 71
include 71
compilerOptions.target 71
compilerOptions.jsx 71
compilerOptions.allowSyntheticDefaultImports 71
Mi primer componente 72
Capítulo 16: Introducción a la representación del lado del servidor 73
Examples 73
Componentes de renderizado 73
renderToString 73
renderToStaticMarkup 73
Capítulo 17: JSX 74
Observaciones 74
Examples 75
Puntales en JSX 75
Expresiones de JavaScript 75
Literales de cuerda 75
Valor predeterminado de accesorios 75
Atributos de propagación 76
Niños en JSX 76
Literales de cuerda 76
Niños jsx 77
Expresiones de JavaScript 77
Funciones como niños 78
Valores ignorados 78
Capítulo 18: Reaccionar con redux 80
Introducción 80
Observaciones 80
Examples 80
Usando Connect 80

Capítulo 19: Reaccionar enrutamiento ... 82
Examples ... 82
Ejemplo del archivo Routes.js, seguido del uso de Router Link en el componente ... 82
Reaccionar enrutamiento asíncrono ... 83
Capítulo 20: Reaccionar formas ... 84
Examples ... 84
Componentes controlados ... 84
Capítulo 21: Reaccionar herramientas ... 86
Examples ... 86
Campo de golf ... 86
Capítulo 22: Reaccionar llamada AJAX ... 87
Examples ... 87
Solicitud HTTP GET ... 87
Ajax en Reaccionar sin una biblioteca de terceros, también conocido como VanillaJS ... 88
Solicitud HTTP GET y bucle a través de los datos ... 88
Capítulo 23: React Boilerplate [React + Babel + Webpack] ... 90
Examples ... 90
Configurando el proyecto ... 90
proyecto react-starter ... 92
Capítulo 24: React Component Lifecycle ... 95
Introducción ... 95
Examples ... 95
Creación de componentes ... 95
getDefaultProps() (solo ES5) ... 95
getInitialState() (solo ES5) ... 95
componentWillMount() (ES5 y ES6) ... 96
render() (ES5 y ES6) ... 96
componentDidMount() (ES5 y ES6) ... 96
Sintaxis de ES6 ... 97
Reemplazo de getDefaultProps() ... 97
Sustitución de getInitialState() ... 98

Actualización de componentes 98
componentWillReceiveProps(nextProps) 98
shouldComponentUpdate(nextProps, nextState) 98
componentWillUpdate(nextProps, nextState) 99
render() 99
componentDidUpdate(prevProps, prevState) 99
Eliminación de componentes 99
componentWillUnmount() 99
React Component Container 100
Método de ciclo de vida llamado en diferentes estados 101
Capítulo 25: React.createClass vs extiende React.Component 103
Sintaxis 103
Observaciones 103
Examples 103
Crear Componente React 103
React.createClass (en desuso) 103
React.Component 103
Declare Props y PropTypes por defecto 104
React.createClass 104
React.Component 105
Establecer estado inicial 106
React.createClass 106
React.Component 106
Mixins 107
React.createClass 107
React.Component 107
"este contexto 107
React.createClass 107
React.Component 108
Caso 1: Enlace en línea: 108

Caso 2: Enlace en el constructor de la clase ... 109
Caso 3: utilizar la función anónima de ES6 ... 109
ES6 / Reacciona la palabra clave "this" con ajax para obtener datos del servidor ... 109
Capítulo 26: Soluciones de interfaz de usuario ... 111
Introducción ... 111
Examples ... 111
Panel básico ... 111
Panel ... 111
Lengüeta ... 112
Grupo de paneles ... 112
Aclaración ... 113
Vista de ejemplo con `PanelGroup`s ... 114
Capítulo 27: Teclas en reaccionar ... 116
Introducción ... 116
Observaciones ... 116
Examples ... 116
Usando el id de un elemento ... 116
Usando el índice de matriz ... 117
Capítulo 28: Usando React con Flujo ... 118
Introducción ... 118
Observaciones ... 118
Examples ... 118
Uso de Flow para verificar tipos de accesorios de componentes funcionales sin estado ... 118
Usando Flow para verificar los tipos de accesorios ... 118
Capítulo 29: Usando ReactJS con jQuery ... 119
Examples ... 119
Reacciona con jQuery ... 119
Capítulo 30: Usando ReactJS en forma de flujo ... 121
Introducción ... 121
Observaciones ... 121
Examples ... 121

Flujo de datos ... 121
Revertido ... 122
Capítulo 31: Utilizando ReactJS con Typescript ... 123
Examples ... 123
Componente ReactJS escrito en Typescript ... 123
Componentes de Stateless React en Typescript ... 124
Instalación y configuración ... 124
Componentes sin estado y sin propiedad ... 125
Creditos ... 127

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: react

It is an unofficial and free React ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official React.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con React

Observaciones

React es una biblioteca declarativa de JavaScript basada en componentes que se utiliza para crear interfaces de usuario.

Para lograr las funcionalidades de MVC framework en React, los desarrolladores lo utilizan junto con el sabor de Flux de su elección, por ejemplo, Redux .

Versiones

Versión	Fecha de lanzamiento
0.3.0	2013-05-29
0.4.0	2013-07-17
0.5.0	2013-10-16
0.8.0	2013-12-19
0.9.0	2014-02-20
0.10.0	2014-03-21
0.11.0	2014-07-17
0.12.0	2014-10-28
0.13.0	2015-03-10
0.14.0	2015-10-07
15.0.0	2016-04-07
15.1.0	2016-05-20
15.2.0	2016-07-01
15.2.1	2016-07-08
15.3.0	2016-07-29
15.3.1	2016-08-19
15.3.2	2016-09-19

Versión	Fecha de lanzamiento
15.4.0	2016-11-16
15.4.1	2016-11-23
15.4.2	2017-01-06
15.5.0	2017-04-07
15.6.0	2017-06-13

Examples

Instalación o configuración

ReactJS es una biblioteca de JavaScript contenida en un solo archivo `react-<version>.js` que se puede incluir en cualquier página HTML. La gente también suele instalar la biblioteca React DOM `react-dom-<version>.js` junto con el archivo principal React:

Inclusión básica

```
<!DOCTYPE html>
<html>
    <head></head>
    <body>
    <script type="text/javascript" src="/path/to/react.js"></script>
    <script type="text/javascript" src="/path/to/react-dom.js"></script>
    <script type="text/javascript">
        // Use react JavaScript code here or in a separate file
    </script>
    </body>
</html>
```

Para obtener los archivos JavaScript, vaya a la página de instalación de la documentación oficial de React.

React también soporta la sintaxis JSX . JSX es una extensión creada por Facebook que agrega sintaxis XML a JavaScript. Para utilizar JSX, debe incluir la biblioteca de Babel y cambiar `<script type="text/javascript">` a `<script type="text/babel">` para traducir JSX al código Javascript.

```
<!DOCTYPE html>
<html>
    <head></head>
    <body>
    <script type="text/javascript" src="/path/to/react.js"></script>
    <script type="text/javascript" src="/path/to/react-dom.js"></script>
    <script src="https://npmcdn.com/babel-core@5.8.38/browser.min.js"></script>
    <script type="text/babel">
        // Use react JSX code here or in a separate file
    </script>
    </body>
```

```
</html>
```

Instalación a través de npm

También puede instalar React usando npm haciendo lo siguiente:

```
npm install --save react react-dom
```

Para usar React en su proyecto de JavaScript, puede hacer lo siguiente:

```
var React = require('react');
var ReactDOM = require('react-dom');
ReactDOM.render(<App />, ...);
```

Instalación a través de hilo

Facebook lanzó su propio administrador de paquetes llamado Yarn , que también se puede usar para instalar React. Después de instalar Yarn solo necesitas ejecutar este comando:

```
yarn add react react-dom
```

Luego puede usar React en su proyecto exactamente de la misma manera que si hubiera instalado React a través de npm.

Hola componente mundial

Un componente React se puede definir como una clase ES6 que extiende la clase `React.Component` base. En su forma mínima, un componente *debe* definir un método de `render` que especifique cómo se procesa el componente en el DOM. El método de `render` devuelve los nodos React, que pueden definirse utilizando la sintaxis JSX como etiquetas similares a HTML. El siguiente ejemplo muestra cómo definir un componente mínimo:

```
import React from 'react'

class HelloWorld extends React.Component {
    render() {
        return <h1>Hello, World!</h1>
    }
}

export default HelloWorld
```

Un componente también puede recibir `props` . Estas son propiedades pasadas por su padre para especificar algunos valores que el componente no puede conocer por sí mismo; una propiedad también puede contener una función a la que el componente puede llamar después de que se produzcan ciertos eventos; por ejemplo, un botón podría recibir una función para su propiedad `onClick` y llamarla cada vez que se haga clic en ella. Al escribir un componente, se puede acceder a sus `props` a través del objeto `props` en el propio Componente:

```
import React from 'react'
```

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

export default Hello
```

El ejemplo anterior muestra cómo el componente puede representar una cadena arbitraria pasada a la propiedad de `name` por su padre. Tenga en cuenta que un componente no puede modificar los accesorios que recibe.

Un componente puede representarse dentro de cualquier otro componente, o directamente en el DOM si es el componente más alto, usando `ReactDOM.render` y proporcionándole tanto el componente como el nodo DOM donde desea que se represente el árbol React:

```
import React from 'react'
import ReactDOM from 'react-dom'
import Hello from './Hello'

ReactDOM.render(<Hello name="Billy James" />, document.getElementById('main'))
```

A estas alturas ya sabes cómo hacer un componente básico y aceptar `props` . Vamos a llevar esto un paso más allá e introducir el `state` .

Para la demostración, hagamos nuestra aplicación Hello World, muestre solo el primer nombre si se le da un nombre completo.

```
import React from 'react'

class Hello extends React.Component {

    constructor(props){

        //Since we are extending the default constructor,
        //handle default activities first.
        super(props);

        //Extract the first-name from the prop
        let firstName = this.props.name.split(" ")[0];

        //In the constructor, feel free to modify the
        //state property on the current context.
        this.state = {
            name: firstName
        }

    } //Look maa, no comma required in JSX based class defs!

    render() {
        return <h1>Hello, {this.state.name}!</h1>
    }
}
```

```
export default Hello
```

Nota: Cada componente puede tener su propio estado o aceptar su estado principal como prop.

Codepen Enlace a Ejemplo.

Hola Mundo

Sin JSX

Este es un ejemplo básico que utiliza la API principal de React para crear un elemento React y la API React DOM para representar el elemento React en el navegador.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>

    <!-- Include the React and ReactDOM libraries -->
    <script src="https://fb.me/react-15.2.1.js"></script>
    <script src="https://fb.me/react-dom-15.2.1.js"></script>

  </head>
  <body>
    <div id="example"></div>

    <script type="text/javascript">

      // create a React element rElement
      var rElement = React.createElement('h1', null, 'Hello, world!');

      // dElement is a DOM container
      var dElement = document.getElementById('example');

      // render the React element in the DOM container
      ReactDOM.render(rElement, dElement);

    </script>

  </body>
</html>
```

Con jsx

En lugar de crear un elemento React a partir de cadenas, se puede usar JSX (una extensión de Javascript creada por Facebook para agregar sintaxis XML a JavaScript), que permite escribir

```
var rElement = React.createElement('h1', null, 'Hello, world!');
```

como el equivalente (y más fácil de leer para alguien familiarizado con HTML)

```
var rElement = <h1>Hello, world!</h1>;
```

El código que contiene JSX debe incluirse en una etiqueta `<script type="text/babel">` . Todo lo que se encuentre dentro de esta etiqueta se transformará a Javascript simple mediante la biblioteca de Babel (que debe incluirse además de las bibliotecas React).

Así que finalmente el ejemplo anterior se convierte en:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>

    <!-- Include the React and ReactDOM libraries -->
    <script src="https://fb.me/react-15.2.1.js"></script>
    <script src="https://fb.me/react-dom-15.2.1.js"></script>
    <!-- Include the Babel library -->
    <script src="https://npmcdn.com/babel-core@5.8.38/browser.min.js"></script>

  </head>
  <body>
   <div id="example"></div>

    <script type="text/babel">

      // create a React element rElement using JSX
      var rElement = <h1>Hello, world!</h1>;

      // dElement is a DOM container
      var dElement = document.getElementById('example');

      // render the React element in the DOM container
      ReactDOM.render(rElement, dElement);

    </script>

  </body>
</html>
```

¿Qué es ReactJS?

ReactJS es una biblioteca de front-end de código abierto, basada en componentes, responsable solo de la **capa** de **visualización** de la aplicación. Es mantenido por Facebook.

ReactJS utiliza un mecanismo virtual basado en DOM para completar los datos (vistas) en HTML DOM. El DOM virtual funciona rápidamente, ya que solo cambia los elementos individuales del DOM en lugar de volver a cargar el DOM completo cada vez

> Una aplicación React se compone de varios **componentes** , cada uno responsable de generar una pieza pequeña y reutilizable de HTML. Los componentes se pueden anidar dentro de otros componentes para permitir que las aplicaciones complejas se construyan a partir de bloques de construcción simples. Un componente también puede mantener un estado interno; por ejemplo, un componente TabList puede almacenar una variable correspondiente a la pestaña actualmente abierta.

React nos permite escribir componentes utilizando un lenguaje específico de dominio llamado

JSX. JSX nos permite escribir nuestros componentes utilizando HTML, mientras mezclamos eventos de JavaScript. React lo convertirá internamente en un DOM virtual, y finalmente generará nuestro HTML para nosotros.

Reaccionar " *reacciona* " a los cambios de estado en sus componentes de forma rápida y automática para volver a enviar los componentes en el DOM de HTML utilizando el DOM virtual. El DOM virtual es una representación en memoria de un DOM real. Al realizar la mayor parte del procesamiento dentro del DOM virtual en lugar de hacerlo directamente en el DOM del navegador, React puede actuar rápidamente y solo agregar, actualizar y eliminar componentes que han cambiado desde que se produjo el último ciclo de procesamiento.

Hola mundo con funciones sin estado

Los componentes sin estado están obteniendo su filosofía de la programación funcional. Lo que implica que: Una función devuelve todo el tiempo exactamente lo mismo que se le da.

Por ejemplo:

```
const statelessSum = (a, b) => a + b;

let a = 0;
const statefulSum = () => a++;
```

Como puede ver en el ejemplo anterior, statelessSum siempre devolverá los mismos valores dados a y b. Sin embargo, la función statefulSum no devolverá los mismos valores incluso sin parámetros. Este tipo de comportamiento de la función también se llama como *efecto secundario* . Desde entonces, el componente afecta a algunas cosas más allá.

Por lo tanto, se recomienda utilizar componentes sin estado con más frecuencia, ya que no tienen *efectos secundarios* y siempre crearán el mismo comportamiento. Eso es lo que quieres buscar en tus aplicaciones porque el estado fluctuante es el peor escenario para un programa mantenible.

El tipo más básico de componente de reacción es uno sin estado. Los componentes de React que son funciones puras de sus propiedades y no requieren ninguna administración de estado interna pueden escribirse como simples funciones de JavaScript. Se dice que estos son `Stateless Functional Components` porque son una función solo de `props` , sin tener ningún `state` que seguir.

Aquí hay un ejemplo simple para ilustrar el concepto de un `Stateless Functional Component` :

```
// In HTML
<div id="element"></div>

// In React
const MyComponent = props => {
    return <h1>Hello, {props.name}!</h1>;
};

ReactDOM.render(<MyComponent name="Arun" />, element);
```

```
// Will render <h1>Hello, Arun!</h1>
```

Tenga en cuenta que todo lo que hace este componente es generar un elemento `h1` que contenga el `name` prop. Este componente no hace un seguimiento de ningún estado. Aquí hay un ejemplo de ES6 también:

```
import React from 'react'

const HelloWorld = props => (
    <h1>Hello, {props.name}!</h1>
)

HelloWorld.propTypes = {
    name: React.PropTypes.string.isRequired
}

export default HelloWorld
```

Dado que estos componentes no requieren una instancia de respaldo para administrar el estado, React tiene más espacio para las optimizaciones. La implementación está limpia, pero hasta ahora no se han implementado tales optimizaciones para componentes sin estado .

Crear React App

create-react-app es un generador repetitivo de la aplicación React creado por Facebook. Proporciona un entorno de desarrollo configurado para facilitar su uso con una configuración mínima, que incluye:

- Transpilación ES6 y JSX.
- Servidor de desarrollo con recarga de modulo caliente
- Código de línea
- Auto-prefijo CSS
- Crear script con JS, CSS y agrupación de imágenes, y sourcemaps
- Marco de pruebas de broma

Instalación

Primero, instale la aplicación create-react-global con el administrador de paquetes de nodo (npm).

```
npm install -g create-react-app
```

Luego ejecuta el generador en tu directorio elegido.

```
create-react-app my-app
```

Navegue hasta el directorio recién creado y ejecute el script de inicio.

```
cd my-app/
npm start
```

Configuración

create-react-app es intencionalmente no configurable por defecto. Si se requiere un uso no predeterminado, por ejemplo, para usar un lenguaje CSS compilado como Sass, entonces se puede usar el comando eject.

```
npm run eject
```

Esto permite la edición de todos los archivos de configuración. NB este es un proceso irreversible.

Alternativas

Las placas de calderas React alternativas incluyen:

- enclave
- nwb
- movimiento
- rackt-cli
- budō
- rwb
- quik
- Sagui
- roc

Construir React App

Para construir su aplicación para producción lista, ejecute el siguiente comando

```
npm run build
```

Fundamentos absolutos de la creación de componentes reutilizables

Componentes y accesorios

Como React se ocupa solo de la vista de una aplicación, la mayor parte del desarrollo en React será la creación de componentes. Un componente representa una parte de la vista de su aplicación. Los "apoyos" son simplemente los atributos utilizados en un nodo JSX (por ejemplo, `<SomeComponent someProp="some prop's value" />`), y son la principal forma en que nuestra aplicación interactúa con nuestros componentes. En el fragmento anterior, dentro de SomeComponent, tendríamos acceso a `this.props` , cuyo valor sería el objeto `{someProp: "some prop's value"}` .

Puede ser útil pensar en los componentes de React como funciones simples: toman información en forma de "accesorios" y producen resultados como marcado. Muchos componentes simples llevan esto un paso más allá, convirtiéndose en "Funciones puras", lo que significa que no emiten

efectos secundarios y son idempotentes (dado un conjunto de entradas, el componente siempre producirá la misma salida). Este objetivo se puede cumplir formalmente creando componentes como funciones, en lugar de "clases". Hay tres formas de crear un componente React:

- **Componentes funcionales ("sin estado")**

```
const FirstComponent = props => (
    <div>{props.content}</div>
);
```

- **React.createClass ()**

```
const SecondComponent = React.createClass({
    render: function () {
        return (
            <div>{this.props.content}</div>
        );
    }
});
```

- **Clases de ES2015**

```
class ThirdComponent extends React.Component {
    render() {
        return (
            <div>{this.props.content}</div>
        );
    }
}
```

Estos componentes se utilizan exactamente de la misma manera:

```
const ParentComponent = function (props) {
    const someText = "FooBar";
    return (
        <FirstComponent content={someText} />
        <SecondComponent content={someText} />
        <ThirdComponent content={someText} />
    );
}
```

Los ejemplos anteriores producirán un marcado idéntico.

Los componentes funcionales no pueden tener "estado" dentro de ellos. Entonces, si su componente necesita tener un estado, entonces elija componentes basados en clases. Consulte Crear componentes para obtener más información.

Como nota final, los accesorios de reacción son inmutables una vez que se han pasado, lo que significa que no pueden modificarse desde un componente. Si el padre de un componente cambia el valor de un prop, React se encarga de reemplazar los antiguos props por el nuevo, el componente se redirigirá utilizando los nuevos valores.

Vea Thinking In React y Reutilizable Components para profundizar en la relación de los accesorios con los componentes.

Lea Empezando con React en línea: https://riptutorial.com/es/reactjs/topic/797/empezando-con-react

Capítulo 2: Actuación

Examples

Lo básico - HTML DOM vs Virtual DOM

HTML DOM es caro

Cada página web se representa internamente como un árbol de objetos. Esta representación se llama *modelo de objeto de documento* . Además, es una interfaz de lenguaje neutral que permite a los lenguajes de programación (como JavaScript) acceder a los elementos HTML.

En otras palabras

> El DOM de HTML es un estándar sobre cómo obtener, cambiar, agregar o eliminar elementos HTML.

Sin embargo, esas **operaciones de DOM** son extremadamente **caras** .

Virtual DOM es una solución

Entonces, el equipo de React tuvo la idea de abstraer el *DOM de HTML* y crear su propio *DOM virtual* para calcular el número mínimo de operaciones que necesitamos aplicar en el *DOM de HTML* para replicar el estado actual de nuestra aplicación.

El DOM virtual ahorra tiempo de modificaciones innecesarias del DOM.

¿Cómo exactamente?

En cada momento, React tiene el estado de la aplicación representado como un `Virtual DOM` . Cada vez que cambia el estado de la aplicación, estos son los pasos que React realiza para optimizar el rendimiento.

1. Genere un nuevo *DOM virtual* que represente el nuevo estado de nuestra aplicación
2. Compare el antiguo DOM virtual (que representa el actual DOM de HTML) versus el nuevo DOM virtual
3. Basado en 2. encuentre el número mínimo de operaciones para transformar el DOM virtual antiguo (que representa el DOM HTML actual) en el DOM virtual nuevo.

- para aprender más sobre eso - lee el algoritmo de diferencia de React

4. Una vez que se encuentran esas operaciones, se asignan a sus operaciones equivalentes de *HTML DOM*

- recuerde, el *DOM virtual* es solo una abstracción del *DOM de HTML* y existe una relación isomórfica entre ellos

5. Ahora, el número mínimo de operaciones que se han encontrado y transferido a sus operaciones equivalentes de *HTML DOM* ahora se aplican directamente al *HTML DOM de* la aplicación, lo que ahorra tiempo de modificar el *DOM HTML* innecesariamente.

Nota: Las operaciones aplicadas en el DOM virtual son baratas, porque el DOM virtual es un objeto de JavaScript.

Algoritmo de diferencia de React

La generación del número mínimo de operaciones para transformar un árbol en otro tiene una complejidad en el orden de O (n ^ 3) donde n es el número de nodos en el árbol. React se basa en dos suposiciones para resolver este problema en un tiempo lineal: O (n)

1. Dos componentes de la misma clase generarán árboles similares y dos componentes de clases diferentes generarán árboles diferentes.

2. Es posible proporcionar una clave única para elementos que sea estable en diferentes renders.

Para decidir si dos nodos son diferentes, React diferencia 3 casos

1. Dos nodos son diferentes, si tienen tipos diferentes.

- por ejemplo, `<div>...</div>` es diferente de `<span>...</span>`

2. Cada vez que dos nodos tienen claves diferentes

- por ejemplo, `<div key="1">...</div>` es diferente de `<div key="2">...</div>`

Además, **lo que sigue es crucial y extremadamente importante para entender** si desea optimizar el rendimiento.

> Si los [dos nodos] no son del mismo tipo, React ni siquiera intentará hacer coincidir lo que representan. Solo eliminará el primero del DOM e insertará el segundo.

Este es el por qué

> Es muy poco probable que un elemento genere un DOM que se verá como lo que generaría. En lugar de gastar tiempo tratando de emparejar esas dos estructuras, React simplemente reconstruye el árbol desde cero.

consejos y trucos

Cuando dos nodos no son del mismo tipo, React no trata de hacerlos coincidir, simplemente elimina el primer nodo del DOM e inserta el segundo. Por eso dice el primer consejo.

1. Si te ves alternando entre dos clases de componentes con resultados muy similares, es

posible que desees que sea la misma clase.

---

2. Use shouldComponentUpdate para evitar que el componente se vuelva a enviar, si sabe que no va a cambiar, por ejemplo

```
shouldComponentUpdate: function(nextProps, nextState) {
  return nextProps.id !== this.props.id;
}
```

Medición de rendimiento con ReactJS

No puedes mejorar algo que no puedes medir . Para mejorar el rendimiento de los componentes React, debe poder medirlo. ReactJS proporciona herramientas para medir el rendimiento *de montaje anexo.* Importe el módulo `react-addons-perf` para medir el rendimiento

```
import Perf from 'react-addons-perf' // ES6
var Perf = require('react-addons-perf') // ES5 with npm
var Perf = React.addons.Perf; // ES5 with react-with-addons.js
```

Puedes usar los siguientes métodos del módulo `Perf` importado:

- Perf.printInclusive ()
- Perf.printExclusive ()
- Perf.printWasted ()
- Perf.printOperations ()
- Perf.printDOM ()

El más importante que necesitará la mayoría del tiempo es `Perf.printWasted()` que le brinda la representación tabular del tiempo perdido de su componente individual.

(index)	Owner > component	Waste
0	"Todos > TodoItem"	102.7

Total time: 132.71 ms

Puede anotar la columna **Tiempo** perdido en la tabla y mejorar el rendimiento del Componente utilizando la sección de **Consejos y trucos que** se encuentra arriba

Consulte la Guía oficial React y el excelente artículo de Benchling Engg. en React Performance

Lea Actuación en línea: https://riptutorial.com/es/reactjs/topic/6875/actuacion

Capítulo 3: Apoyos en reaccionar

Observaciones

NOTA: A partir de React 15.5 y superior, el componente PropTypes vive en su propio paquete npm, es decir, 'prop-types' y necesita su propia declaración de importación cuando se usa PropTypes. Consulte la documentación oficial de reacción para el cambio de última hora: https://facebook.github.io/react/blog/2017/04/07/react-v15.5.0.html

Examples

Introducción

`props` se utilizan para pasar datos y métodos de un componente principal a un componente secundario.

Cosas interesantes sobre `props`

1. Son inmutables.
2. Nos permiten crear componentes reutilizables.

Ejemplo basico

```
class Parent extends React.Component{
  doSomething(){
     console.log("Parent component");
  }
  render() {
    return <div>
         <Child
           text="This is the child number 1"
           title="Title 1"
           onClick={this.doSomething} />
         <Child
           text="This is the child number 2"
           title="Title 2"
           onClick={this.doSomething} />
      </div>
  }
}

class Child extends React.Component{
  render() {
    return <div>
       <h1>{this.props.title}</h1>
       <h2>{this.props.text}</h2>
      </div>
  }
}
```

Como puede ver en el ejemplo, gracias a los `props` podemos crear componentes reutilizables.

Accesorios por defecto

`defaultProps` permite establecer valores predeterminados, o de respaldo, para los `props` sus componentes. `defaultProps` son útiles cuando llama a componentes desde diferentes vistas con apoyos fijos, pero en algunas vistas debe pasar un valor diferente.

Sintaxis

ES5

```
var MyClass = React.createClass({
  getDefaultProps: function() {
    return {
      randomObject: {},
      ...
    };
  }
}
```

ES6

```
class MyClass extends React.Component {...}

MyClass.defaultProps = {
    randomObject: {},
    ...
}
```

ES7

```
class MyClass extends React.Component {
    static defaultProps = {
        randomObject: {},
        ...
    };
}
```

El resultado de `getDefaultProps()` o `defaultProps` se almacenará en caché y se usará para garantizar que `this.props.randomObject` tendrá un valor si no fue especificado por el componente principal.

PropTypes

`propTypes` permite especificar qué `props` necesita su componente y el tipo que deben ser. Su componente funcionará sin establecer `propTypes` , pero es una buena práctica definirlos, ya que hará que su componente sea más legible, actuar como documentación para otros desarrolladores que están leyendo su componente, y durante el desarrollo, React le avisará si intenta establezca

un prop, que es un tipo diferente a la definición que ha establecido para él.

Algunos primitivos `propTypes` y comúnmente utilizables `propTypes` son -

```
  optionalArray: React.PropTypes.array,
  optionalBool: React.PropTypes.bool,
  optionalFunc: React.PropTypes.func,
  optionalNumber: React.PropTypes.number,
  optionalObject: React.PropTypes.object,
  optionalString: React.PropTypes.string,
  optionalSymbol: React.PropTypes.symbol
```

Si adjunta `isRequired` a cualquier `propType` entonces ese prop debe suministrarse al crear la instancia de ese componente. Si no proporciona los `propTypes` **necesarios,** no se puede crear la instancia del componente.

Sintaxis

ES5

```
var MyClass = React.createClass({
  propTypes: {
    randomObject: React.PropTypes.object,
    callback: React.PropTypes.func.isRequired,
    ...
  }
}
```

ES6

```
class MyClass extends React.Component {...}

MyClass.propTypes = {
    randomObject: React.PropTypes.object,
    callback: React.PropTypes.func.isRequired,
    ...
};
```

ES7

```
class MyClass extends React.Component {
     static propTypes = {
        randomObject: React.PropTypes.object,
        callback: React.PropTypes.func.isRequired,
        ...
    };
}
```

Validación de utilería más compleja.

De la misma manera, `PropTypes` permite especificar una validación más compleja

Validando un objeto

```
...
    randomObject: React.PropTypes.shape({
        id: React.PropTypes.number.isRequired,
        text: React.PropTypes.string,
    }).isRequired,
...
```

Validando en matriz de objetos

```
...
    arrayOfObjects: React.PropTypes.arrayOf(React.PropTypes.shape({
        id: React.PropTypes.number.isRequired,
        text: React.PropTypes.string,
    })).isRequired,
...
```

Pasando los puntales utilizando el operador extendido

En lugar de

```
var component = <Component foo={this.props.x} bar={this.props.y} />;
```

Cuando sea necesario pasar cada propiedad como un único valor prop, podría usar el operador de propagación . . . compatible con las matrices en ES6 para transmitir todos sus valores. El componente ahora se verá así.

```
var component = <Component {...props} />;
```

Recuerde que las propiedades del objeto que usted pasa se copian en los accesorios del componente.

El orden es importante. Los atributos posteriores anulan los anteriores.

```
var props = { foo: 'default' };
var component = <Component {...props} foo={'override'} />;
console.log(component.props.foo); // 'override'
```

Otro caso es que también puede usar el operador de propagación para pasar solo partes de accesorios a componentes secundarios, luego puede usar la sintaxis de desestructuración de los accesorios de nuevo.

Es muy útil cuando los componentes de los niños necesitan muchos accesorios pero no quieren pasarlos uno por uno.

```
const { foo, bar, other } = this.props // { foo: 'foo', bar: 'bar', other: 'other' };
var component = <Component {...{foo, bar}} />;
```

```
const { foo, bar } = component.props
console.log(foo, bar); // 'foo bar'
```

Props.children y composición de componentes

Los componentes "secundarios" de un componente están disponibles en una utilería especial, `props.children` . Este elemento es muy útil para "componer" componentes juntos y puede hacer que el marcado JSX sea más intuitivo o reflexivo de la estructura final prevista del DOM:

```
var SomeComponent = function () {
    return (
        <article className="textBox">
            <header>{this.props.heading}</header>
            <div className="paragraphs">
                {this.props.children}
            </div>
        </article>
    );
}
```

Lo que nos permite incluir un número arbitrario de subelementos al usar el componente más adelante:

```
var ParentComponent = function () {
    return (
        <SomeComponent heading="Amazing Article Box" >
            <p className="first"> Lots of content </p>
            <p> Or not </p>
        </SomeComponent>
    );
}
```

Props.children también puede ser manipulado por el componente. Debido a que props.children puede o no ser una matriz , React proporciona funciones de utilidad para ellos como React.Children . Considere en el ejemplo anterior si hubiéramos querido incluir cada párrafo en su propio elemento `<section>` :

```
var SomeComponent = function () {
    return (
        <article className="textBox">
            <header>{this.props.heading}</header>
            <div className="paragraphs">
                {React.Children.map(this.props.children, function (child) {
                    return (
                        <section className={child.props.className}>
                            React.cloneElement(child)
                        </section>
                    );
                })}
            </div>
        </article>
    );
}
```

Tenga en cuenta el uso de React.cloneElement para eliminar los accesorios de la etiqueta `<p>` secundaria: como los objetos son inmutables, estos valores no se pueden cambiar directamente. En su lugar, debe utilizarse un clon sin estos accesorios.

Además, cuando agregue elementos en bucles, tenga en cuenta cómo React reconcilia a los niños durante un cambio de rumbo , y considere la posibilidad de incluir un elemento `key` único global en los elementos secundarios agregados en un bucle.

Detección del tipo de componentes para niños.

A veces es realmente útil conocer el tipo de componente hijo cuando se itera a través de ellos. Para recorrer los componentes de los niños, puede usar la función React `Children.map` util:

```
React.Children.map(this.props.children, (child) => {
  if (child.type === MyComponentType) {
    ...
  }
});
```

El objeto secundario expone la propiedad de `type` que puede comparar con un componente específico.

Lea Apoyos en reaccionar en línea: https://riptutorial.com/es/reactjs/topic/2749/apoyos-en-reaccionar

Capítulo 4: Cómo configurar un webpack básico, reaccionar y babel.

Observaciones

Este canal de compilación no es exactamente lo que llamaría "listo para la producción", pero le brinda un buen comienzo para que agregue las cosas que necesita para obtener la experiencia de desarrollo que está buscando. El enfoque que algunas personas adoptan (incluyéndome a mí mismo) es tomar una tubería completamente construida de Yeoman.io o en otro lugar y luego quitarle las cosas que no quieren hasta que se adapte al estilo. No hay nada de malo en esto, pero tal vez con el ejemplo anterior podría optar por el enfoque opuesto y acumularse a partir de los huesos.

Algunas cosas que le gustaría agregar son cosas como un marco de prueba y estadísticas de cobertura como Karma con Mocha o Jasmine. Lintado con ESLint. Reemplazo de módulos en caliente en webpack-dev-server para que pueda obtener esa experiencia de desarrollo de Ctrl + S, F5. Además, la tubería actual solo se compila en modo dev, por lo que una tarea de compilación de producción sería buena.

Gotchas!

Observe que en la propiedad de contexto de `webpack.config.js` hemos utilizado el módulo de ruta de nodo para definir nuestra ruta en lugar de simplemente concatenar `__dirname` a la cadena `/src` esto se debe a que Windows odia las barras diagonales . Entonces, para hacer que la solución sea más compatible con varias plataformas, use el nodo apalancamiento para ayudarnos.

Explicación de las propiedades webpack.config.js

contexto

Esta es la ruta de archivo que el paquete web utilizará como ruta raíz para resolver rutas de archivos relativas. Entonces, en index.jsx donde usamos `require('./index.html')` ese punto realmente se resuelve en el directorio `src/` porque lo hemos definido como tal en esta propiedad.

entrada

Donde webpack busca primero para comenzar a agrupar la solución. Esta es la razón por la que verá que en el index.jsx estamos combinando la solución con requisitos e importaciones.

salida

Aquí es donde definimos donde webpack debería estar eliminando los archivos que ha encontrado para agrupar. También hemos definido un nombre para el archivo en el que se eliminarán nuestros estilos y javascript empaquetados.

devServer

Estas son configuraciones específicas para webpack-dev-server. La base de `contentBase` define donde el servidor debe hacer que sea su raíz, hemos definido la carpeta `dist/` como nuestra base aquí. El `port` es el puerto en el que se alojará el servidor. `open` es lo que se usa para instruir a webpack-dev-server para que abra su navegador predeterminado una vez que haya girado el servidor.

módulo> cargadores

Esto define una asignación para el paquete web para usar, de modo que se sepa qué hacer cuando encuentre archivos diferentes. La propiedad de `test` otorga expresiones regulares para el paquete web para determinar si debe aplicar este módulo, en la mayoría de los casos tenemos coincidencias en las extensiones de archivo. `loader` o los `loaders` proporcionan el nombre del módulo del cargador que nos gustaría usar para cargar el archivo en el paquete web y dejar que el cargador se encargue de la agrupación de ese tipo de archivo. También hay una propiedad de `query` en el javascript, esto solo proporciona una cadena de consulta al cargador, por lo que probablemente podríamos haber usado una propiedad de consulta en el cargador html también si quisiéramos. Es solo una forma diferente de hacer las cosas.

Examples

Cómo construir una tubería para un "Hola mundo" personalizado con imágenes.

Paso 1: Instala Node.js

La canalización de compilación que construirá se basa en Node.js, por lo que debe asegurarse en primer lugar de que tenga esto instalado. Para obtener instrucciones sobre cómo instalar Node.js, puede consultar los documentos SO para eso aquí

Paso 2: Inicializa tu proyecto como un módulo de nodo

Abra su carpeta de proyectos en la línea de comandos y use el siguiente comando:

```
npm init
```

Para los propósitos de este ejemplo, puede sentirse libre de tomar los valores predeterminados o si desea obtener más información sobre lo que significa todo esto, puede consultar este documento SO sobre la configuración de paquetes.

Paso 3: Instalar los paquetes npm necesarios

Ejecute el siguiente comando en la línea de comandos para instalar los paquetes necesarios para este ejemplo:

```
npm install --save react react-dom
```

Luego, para las dependencias de desarrollo, ejecute este comando:

```
npm install --save-dev babel-core babel-preset-react babel-preset-es2015 webpack babel-loader
css-loader style-loader file-loader image-webpack-loader
```

Finalmente, webpack y webpack-dev-server son cosas que vale la pena instalar globalmente en lugar de como una dependencia de su proyecto. Si prefiere agregarlo como una dependencia, eso funcionará, no lo hago. Aquí está el comando para ejecutar:

```
npm install --global webpack webpack-dev-server
```

Paso 3: Agregue un archivo .babelrc a la raíz de su proyecto

Esto configurará babel para usar los ajustes preestablecidos que acabas de instalar. Su archivo .babelrc debería verse así:

```
{
  "presets": ["react", "es2015"]
}
```

Paso 4: Configurar la estructura de directorios del proyecto

Establézcase una estructura de directorio que se vea como la siguiente en la raíz de su directorio:

```
|- node_modules
|- src/
  |- components/
  |- images/
  |- styles/
  |- index.html
  |- index.jsx
|- .babelrc
|- package.json
```

NOTA: Los `node_modules` , `.babelrc` y `package.json` deberían haber estado allí desde los pasos anteriores, simplemente los `.babelrc` para que pueda ver dónde encajan.

Paso 5: rellenar el proyecto con los archivos del proyecto Hello World

Esto no es realmente importante para el proceso de construcción de una tubería, así que solo le daré el código para que pueda copiarlos y pegarlos en:

src / components / HelloWorldComponent.jsx

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

class HelloWorldComponent extends Component {
  constructor(props) {
    super(props);
    this.state = {name: 'Student'};
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(e) {
    this.setState({name: e.target.value});
```

```
  }

  render() {
    return (
      <div>
        <div className="image-container">
          <img src="./images/myImage.gif" />
        </div>
        <div className="form">
          <input type="text" onChange={this.handleChange} />
          <div>
            My name is {this.state.name} and I'm a clever cloggs because I built a React build
pipeline
          </div>
        </div>
      </div>
    );
  }
}

export default HelloWorldComponent;
```

src / images / myImage.gif

Siéntase libre de sustituir esto con cualquier imagen que le gustaría, simplemente está ahí para demostrar el punto en el que podemos agrupar las imágenes también. Si proporciona su propia imagen y le da un nombre diferente, tendrá que actualizar `HelloWorldComponent.jsx` para reflejar sus cambios. Del mismo modo, si elige una imagen con una extensión de archivo diferente, deberá modificar la propiedad de `test` del cargador de imágenes en webpack.config.js con la expresión regular correspondiente para que coincida con su nueva extensión de archivo.

src / styles / styles.css

```
.form {
  margin: 25px;
  padding: 25px;
  border: 1px solid #ddd;
  background-color: #eaeaea;
  border-radius: 10px;
}

.form div {
  padding-top: 25px;
}

.image-container {
  display: flex;
  justify-content: center;
}
```

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
```

```
  <title>Learning to build a react pipeline</title>
</head>
<body>
  <div id="content"></div>
  <script src="app.js"></script>
</body>
</html>
```

index.jsx

```
import React from 'react';
import { render } from 'react-dom';
import HelloWorldComponent from './components/HelloWorldComponent.jsx';

require('./images/myImage.gif');
require('./styles/styles.css');
require('./index.html');

render(<HelloWorldComponent />, document.getElementById('content'));
```

Paso 6: Crear la configuración del webpack

Cree un archivo llamado webpack.config.js en la raíz de su proyecto y copie este código en él:

webpack.config.js

```
var path = require('path');

var config = {
  context: path.resolve(__dirname + '/src'),
  entry: './index.jsx',
  output: {
    filename: 'app.js',
    path: path.resolve(__dirname + '/dist'),
  },
  devServer: {
    contentBase: path.join(__dirname + '/dist'),
    port: 3000,
    open: true,
  },
  module: {
    loaders: [
      {
        test: /\.(js|jsx)$/,
        exclude: /node_modules/,
        loader: 'babel-loader'
      },
      {
        test: /\.css$/,
        loader: "style!css"
      },
      {
        test: /\.gif$/,
        loaders: [
          'file?name=[path][name].[ext]',
          'image-webpack',
        ]
      },
```

```
      { test: /\.(html)$/,
        loader: "file?name=[path][name].[ext]"
      }
    ],
  },
};

module.exports = config;
```

Paso 7: Crea tareas npm para tu pipeline

Para hacer esto, deberá agregar dos propiedades a la clave de scripts de JSON definida en el archivo package.json en la raíz de su proyecto. Haz que la clave de tus scripts se vea así:

```
  "scripts": {
    "start": "webpack-dev-server",
    "build": "webpack",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
```

El guión de `test` ya habrá estado allí y puede elegir si desea mantenerlo o no, no es importante para este ejemplo.

Paso 8: utilizar la tubería

Desde la línea de comandos, si se encuentra en el directorio raíz del proyecto, ahora debería poder ejecutar el comando:

```
npm run build
```

Esto agrupará la pequeña aplicación que ha creado y la colocará en el directorio `dist/` que creará en la raíz de la carpeta de su proyecto.

Si ejecuta el comando:

```
npm start
```

Luego, la aplicación que haya creado se servirá en su navegador web predeterminado dentro de una instancia de servidor webpack dev.

Lea Cómo configurar un webpack básico, reaccionar y babel. en línea: https://riptutorial.com/es/reactjs/topic/6294/como-configurar-un-webpack-basico--reaccionar-y-babel-

Capítulo 5: Cómo y por qué usar llaves en React.

Introducción

Siempre que esté mostrando una lista de componentes React, cada componente debe tener un atributo `key` . La clave puede ser cualquier valor, pero debe ser única para esa lista.

Cuando React tiene que representar los cambios en una lista de elementos, React simplemente itera en ambas listas de niños al mismo tiempo y genera una mutación siempre que haya una diferencia. Si no hay claves configuradas para los niños, React escanea cada niño. De lo contrario, React compara las claves para saber cuáles se agregaron o eliminaron de la lista

Observaciones

Para obtener más información, visite este enlace para leer cómo usar las claves: https://facebook.github.io/react/docs/lists-and-keys.html

Y visite este enlace para leer por qué se recomienda usar las claves: https://facebook.github.io/react/docs/reconciliation.html#recursing-on-children

Examples

Ejemplo básico

Para un componente React sin clase:

```
function SomeComponent(props){

    const ITEMS = ['cat', 'dog', 'rat']
    function getItemsList(){
        return ITEMS.map(item => <li key={item}>{item}</i>);
    }

    return (
        <ul>
            {getItemsList()}
        </ul>
    );
}
```

Para este ejemplo, el componente anterior resuelve:

```
<ul>
    <li key='cat'>cat</li>
    <li key='dog'>dog</li>
    <li key='rat'>rat</li>
```

```
<ul>
```

Lea Cómo y por qué usar llaves en React. en línea:
https://riptutorial.com/es/reactjs/topic/9665/como-y-por-que-usar-llaves-en-react-

Capítulo 6: Componentes

Observaciones

`React.createClass` quedó en desuso en v15.5 y se espera que se elimine en v16 . Hay un paquete de reemplazo para aquellos que todavía lo requieren. Los ejemplos que lo usan deben ser actualizados.

Examples

Componente basico

Dado el siguiente archivo HTML:

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Tutorial</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
core/5.8.34/browser.min.js"></script>
  </head>
  <body>
    <div id="content"></div>
    <script type="text/babel" src="scripts/example.js"></script>
  </body>
</html>
```

Puede crear un componente básico utilizando el siguiente código en un archivo separado:

scripts / example.js

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';

class FirstComponent extends Component {
  render() {
    return (
      <div className="firstComponent">
        Hello, world! I am a FirstComponent.
      </div>
    );
  }
}
ReactDOM.render(
  <FirstComponent />, // Note that this is the same as the variable you stored above
  document.getElementById('content')
);
```

Obtendrá el siguiente resultado (tenga en cuenta lo que está dentro del `div#content`):

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Tutorial</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
core/5.8.34/browser.min.js"></script>
  </head>
  <body>
    <div id="content">
        <div className="firstComponent">
            Hello, world! I am a FirstComponent.
        </div>
    </div>
    <script type="text/babel" src="scripts/example.js"></script>
  </body>
</html>
```

Componentes de anidación

Gran parte del poder de ReactJS es su capacidad para permitir el anidamiento de componentes. Tome los siguientes dos componentes:

```
var React = require('react');
var createReactClass = require('create-react-class');

var CommentList = reactCreateClass({
  render: function() {
    return (
      <div className="commentList">
        Hello, world! I am a CommentList.
      </div>
    );
  }
});

var CommentForm = reactCreateClass({
  render: function() {
    return (
      <div className="commentForm">
        Hello, world! I am a CommentForm.
      </div>
    );
  }
});
```

Puede anidar y referirse a esos componentes en la definición de un componente diferente:

```
var React = require('react');
var createReactClass = require('create-react-class');

var CommentBox = reactCreateClass({
  render: function() {
```

```
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList /> // Which was defined above and can be reused
        <CommentForm /> // Same here
      </div>
    );
  }
});
```

La anidación adicional se puede hacer de tres maneras, todas ellas tienen sus propios lugares para ser utilizados.

1. Anidar sin usar niños.

(continuado desde arriba)

```
var CommentList = reactCreateClass({
  render: function() {
    return (
      <div className="commentList">
        <ListTitle/>
        Hello, world! I am a CommentList.
      </div>
    );
  }
});
```

Este es el estilo donde A compone B y B compone C.

Pros

- Fácil y rápido de separar elementos de la interfaz de usuario
- Fácil de inyectar accesorios a los niños según el estado del componente principal

Contras

- Menos visibilidad en la arquitectura de la composición.
- Menos reusabilidad

Bien si

- B y C son solo componentes de presentación
- B debe ser responsable del ciclo de vida de C

2. Anidar usando niños.

(continuado desde arriba)

```
var CommentBox = reactCreateClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList>
            <ListTitle/> // child
        </CommentList>
        <CommentForm />
      </div>
    );
  }
});
```

Este es el estilo en el que A compone B y A le dice a B que componga C. Más poder para los componentes principales.

Pros

- Mejor gestión del ciclo de vida de los componentes.
- Mejor visibilidad en la arquitectura de la composición.
- Mejor reusuabilidad

Contras

- Inyectar accesorios puede ser un poco caro.
- Menos flexibilidad y potencia en componentes infantiles

Bien si

- B debería aceptar componer algo diferente de C en el futuro o en otro lugar
- A debe controlar el ciclo de vida de C

B representaría a C usando `this.props.children` , y no hay una forma estructurada para que B sepa para qué sirven esos niños. Entonces, B puede enriquecer los componentes secundarios al dar apoyo adicional, pero si B necesita saber exactamente qué son, el # 3 podría ser una mejor opción.

3. Anidar utilizando puntales.

(continuado desde arriba)

```
var CommentBox = reactCreateClass({
  render: function() {
    return (
      <div className="commentBox">
```

```
        <h1>Comments</h1>
        <CommentList title={ListTitle}/> //prop
        <CommentForm />
      </div>
    );
  }
});
```

Este es el estilo donde A compone B y B proporciona una opción para que A pase algo para componer para un propósito específico. Composición más estructurada.

Pros

- La composición como característica
- Validación fácil
- Mejor composibilidad

Contras

- Inyectar accesorios puede ser un poco caro.
- Menos flexibilidad y potencia en componentes infantiles

Bien si

- B tiene características específicas definidas para componer algo.
- B solo debe saber cómo renderizar no qué renderizar

3 es generalmente una necesidad para hacer una biblioteca pública de componentes, pero también es una buena práctica en general para hacer componentes compositivos y definir claramente las características de la composición. El # 1 es el más fácil y rápido de hacer algo que funciona, pero el # 2 y el # 3 deberían proporcionar ciertos beneficios en varios casos de uso.

Creando Componentes

Esta es una extensión del Ejemplo Básico:

Estructura basica

```
import React, { Component } from 'react';
import { render } from 'react-dom';

class FirstComponent extends Component {
    render() {
        return (
            <div>
                Hello, {this.props.name}! I am a FirstComponent.
            </div>
        );
```

```
    }
}

render(
    <FirstComponent name={ 'User' } />,
    document.getElementById('content')
);
```

El ejemplo anterior se llama un componente **sin estado** ya que no contiene el estado (en el sentido de Reacción de la palabra).

En tal caso, a algunas personas les resulta preferible utilizar componentes funcionales sin estado, que se basan en las funciones de flecha de ES6 .

Componentes funcionales sin estado

En muchas aplicaciones, hay componentes inteligentes que mantienen el estado pero generan componentes simples que simplemente reciben accesorios y devuelven HTML como JSX. Los componentes funcionales sin estado son mucho más reutilizables y tienen un impacto positivo en el rendimiento de su aplicación.

Tienen 2 características principales:

1. Cuando se procesan, reciben un objeto con todos los accesorios que se transmitieron
2. Deben devolver el JSX para ser renderizados.

```
// When using JSX inside a module you must import React
import React from 'react';
import PropTypes from 'prop-types';

const FirstComponent = props => (
    <div>
        Hello, {props.name}! I am a FirstComponent.
    </div>
);

//arrow components also may have props validation
FirstComponent.propTypes = {
    name: PropTypes.string.isRequired,
}

// To use FirstComponent in another file it must be exposed through an export call:
export default FirstComponent;
```

Componentes de estado

A diferencia de los componentes 'sin estado' que se muestran arriba, los componentes 'con estado' tienen un objeto de estado que puede actualizarse con el método `setState` . El estado debe inicializarse en el `constructor` antes de que se pueda configurar:

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

class SecondComponent extends Component {
    constructor(props) {
        super(props);

        this.state = {
            toggle: true
        };

        // This is to bind context when passing onClick as a callback
        this.onClick = this.onClick.bind(this);
    }

    onClick() {
        this.setState((prevState, props) => ({
            toggle: !prevState.toggle
        }));
    }

    render() {
        return (
            <div onClick={this.onClick}>
                Hello, {this.props.name}! I am a SecondComponent.
                <br />
                Toggle is: {this.state.toggle}
            </div>
        );
    }
}
```

La extensión de un componente con `PureComponent` lugar de `Component` implementará automáticamente el `shouldComponentUpdate()` ciclo de vida con una comparación superficial de propiedades y estado. Esto mantiene su aplicación más eficaz al reducir la cantidad de renders innecesarios que ocurren. Esto supone que sus componentes son 'Puros' y siempre generan la misma salida con el mismo estado y entrada de propiedades.

Componentes de orden superior

Los componentes de orden superior (HOC) permiten compartir la funcionalidad de los componentes.

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

const PrintHello = ComposedComponent => class extends Component {
    onClick() {
        console.log('hello');
    }

    /* The higher order component takes another component as a parameter
    and then renders it with additional props */
    render() {
        return <ComposedComponent {...this.props } onClick={this.onClick} />
    }
}
```

```
const FirstComponent = props => (
    <div onClick={ props.onClick }>
        Hello, {props.name}! I am a FirstComponent.
    </div>
);

const ExtendedComponent = PrintHello(FirstComponent);
```

Los componentes de orden superior se utilizan cuando desea compartir la lógica entre varios componentes, independientemente de qué tan diferentes se representen.

trampas setState

Debe tener cuidado al usar `setState` en un contexto asíncrono. Por ejemplo, puede intentar llamar a `setState` en la devolución de llamada de una solicitud de obtención:

```
class MyClass extends React.Component {
    constructor() {
        super();

        this.state = {
            user: {}
        };
    }

    componentDidMount() {
        this.fetchUser();
    }

    fetchUser() {
        $.get('/api/users/self')
            .then((user) => {
                this.setState({user: user});
            });
    }

    render() {
        return <h1>{this.state.user}</h1>;
    }
}
```

Esto podría provocar problemas: si se llama a la devolución de llamada después de que se desmonta el `Component` , entonces este `this.setState` no será una función. Siempre que este sea el caso, debe tener cuidado de asegurarse de que su uso de `setState` sea `setState` .

En este ejemplo, es posible que desee cancelar la solicitud XHR cuando el componente se desmonta:

```
class MyClass extends React.Component {
    constructor() {
        super();

        this.state = {
            user: {},
            xhr: null
```

```
        };
    }

    componentWillUnmount() {
        let xhr = this.state.xhr;

        // Cancel the xhr request, so the callback is never called
        if (xhr && xhr.readyState != 4) {
            xhr.abort();
        }
    }

    componentDidMount() {
        this.fetchUser();
    }

    fetchUser() {
        let xhr = $.get('/api/users/self')
            .then((user) => {
                this.setState({user: user});
            });

        this.setState({xhr: xhr});
    }
}
```

El método asíncrono se guarda como un estado. En el `componentWillUnmount` , realiza toda la limpieza, incluida la cancelación de la solicitud XHR.

También podrías hacer algo más complejo. En este ejemplo, estoy creando una función 'stateSetter' que acepta el objeto de este como un argumento y evita este `this.setState` cuando se ha llamado a la función `cancel` :

```
function stateSetter(context) {
    var cancelled = false;
    return {
        cancel: function () {
            cancelled = true;
        },
        setState(newState) {
            if (!cancelled) {
                context.setState(newState);
            }
        }
    }
}

class Component extends React.Component {
    constructor(props) {
        super(props);
        this.setter = stateSetter(this);
        this.state = {
            user: 'loading'
        };
    }
    componentWillUnmount() {
        this.setter.cancel();
    }
    componentDidMount() {
```

```
        this.fetchUser();
    }
    fetchUser() {
        $.get('/api/users/self')
            .then((user) => {
                this.setter.setState({user: user});
            });
    }
    render() {
        return <h1>{this.state.user}</h1>
    }
}
```

Esto funciona porque la variable `cancelled` es visible en el cierre de `setState` que creamos.

Accesorios

Los apoyos son una forma de pasar información a un componente React, pueden tener cualquier tipo, incluidas funciones, a veces denominadas devoluciones de llamada.

En JSX los props se pasan con la sintaxis del atributo.

```
<MyComponent userID={123} />
```

Dentro de la definición de MyComponent userID ahora será accesible desde el objeto props

```
// The render function inside MyComponent
render() {
    return (
        <span>The user's ID is {this.props.userID}</span>
    )
}
```

Es importante definir todos los `props` , sus tipos y, cuando corresponda, su valor predeterminado:

```
// defined at the bottom of MyComponent
MyComponent.propTypes = {
    someObject: React.PropTypes.object,
    userID: React.PropTypes.number.isRequired,
    title: React.PropTypes.string
};

MyComponent.defaultProps = {
    someObject: {},
    title: 'My Default Title'
}
```

En este ejemplo, el prop `someObject` es opcional, pero se requiere el prop `userID` . Si usted no proporciona `userID` a `MyComponent` , en tiempo de ejecución del motor Reaccionar mostrará una consola que le advierte que no se proporcionó la hélice requerida. Sin embargo, tenga cuidado, esta advertencia solo se muestra en la versión de desarrollo de la biblioteca React, la versión de producción no registrará ninguna advertencia.

Usando `defaultProps` te permite simplificar

```
const { title = 'My Default Title' } = this.props;
console.log(title);
```

a

```
console.log(this.props.title);
```

También es una garantía para el uso de la `array` de `object` y `functions` . Si no proporciona un prop predeterminado para un objeto, lo siguiente arrojará un error si no se pasa el prop:

```
if (this.props.someObject.someKey)
```

En el ejemplo anterior, `this.props.someObject` `undefined` está `undefined` y, por lo tanto, la comprobación de `someKey` generará un error y el código se interrumpirá. Con el uso de `defaultProps` puede usar de forma segura la verificación anterior.

Estados del componente - Interfaz de usuario dinámica

Supongamos que queremos tener el siguiente comportamiento: tenemos un encabezado (por ejemplo, elemento h3) y al hacer clic en él, queremos que se convierta en un cuadro de entrada para que podamos modificar el nombre del encabezado. React hace esto altamente simple e intuitivo utilizando estados de componentes y, si no, declaraciones. (Explicación del código abajo)

```
// I have used ReactBootstrap elements. But the code works with regular html elements also
var Button = ReactBootstrap.Button;
var Form = ReactBootstrap.Form;
var FormGroup = ReactBootstrap.FormGroup;
var FormControl = ReactBootstrap.FormControl;

var Comment = reactCreateClass({
  getInitialState: function() {
    return {show: false, newTitle: ''};
  },

  handleTitleSubmit: function() {
     //code to handle input box submit - for example, issue an ajax request to change name in
database
  },

  handleTitleChange: function(e) {
    //code to change the name in form input box. newTitle is initialized as empty string. We
need to update it with the string currently entered by user in the form
    this.setState({newTitle: e.target.value});
  },

  changeComponent: function() {
    // this toggles the show variable which is used  for dynamic UI
    this.setState({show: !this.state.show)};
  },

  render: function() {
```

```
    var clickableTitle;

    if(this.state.show) {
        clickableTitle = <Form inline onSubmit={this.handleTitleSubmit}>
                            <FormGroup controlId="formInlineTitle">
                                <FormControl type="text" onChange={this.handleTitleChange}>
                            </FormGroup>
                        </Form>;
    } else {
        clickabletitle = <div>
                           <Button bsStyle="link" onClick={this.changeComponent}>
                               <h3> Default Text </h3>
                           </Button>
                        </div>;
    }

    return (
        <div className="comment">
            {clickableTitle}
        </div>
    );
  }
});

ReactDOM.render(
    <Comment />, document.getElementById('content')
);
```

La parte principal del código es la variable **clickableTitle** . En función de la **demostración de** variable de estado, puede ser un elemento de formulario o un elemento de botón. React permite el anidamiento de componentes.

Entonces podemos agregar un elemento {clickableTitle} en la función de render. Busca la variable clickableTitle. Basado en el valor 'this.state.show', muestra el elemento correspondiente.

Variaciones de los componentes funcionales sin estado

```
const languages = [
  'JavaScript',
  'Python',
  'Java',
  'Elm',
  'TypeScript',
  'C#',
  'F#'
]
```

```
// one liner
const Language = ({language}) => <li>{language}</li>

Language.propTypes = {
  message: React.PropTypes.string.isRequired
}
```

```
/**
* If there are more than one line.
* Please notice that round brackets are optional here,
* However it's better to use them for readability
*/
const LanguagesList = ({languages}) => {
  <ul>
    {languages.map(language => <Language language={language} />)}
  </ul>
}

LanguagesList.PropTypes = {
  languages: React.PropTypes.array.isRequired
}
```

```
/**
 * This syntax is used if there are more work beside just JSX presentation
 * For instance some data manipulations needs to be done.
 * Please notice that round brackets after return are required,
 * Otherwise return will return nothing (undefined)
 */
const LanguageSection = ({header, languages}) => {
  // do some work
  const formattedLanguages = languages.map(language => language.toUpperCase())
  return (
    <fieldset>
      <legend>{header}</legend>
      <LanguagesList languages={formattedLanguages} />
    </fieldset>
  )
}

LanguageSection.PropTypes = {
  header: React.PropTypes.string.isRequired,
  languages: React.PropTypes.array.isRequired
}
```

```
ReactDOM.render(
  <LanguageSection
    header="Languages"
    languages={languages} />,
  document.getElementById('app')
)
```

Aquí puedes encontrar un ejemplo de ello.

Lea Componentes en línea: https://riptutorial.com/es/reactjs/topic/1185/componentes

Capítulo 7: Componentes de orden superior

Introducción

Los componentes de orden superior ("HOC" en breve) es un patrón de diseño de aplicación de reacción que se utiliza para mejorar los componentes con código reutilizable. Permiten agregar funcionalidad y comportamientos a las clases de componentes existentes.

Un HOC es una función javascript pura que acepta un componente como argumento y devuelve un componente nuevo con la funcionalidad extendida.

Observaciones

Los HOC se usan con frecuencia en bibliotecas de terceros. Como la función de conexión de Redux.

Examples

Componente de orden superior simple

Digamos que queremos console.log cada vez que el componente se monta:

hocLogger.js

```
export default function hocLogger(Component) {
  return class extends React.Component {
    componentDidMount() {
      console.log('Hey, we are mounted!');
    }
    render() {
      return <Component {...this.props} />;
    }
  }
}
```

Utilice este HOC en su código:

MyLoggedComponent.js

```
import React from "react";
import {hocLogger} from "./hocLogger";

export class MyLoggedComponent extends React.Component {
    render() {
        return (
            <div>
                This component get's logged to console on each mount.
            </div>
        );
```

```
    }
}

// Now wrap MyLoggedComponent with the hocLogger function
export default hocLogger(MyLoggedComponent);
```

Componente de orden superior que comprueba la autenticación

Digamos que tenemos un componente que solo debería mostrarse si el usuario ha iniciado sesión.

Así que creamos un HOC que verifica la autenticación en cada render ():

AuthenticatedComponent.js

```
import React from "react";

export function requireAuthentication(Component) {
    return class AuthenticatedComponent extends React.Component {

        /**
         * Check if the user is authenticated, this.props.isAuthenticated
         * has to be set from your application logic (or use react-redux to retrieve it from
global state).
         */
        isAuthenticated() {
            return this.props.isAuthenticated;
        }

        /**
         * Render
         */
        render() {
            const loginErrorMessage = (
                <div>
                    Please <a href="/login">login</a> in order to view this part of the
application.
                </div>
            );

            return (
                <div>
                    { this.isAuthenticated === true ? <Component {...this.props} /> :
loginErrorMessage }
                </div>
            );
        }
    };
}

export default requireAuthentication;
```

Luego, solo usamos este componente de orden superior en nuestros componentes que debe estar oculto a usuarios anónimos:

MyPrivateComponent.js

```
import React from "react";
import {requireAuthentication} from "./AuthenticatedComponent";

export class MyPrivateComponent extends React.Component {
    /**
     * Render
     */
    render() {
        return (
            <div>
                My secret search, that is only viewable by authenticated users.
            </div>
        );
    }
}

// Now wrap MyPrivateComponent with the requireAuthentication function
export default requireAuthentication(MyPrivateComponent);
```

Este ejemplo se describe con más detalle aquí .

Lea Componentes de orden superior en línea:
https://riptutorial.com/es/reactjs/topic/9819/componentes-de-orden-superior

Capítulo 8: Componentes funcionales sin estado

Observaciones

Los componentes funcionales sin estado en React son funciones puras de los `props` aprobados. Estos componentes no se basan en el estado y descartan el uso de los métodos del ciclo de vida de los componentes. Sin embargo, aún puede definir `propTypes` y `defaultPropts` .

Consulte https://facebook.github.io/react/docs/reusable-components.html#stateless-functions para obtener más información sobre los componentes funcionales sin estado.

Examples

Componente funcional sin estado

Los componentes le permiten dividir la interfaz de usuario en piezas *independientes* y *reutilizables* . Esta es la belleza de React; Podemos separar una página en muchos pequeños **componentes** reutilizables.

Antes de React v14, podríamos crear un componente React con estado utilizando `React.Component` (en ES6) o `React.createClass` (en ES5), independientemente de si requiere algún estado para administrar los datos o no.

React v14 introdujo una forma más sencilla de definir componentes, generalmente denominados **componentes funcionales sin estado** . Estos componentes utilizan funciones simples de JavaScript.

Por ejemplo:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

Esta función es un componente React válido porque acepta un único argumento de objeto de `props` con datos y devuelve un elemento React. Llamamos a estos componentes **funcional** porque son, literalmente, *funciones de* JavaScript.

Los componentes funcionales sin estado típicamente se enfocan en la IU; el estado debe ser administrado por componentes de "contenedor" de nivel superior, o mediante Flux / Redux, etc. Los componentes funcionales sin estado no admiten los métodos de estado o ciclo de vida.

Beneficios:

1. Sin sobrecarga de clase

2. No tiene que preocuparse por `this` palabra clave
3. Fácil de escribir y fácil de entender
4. No tienes que preocuparte por administrar valores estatales
5. Mejora del rendimiento

Resumen : si está escribiendo un componente React que no requiere estado y desea crear una interfaz de usuario reutilizable, en lugar de crear un componente React estándar, puede escribirlo como un **componente funcional sin estado** .

Tomemos un ejemplo simple:

Digamos que tenemos una página que puede registrar un usuario, buscar usuarios registrados o mostrar una lista de todos los usuarios registrados.

Este es el punto de entrada de la aplicación, `index.js` :

```
import React from 'react';
import ReactDOM from 'react-dom';

import HomePage from './homepage'

ReactDOM.render(
    <HomePage/>,
    document.getElementById('app')
);
```

El componente `HomePage` proporciona la IU para registrarse y buscar usuarios. Tenga en cuenta que es un componente típico de React que incluye el estado, la IU y el código de comportamiento. Los datos para la lista de usuarios registrados se almacenan en la variable de `state` , pero nuestra `List` reutilizable (que se muestra a continuación) encapsula el código de UI para la lista.

`homepage.js` :

```
import React from 'react'
import {Component} from 'react';

import List from './list';

export default class Temp extends Component{

    constructor(props) {
        super();
        this.state={users:[], showSearchResult: false, searchResult: []};
    }

    registerClick(){
        let users = this.state.users.slice();
        if(users.indexOf(this.refs.mail_id.value) == -1){
            users.push(this.refs.mail_id.value);
            this.refs.mail_id.value = '';
            this.setState({users});
        }else{
            alert('user already registered');
        }
    }
```

```
    searchClick(){
        let users = this.state.users;
        let index = users.indexOf(this.refs.search.value);
        if(index >= 0){
            this.setState({searchResult: users[index], showSearchResult: true});
        }else{
            alert('no user found with this mail id');
        }
    }

    hideSearchResult(){
        this.setState({showSearchResult: false});
    }

    render() {
        return (
            <div>
                <input placeholder='email-id' ref='mail_id'/>
                <input type='submit' value='Click here to register'
onClick={this.registerClick.bind(this)}/>
                <input style={{marginLeft: '100px'}} placeholder='search' ref='search'/>
                <input type='submit' value='Click here to register'
onClick={this.searchClick.bind(this)}/>
                {this.state.showSearchResult ?
                    <div>
                        Search Result:
                        <List users={[this.state.searchResult]}/>
                        <p onClick={this.hideSearchResult.bind(this)}>Close this</p>
                    </div>
                    :
                    <div>
                        Registered users:
                        <br/>
                        {this.state.users.length ?
                            <List users={this.state.users}/>
                            :
                            "no user is registered"
                        }
                    </div>
                }
            </div>
        );
    }
}
```

Finalmente, nuestra `List` **componentes funcionales sin estado** , que se utiliza muestra tanto la lista de usuarios registrados *como* los resultados de búsqueda, pero sin mantener ningún estado en sí.

`list.js` :

```
import React from 'react';
var colors = ['#6A1B9A', '#76FF03', '#4527A0'];

var List = (props) => {
    return(
        <div>
            {
```

```
                props.users.map((user, i)=>{
                    return(
                        <div key={i} style={{color: colors[i%3]}}>
                            {user}
                        </div>
                    );
                })
            }
        </div>
    );
}

export default List;
```

Referencia: https://facebook.github.io/react/docs/components-and-props.html

Lea Componentes funcionales sin estado en línea:
https://riptutorial.com/es/reactjs/topic/6588/componentes-funcionales-sin-estado

Capítulo 9: Comunicación Entre Componentes

Observaciones

Hay un total de 3 casos de comunicación entre los componentes de React:

- Caso 1: Comunicación padre a hijo.
- Caso 2: Comunicación niño a padre
- Caso 3: comunicación de componentes no relacionados (cualquier componente a cualquier componente)

Examples

Componentes de padres a hijos

Ese es el caso más fácil en realidad, muy natural en el mundo React y es probable que ya lo estés usando.

Puedes **pasar los accesorios a los componentes secundarios** . En este ejemplo, el `message` es el elemento que transmitimos al componente secundario, el mensaje de nombre se elige de forma arbitraria, puede asignarle el nombre que desee.

```
import React from 'react';

class Parent extends React.Component {
    render() {
        const variable = 5;
        return (
            <div>
                <Child message="message for child" />
                <Child message={variable} />
            </div>
        );
    }
}

class Child extends React.Component {
    render() {
        return <h1>{this.props.message}</h1>
    }
}

export default Parent;
```

Aquí, el componente `<Parent />` representa dos componentes `<Child />` , pasando un `message for child` dentro del primer componente y `5` dentro del segundo.

En resumen, usted tiene un componente (padre) que presta otro (hijo) y le pasa algunos

accesorios.

Componentes del niño al padre

Al enviar datos de vuelta al padre, para hacer esto, simplemente **pasamos una función como prop del componente principal al componente secundario** , y **el componente secundario llama a esa función** .

En este ejemplo, cambiaremos el estado principal pasando una función al componente secundario e invocando esa función dentro del componente secundario.

```
import React from 'react';

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

        this.outputEvent = this.outputEvent.bind(this);
    }
    outputEvent(event) {
        // the event context comes from the Child
        this.setState({ count: this.state.count++ });
    }

    render() {
        const variable = 5;
        return (
            <div>
                Count: { this.state.count }
                <Child clickHandler={this.outputEvent} />
            </div>
        );
    }
}

class Child extends React.Component {
    render() {
        return (
            <button onClick={this.props.clickHandler}>
                Add One More
            </button>
        );
    }
}

export default Parent;
```

Tenga en cuenta que los padres de la `outputEvent` método (que cambia el estado de Padres) es invocado por el botón del Niño `onClick` evento.

Componentes no relacionados

La única manera si sus componentes no tienen una relación padre-hijo (o están relacionados, pero están demasiado lejos, como por ejemplo, un bisnieto), es tener algún tipo de señal a la que

uno de los componentes se suscriba y el otro escriba.

Esas son las 2 operaciones básicas de cualquier sistema de eventos: **suscribirse / escuchar** un evento para ser notificado, y **enviar / desencadenar / publicar / enviar** un evento para notificar a los que quieren.

Hay al menos 3 patrones para hacer eso. Puedes encontrar una comparación aquí .

Aquí hay un breve resumen:

- Patrón 1: **Emisor de eventos / Destino / Despachador** : los oyentes deben hacer referencia a la fuente para suscribirse.
    - para suscribirse: `otherObject.addEventListener('click', () => { alert('click!'); });`
    - para enviar: `this.dispatchEvent('click');`
- Patrón 2: **publicación / suscripción** : no necesita una referencia específica a la fuente que desencadena el evento, hay un objeto global accesible en todas partes que maneja todos los eventos.
    - para suscribirse: `globalBroadcaster.subscribe('click', () => { alert('click!'); });`
    - para enviar: `globalBroadcaster.publish('click');`
- Patrón 3: **Señales** : similar a Event Emitter / Target / Dispatcher, pero aquí no usa cadenas aleatorias. Cada objeto que podría emitir eventos debe tener una propiedad específica con ese nombre. De esta manera, usted sabe exactamente qué eventos puede emitir un objeto.
    - para suscribirse: `otherObject.clicked.add( () => { alert('click'); });`
    - para enviar: `this.clicked.dispatch();`

Lea Comunicación Entre Componentes en línea:
https://riptutorial.com/es/reactjs/topic/6567/comunicacion-entre-componentes

Capítulo 10: Comunicar Entre Componentes

Examples

Comunicación entre componentes funcionales sin estado

En este ejemplo, haremos uso de los módulos `Redux` y `React Redux` para manejar el estado de nuestra aplicación y para la reproducción automática de nuestros componentes funcionales. Y, por supuesto, `React` y `React Dom`

Puedes ver la demo completa aquí

En el siguiente ejemplo tenemos tres componentes diferentes y un componente conectado

- **UserInputForm** : este componente muestra un campo de entrada Y cuando el valor del campo cambia, llama `inputChange` método `inputChange` en `props` (que es proporcionado por el componente principal) y si los datos también se proporcionan, muestra eso en el campo de entrada.
- **UserDashboard:** Este componente muestra un mensaje simple y también anida `UserInputForm` componente, sino que también pasa `inputChange` método para `UserInputForm` componente, `UserInputForm` componente inturn hace uso de este método para comunicarse con el componente de matriz.
    - **UserDashboardConnected** : este componente simplemente envuelve el componente `UserDashboard` utilizando el método de `ReactRedux connect` . Esto nos facilita la administración del estado del componente y la actualización del componente cuando el estado cambia.
- **Aplicación** : Este componente solo representa el componente `UserDashboardConnected` .

```
const UserInputForm = (props) => {

  let handleSubmit = (e) => {
    e.preventDefault();
  }

  return(
    <form action="" onSubmit={handleSubmit}>
      <label htmlFor="name">Please enter your name</label>
      <br />
      <input type="text" id="name" defaultValue={props.data.name || ''} onChange={
props.inputChange } />
    </form>
  )

}


const UserDashboard = (props) => {
```

```
  let inputChangeHandler = (event) => {
    props.updateName(event.target.value);
  }

  return(
    <div>
      <h1>Hi { props.user.name || 'User' }</h1>
      <UserInputForm data={props.user} inputChange={inputChangeHandler} />
    </div>
  )

}

const mapStateToProps = (state) => {
  return {
    user: state
  };
}
const mapDispatchToProps = (dispatch) => {
  return {
    updateName: (data) => dispatch( Action.updateName(data) ),
  };
};

const { connect, Provider } = ReactRedux;
const UserDashboardConnected = connect(
  mapStateToProps,
  mapDispatchToProps
)(UserDashboard);



const App = (props) => {
  return(
    <div>
      <h1>Communication between Stateless Functional Components</h1>
      <UserDashboardConnected />
    </div>
  )
}



const user = (state={name: 'John'}, action) => {
  switch (action.type) {
    case 'UPDATE_NAME':
      return Object.assign( {}, state, {name: action.payload}  );

    default:
      return state;
  }
};

const { createStore } = Redux;
const store = createStore(user);
const Action = {
  updateName: (data) => {
    return { type : 'UPDATE_NAME', payload: data }
  },
}
```

```
ReactDOM.render(
  <Provider store={ store }>
    <App />
  </Provider>,
  document.getElementById('application')
);
```

URL de JS Bin

Lea Comunicar Entre Componentes en línea:
https://riptutorial.com/es/reactjs/topic/6137/comunicar-entre-componentes

Capítulo 11: Configuración de React Ambiente

Examples

Componente Reactivo Simple

Queremos poder compilar el componente a continuación y mostrarlo en nuestra página web

Nombre de archivo : src / index.jsx

```
import React from 'react';
import ReactDOM from 'react-dom';

class ToDo extends React.Component {
    render() {
        return (<div>I am working</div>);
    }
}

ReactDOM.render(<ToDo />, document.getElementById('App'));
```

Instalar todas las dependencias

```
# install react and react-dom
$ npm i react react-dom --save

# install webpack for bundling
$ npm i webpack -g

# install babel for module loading, bundling and transpiling
$ npm i babel-core babel-loader --save

# install babel presets for react and es6
$ npm i babel-preset-react babel-preset-es2015 --save
```

Configurar webpack

Cree un archivo `webpack.config.js` en la raíz de su directorio de trabajo

Nombre de archivo : webpack.config.js

```
module.exports = {
    entry: __dirname + "/src/index.jsx",
    devtool: "source-map",
    output: {
        path: __dirname + "/build",
        filename: "bundle.js"
    },
```

```
    module: {
        loaders: [
            {test: /\.jsx?$/, exclude: /node_modules/, loader: "babel-loader"}
        ]
    }
}
```

Configurar babel

Cree un archivo `.babelrc` en la raíz de nuestro directorio de trabajo

Nombre de archivo : .babelrc

```
{
    "presets": ["es2015","react"]
}
```

Archivo HTML para usar el componente reaccionar

Configure un simple archivo html en la raíz del directorio del proyecto

Nombre de archivo : index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id="App"></div>
    <script src="build/bundle.js" charset="utf-8"></script>
  </body>
</html>
```

Transpilar y agrupar su componente

Usando webpack, puedes agrupar tu componente:

```
$ webpack
```

Esto creará nuestro archivo de salida en el directorio de `build` .

Abra la página HTML en un navegador para ver el componente en acción

Lea Configuración de React Ambiente en línea:
https://riptutorial.com/es/reactjs/topic/7480/configuracion-de-react-ambiente

Capítulo 12: Estado en reaccionar

Examples

Estado basico

El estado en los componentes de React es esencial para administrar y comunicar datos en su aplicación. Se representa como un objeto de JavaScript y tiene *un* alcance de *nivel de componente* , se puede considerar como los datos privados de su componente.

En el siguiente ejemplo, definimos un estado inicial en la función de `constructor` de nuestro componente y lo utilizamos en la función de `render` .

```
class ExampleComponent extends React.Component {
  constructor(props){
    super(props);

    // Set-up our initial state
    this.state = {
      greeting: 'Hiya Buddy!'
    };
  }

  render() {
    // We can access the greeting property through this.state
    return(
      <div>{this.state.greeting}</div>
    );
  }
}
```

setState ()

La principal forma de realizar actualizaciones de IU en sus aplicaciones React es a través de una llamada a la función `setState()` . Esta función realizará una *fusión superficial* entre el nuevo estado que proporcionó y el estado anterior, y activará una nueva representación de su componente y todos los difuntos.

Parámetros

1. `updater` : puede ser un objeto con una serie de pares clave-valor que deben fusionarse en el estado o una función que devuelve dicho objeto.
2. `callback (optional)` : una función que se ejecutará después de que `setState()` se haya ejecutado correctamente. Debido a que React no garantiza que las llamadas a `setState()` sean atómicas, esto a veces puede ser útil si desea realizar alguna acción después de estar seguro de que `setState()` se ha ejecutado correctamente.

Uso:

El método `setState` acepta un argumento `updater` que puede ser un objeto con una cantidad de pares clave-valor que deben fusionarse en el estado, o una función que devuelve dicho objeto computado desde `prevState` y `props` .

Usando `setState()` con un Objeto como `updater`

```
//
// An example ES6 style component, updating the state on a simple button click.
// Also demonstrates where the state can be set directly and where setState should be used.
//
class Greeting extends React.Component {
    constructor(props) {
        super(props);
        this.click = this.click.bind(this);
        // Set initial state (ONLY ALLOWED IN CONSTRUCTOR)
        this.state = {
            greeting: 'Hello!'
        };
    }
    click(e) {
        this.setState({
              greeting: 'Hello World!'
        });
    }
    render() {
        return(
            <div>
                <p>{this.state.greeting}</p>
                <button onClick={this.click}>Click me</button>
            </div>
        );
    }

}
```

Usando `setState()` con una función como `updater`

```
//
// This is most often used when you want to check or make use
// of previous state before updating any values.
//

this.setState(function(previousState, currentProps) {
  return {
    counter: previousState.counter + 1
  };
});
```

Esto puede ser más seguro que usar un argumento de objeto donde se usan múltiples llamadas a `setState()` , ya que React puede agrupar varias llamadas y se ejecutan a la vez, y es el enfoque preferido cuando se usan accesorios actuales para establecer el estado.

```
this.setState({ counter: this.state.counter + 1 });
this.setState({ counter: this.state.counter + 1 });
this.setState({ counter: this.state.counter + 1 });
```

Estas llamadas pueden agruparse mediante React utilizando `Object.assign()` , lo que hace que el contador se incremente en 1 en lugar de 3.

El enfoque funcional también se puede utilizar para mover la lógica de configuración de estado fuera de los componentes. Esto permite el aislamiento y la reutilización de la lógica de estado.

```
// Outside of component class, potentially in another file/module

function incrementCounter(previousState, currentProps) {
    return {
        counter: previousState.counter + 1
    };
}

// Within component

this.setState(incrementCounter);
```

Llamar a `setState()` con un objeto y una función de devolución de llamada

```
//
// 'Hi There' will be logged to the console after setState completes
//

this.setState({ name: 'John Doe' }, console.log('Hi there'));
```

Antipattern común

No debes guardar `props` en `state` . Se considera un anti-patrón . Por ejemplo:

```
export default class MyComponent extends React.Component {
    constructor() {
        super();

        this.state = {
            url: ''
        }

        this.onChange = this.onChange.bind(this);
    }

    onChange(e) {
        this.setState({
            url: this.props.url + '/days=?' + e.target.value
        });
    }
```

```
    componentWillMount() {
        this.setState({url: this.props.url});
    }

    render() {
        return (
            <div>
                <input defaultValue={2} onChange={this.onChange} />

                URL: {this.state.url}
            </div>
        )
    }
}
```

La `url` apoyo se guarda en el `state` y luego se modifica. En su lugar, elija guardar los cambios en un estado y luego genere la ruta completa utilizando tanto el `state` como los `props` :

```
export default class MyComponent extends React.Component {
    constructor() {
        super();

        this.state = {
            days: ''
        }

        this.onChange = this.onChange.bind(this);
    }

    onChange(e) {
        this.setState({
            days: e.target.value
        });
    }

    render() {
        return (
            <div>
                <input defaultValue={2} onChange={this.onChange} />

                URL: {this.props.url + '/days?=' + this.state.days}
            </div>
        )
    }
}
```

Esto se debe a que en una aplicación React queremos tener una única fuente de verdad, es decir, todos los datos son responsabilidad de un solo componente, y solo de un componente. Es responsabilidad de este componente almacenar los datos dentro de su estado y distribuirlos a otros componentes a través de accesorios.

En el primer ejemplo, tanto la clase MyComponent como su padre mantienen el 'url' dentro de su estado. Si actualizamos state.url en MyComponent, estos cambios no se reflejan en el padre. Hemos perdido nuestra única fuente de verdad, y cada vez es más difícil rastrear el flujo de datos a través de nuestra aplicación. Contraste esto con el segundo ejemplo: la url solo se mantiene en el estado del componente principal, y se utiliza como accesorio en MyComponent; por lo tanto,

mantenemos una única fuente de verdad.

Estado, Eventos y Controles Gestionados.

Aquí hay un ejemplo de un componente React con un campo de entrada "gestionado". Cuando el valor del campo de entrada cambia, se llama a un controlador de eventos que actualiza el estado del componente con el nuevo valor del campo de entrada. La llamada a `setState` en el controlador de eventos activará una llamada para `render` actualización del componente en el dom.

```
import React from 'react';
import {render} from 'react-dom';


class ManagedControlDemo extends React.Component {

  constructor(props){
    super(props);
    this.state = {message: ""};
  }

  handleChange(e){
    this.setState({message: e.target.value});
  }

  render() {
    return (
      <div>
        <legend>Type something here</legend>
          <input
            onChange={this.handleChange.bind(this)}
            value={this.state.message}
            autoFocus />
        <h1>{this.state.message}</h1>
      </div>
   );
  }
}


render(<ManagedControlDemo/>, document.querySelector('#app'));
```

Es muy importante tener en cuenta el comportamiento en tiempo de ejecución. Cada vez que un usuario cambia el valor en el campo de entrada

- `handleChange` será llamado y así
- `setState` llamará `setState` y así
- se llamará `render`

Pop quiz, después de escribir un carácter en el campo de entrada, qué elementos DOM cambian

1. todos estos - el div de nivel superior, leyenda, entrada, h1
2. solo la entrada y h1
3. nada
4. ¿Qué es un DOM?

Puedes experimentar más aquí para encontrar la respuesta.

Lea Estado en reaccionar en línea: https://riptutorial.com/es/reactjs/topic/1816/estado-en-reaccionar

Capítulo 13: Formularios y comentarios del usuario

Examples

Componentes controlados

Los componentes de formulario controlados se definen con una propiedad de `value` . El valor de las entradas controladas es administrado por React, las entradas del usuario no tendrán ninguna influencia directa en la entrada renderizada. En cambio, un cambio en la propiedad de `value` debe reflejar este cambio.

```
class Form extends React.Component {
  constructor(props) {
    super(props);

    this.onChange = this.onChange.bind(this);

    this.state = {
      name: ''
    };
  }

  onChange(e) {
    this.setState({
      name: e.target.value
    });
  }

  render() {
    return (
      <div>
        <label for='name-input'>Name: </label>
        <input
          id='name-input'
          onChange={this.onChange}
          value={this.state.name} />
      </div>
    )
  }
}
```

El ejemplo anterior demuestra cómo la propiedad de `value` define el valor actual de la entrada y el controlador de eventos `onChange` actualiza el estado del componente con la entrada del usuario.

Las entradas de formularios deben definirse como componentes controlados cuando sea posible. Esto garantiza que el estado del componente y el valor de entrada estén sincronizados en todo momento, incluso si el valor es cambiado por un disparador que no sea una entrada del usuario.

Componentes no controlados

Los componentes no controlados son entradas que no tienen una propiedad de `value` . Al contrario de los componentes controlados, es responsabilidad de la aplicación mantener el estado del componente y el valor de entrada sincronizados.

```
class Form extends React.Component {
  constructor(props) {
    super(props);

    this.onChange = this.onChange.bind(this);

    this.state = {
      name: 'John'
    };
  }

  onChange(e) {
    this.setState({
      name: e.target.value
    });
  }

  render() {
    return (
      <div>
        <label for='name-input'>Name: </label>
        <input
          id='name-input'
          onChange={this.onChange}
          defaultValue={this.state.name} />
      </div>
    )
  }
}
```

Aquí, el estado del componente se actualiza a través del controlador de eventos `onChange` , al igual que para los componentes controlados. Sin embargo, en lugar de una propiedad de `value` , se proporciona una propiedad `defaultValue` . Esto determina el valor inicial de la entrada durante el primer renderizado. Cualquier cambio posterior en el estado del componente no se refleja automáticamente por el valor de entrada; Si es necesario, se debe utilizar un componente controlado en su lugar.

Lea Formularios y comentarios del usuario en línea:
https://riptutorial.com/es/reactjs/topic/2884/formularios-y-comentarios-del-usuario

Capítulo 14: Instalación

Examples

Configuración simple

Configurando las carpetas

Este ejemplo asume que el código está en `src/` y que la salida se pondrá en `out/` . Como tal, la estructura de la carpeta debería verse como

```
example/
|-- src/
|   |-- index.js
|   `-- ...
|-- out/
`-- package.json
```

Configurando los paquetes

Asumiendo un entorno de configuración de npm, primero debemos configurar babel para transpilar el código React a un código compatible con es5.

```
$npm install --save-dev babel-core babel-loader babel-preset-es2015 babel-preset-react
```

El comando anterior indicará a npm que instale las bibliotecas centrales de babel, así como el módulo de carga para usar con el paquete web. También instalamos los ajustes preestablecidos de es6 y reactivos para que Babel entienda el código de módulo JSX y es6. (Puede encontrar más información acerca de los ajustes preestablecidos aquí presets de Babel)

```
$npm i -D webpack
```

Este comando instalará webpack como una dependencia de desarrollo. (**i** es la abreviatura para instalar y **-D** la abreviatura para --save-dev)

Es posible que también desee instalar cualquier paquete webpack adicional (como cargadores adicionales o la extensión webpack-dev-server)

Por último necesitaremos el código de reacción real.

```
$npm i -D react react-dom
```

Configuración de webpack

Con la configuración de dependencias necesitaremos un archivo webpack.config.js para decirle a webpack qué hacer

simple webpack.config.js:

```
var path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'out'),
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: /(node_modules)/,
        loader: 'babel-loader',
        query: {
          presets: ['es2015', 'react']
        }
      }
    ]
  }
};
```

Este archivo le dice a webpack que comience con el archivo index.js (se supone que está en src /) y lo convierte en un solo archivo bundle.js en el directorio `out` .

El bloque de `module` le dice a webpack que pruebe todos los archivos encontrados en la expresión regular y, si coinciden, invocará el cargador especificado. (`babel-loader` en este caso) Además, `exclude` regex le dice a webpack que ignore este cargador especial para todos los módulos en la carpeta `node_modules` , esto ayuda a acelerar el proceso de transpilación. Por último, la opción de `query` le dice a webpack qué parámetros pasar a babel y se usa para transmitir los ajustes preestablecidos que instalamos anteriormente.

Probando la configuración

Todo lo que queda ahora es crear el archivo `src/index.js` e intentar empaquetar la aplicación

src / index.js:

```
'use strict'

import React from 'react'
import { render } from 'react-dom'

const App = () => {
  return <h1>Hello world!</h1>
}

render(
```

```
  <App />,
  document. getElementById('app')
)
```

Normalmente, este archivo representará un simple `<h1>Hello world!</h1>` Encabezado en la etiqueta html con el id 'app', pero por ahora debería ser suficiente para recopilar el código una vez.

`$./node_modules/.bin/webpack` . Ejecutará la versión localmente instalada de webpack (use `$webpack` si instaló webpack globalmente con -g)

Esto debería crear el archivo `out/bundle.js` con el código transpilado dentro y concluir el ejemplo.

Utilizando webpack-dev-server

Preparar

Después de configurar un proyecto simple para usar webpack, babel y reaccionan emitiendo `$npm i -g webpack-dev-server` instalará el servidor http de desarrollo para un desarrollo más rápido.

Modificando webpack.config.js

```
var path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'out'),
    publicPath: '/public/',
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: /(node_modules)/,
        loader: 'babel',
        query: {
          presets: ['es2015', 'react']
        }
      }
    ]
  },
  devServer: {
    contentBase: path.resolve(__dirname, 'public'),
    hot: true
  }
};
```

Las modificaciones estan en

- `output.publicPath` que configura una ruta desde la cual se sirve nuestro paquete (consulte los archivos de configuración del paquete web para obtener más información)
- `devServer`
    - `contentBase` la ruta base para servir archivos estáticos (por ejemplo, `index.html`)
    - `hot` conjuntos del webpack-dev-servidor para recargar caliente cuando se hacen los cambios a los archivos en el disco

Y, finalmente, solo necesitamos un index.html simple para probar nuestra aplicación.

index.html:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>React Sandbox</title>
  </head>
  <body>

    <div id="app" />

    <script src="public/bundle.js"></script>
  </body>
</html>
```

Con esta configuración ejecutando `$webpack-dev-server` debe iniciar un servidor http local en el puerto 8080 y al conectarse debe mostrar una página que contenga un `<h1>Hello world!</h1>` .

Lea Instalación en línea: https://riptutorial.com/es/reactjs/topic/6441/instalacion

Capítulo 15: Instalación de React, Webpack y Typescript.

Observaciones

Para obtener el resaltado de sintaxis en su editor (por ejemplo, VS Code), deberá descargar la información de escritura para los módulos que utiliza en su proyecto.

Digamos, por ejemplo, que utiliza React y ReactDOM en su proyecto, y desea obtener resaltado e Intellisense para ellos. Deberá agregar los tipos a su proyecto usando este comando:

```
npm install --save @types/react @types/react-dom
```

Su editor ahora debe seleccionar automáticamente esta información de escritura y proporcionarle autocompletado e Intellisense para estos módulos.

Examples

webpack.config.js

```
module.exports = {
    entry: './src/index',
    output: {
        path: __dirname + '/build',
        filename: 'bundle.js'
    },
    module: {
        rules: [{
            test: /\.tsx?$/,
            loader: 'ts-loader',
            exclude: /node_modules/
        }]
    },
    resolve: {
        extensions: ['.ts', '.tsx']
    }
};
```

Los componentes principales son (además de la `entry` estándar, la `output` y otras propiedades del paquete web):

El cargador

Para esto necesita crear una regla que pruebe las extensiones de archivo `.ts` y `.tsx` , especifique `ts-loader` como el cargador.

Resolver las extensiones TS

También debe agregar las `.ts` y `.tsx` en la matriz de `resolve` , o el paquete web no las verá.

tsconfig.json

Este es un tsconfig mínimo para ponerlo en marcha.

```
{
    "include": [
        "src/*"
    ],
    "compilerOptions": {
        "target": "es5",
        "jsx": "react",
        "allowSyntheticDefaultImports": true
    }
}
```

Vamos a ver las propiedades una por una:

`include`

Esta es una matriz de código fuente. Aquí solo tenemos una entrada, `src/*` , que especifica que todo en el directorio `src` debe incluirse en la compilación.

`compilerOptions.target`

Especifica que queremos compilar a ES5 target.

`compilerOptions.jsx`

Establecer esto en `true` hará que TypeScript compile automáticamente su sintaxis tsx desde `<div />` a `React.createElement("div")` .

`compilerOptions.allowSyntheticDefaultImports`

Propiedad práctica que le permitirá importar módulos de nodo como si fueran módulos ES6, así que en lugar de hacerlo

```
import * as React from 'react'
const { Component } = React
```

solo puedes hacer

```
import React, { Component } from 'react'
```

sin ningún error que le indique que React no tiene una exportación predeterminada.

Mi primer componente

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';


interface AppProps {
    name: string;
}
interface AppState {
    words: string[];
}

class App extends Component<AppProps, AppState> {
    constructor() {
        super();
        this.state = {
            words: ['foo', 'bar']
        };
    }

    render() {
        const { name } = this.props;
        return (<h1>Hello {name}!</h1>);
    }
}

const root = document.getElementById('root');
ReactDOM.render(<App name="Foo Bar" />, root);
```

Cuando use TypeScript con React, una vez que haya descargado las definiciones de tipo React DefinitelyTyped (`npm install --save @types/react`), cada componente requerirá que agregue anotaciones de tipo.

Haces esto así

```
class App extends Component<AppProps, AppState> { }
```

donde `AppProps` y `AppState` son interfaces (o alias de tipo) para las `AppProps` y el `AppState` de sus componentes, respectivamente.

Lea Instalación de React, Webpack y Typescript. en línea: https://riptutorial.com/es/reactjs/topic/9590/instalacion-de-react--webpack-y-typescript-

Capítulo 16: Introducción a la representación del lado del servidor

Examples

Componentes de renderizado

Hay dos opciones para representar componentes en el servidor: `renderToString` y `renderToStaticMarkup` .

renderToString

Esto renderizará los componentes React a HTML en el servidor. Esta función también agregará propiedades de `data-react-` a elementos HTML, por lo que React on client no tendrá que volver a generar elementos.

```
import { renderToString } from "react-dom/server";
renderToString(<App />);
```

renderToStaticMarkup

Esto convertirá los componentes de React en HTML, pero sin `data-react-` propiedades de `data-react-` , no se recomienda usar componentes que se representarán en el cliente, ya que los componentes se redireccionarán.

```
import { renderToStaticMarkup } from "react-dom/server";
renderToStaticMarkup(<App />);
```

Lea Introducción a la representación del lado del servidor en línea: https://riptutorial.com/es/reactjs/topic/7478/introduccion-a-la-representacion-del-lado-del-servidor

Capítulo 17: JSX

Observaciones

JSX es un **paso de preprocesador** que agrega sintaxis XML a JavaScript. Definitivamente, puedes usar React sin JSX, pero JSX hace que React sea mucho más elegante.

Al igual que XML, las etiquetas JSX tienen un nombre de etiqueta, atributos e hijos. Si un valor de atributo se incluye entre comillas, el valor es una cadena. De lo contrario, ajuste el valor entre llaves y el valor es la expresión de JavaScript incluida.

Fundamentalmente, JSX solo proporciona azúcar sintáctica para la función `React.createElement(component, props, ...children)` .

Por lo tanto, el siguiente código JSX:

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

ReactDOM.render(<HelloMessage name="Kalo" />, mountNode);
```

Compila hasta el siguiente código de JavaScript:

```
class HelloMessage extends React.Component {
  render() {
    return React.createElement(
      "div",
      null,
      "Hello ",
      this.props.name
    );
  }
}

ReactDOM.render(React.createElement(HelloMessage, { name: "Kalo" }), mountNode);
```

En conclusión, tenga en cuenta que **la siguiente línea en JSX no es una cadena ni HTML** :

```
const element = <h1>Hello, world!</h1>;
```

Se llama JSX y es una **extensión de sintaxis para JavaScript** . JSX puede recordarle un lenguaje de plantilla, pero viene con todo el poder de JavaScript.

El equipo de React dice en sus documentos que recomiendan usarlo para describir cómo debería ser la interfaz de usuario.

Examples

Puntales en JSX

Hay varias formas diferentes de especificar props en JSX.

Expresiones de JavaScript

Puedes pasar **cualquier expresión de JavaScript** como prop, al rodearla con `{}` . Por ejemplo, en este JSX:

```
<MyComponent count={1 + 2 + 3 + 4} />
```

Dentro de `MyComponent` , el valor de `props.count` será `10` , porque se evalúa la expresión `1 + 2 + 3 + 4` .

Si las declaraciones y los bucles no son expresiones en JavaScript, no se pueden usar directamente en JSX.

Literales de cuerda

Por supuesto, también puedes pasar cualquier `string literal` como `prop` . Estas dos expresiones JSX son equivalentes:

```
<MyComponent message="hello world" />

<MyComponent message={'hello world'} />
```

Cuando se pasa un literal de cadena, su valor no se guarda en HTML. Así que estas dos expresiones JSX son equivalentes:

```
<MyComponent message="&lt;3" />

<MyComponent message={'<3'} />
```

Este comportamiento no suele ser relevante. Sólo se menciona aquí para completar.

Valor predeterminado de accesorios

Si no pasa ningún valor para un objeto, el valor **predeterminado es** `true` . Estas dos expresiones JSX son equivalentes:

```
<MyTextBox autocomplete />

<MyTextBox autocomplete={true} />
```

Sin embargo, el equipo de React dice que **no se recomienda el uso de este** documento en sus documentos, ya que puede confundirse con la abreviatura del objeto ES6 `{foo}` que es la abreviatura de `{foo: foo}` lugar de `{foo: true}` . Dicen que este comportamiento está ahí para que coincida con el comportamiento de HTML.

Atributos de propagación

Si ya tiene objetos como objeto y desea pasarlos a JSX, puede usar `...` como operador de extensión para pasar todo el objeto de objetos. Estos dos componentes son equivalentes:

```
function Case1() {
  return <Greeting firstName="Kaloyab" lastName="Kosev" />;
}

function Case2() {
  const person = {firstName: 'Kaloyan', lastName: 'Kosev'};
  return <Greeting {...person} />;
}
```

Niños en JSX

En las expresiones JSX que contienen una etiqueta de apertura y una etiqueta de cierre, el contenido entre esas etiquetas se pasa como un elemento especial: `props.children` . Hay varias maneras diferentes de pasar a los niños:

Literales de cuerda

Puede colocar una cadena entre las etiquetas de apertura y cierre y `props.children` solo será esa cadena. Esto es útil para muchos de los elementos HTML integrados. Por ejemplo:

```
<MyComponent>
    <h1>Hello world!</h1>
</MyComponent>
```

Esto es JSX válido, y `props.children` en MyComponent simplemente será `<h1>Hello world!</h1>` .

Tenga en cuenta que **el HTML no se puede escapar** , por lo que generalmente puede escribir JSX como lo haría con HTML.

Ten en cuenta que en este caso JSX:

- elimina los espacios en blanco al principio y al final de una línea;

- elimina las líneas en blanco;
- Se eliminan las nuevas líneas adyacentes a las etiquetas;
- Las nuevas líneas que se producen en medio de cadenas literales se condensan en un solo espacio.

Niños jsx

Puede proporcionar más elementos JSX como los hijos. Esto es útil para mostrar componentes anidados:

```
<MyContainer>
  <MyFirstComponent />
  <MySecondComponent />
</MyContainer>
```

Puedes **mezclar diferentes tipos de niños, así que puedes usar literales de cuerdas junto con niños JSX** . Esta es otra forma en la que JSX es como HTML, de modo que es tanto JSX válido como HTML válido:

```
<div>
  <h2>Here is a list</h2>
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
  </ul>
</div>
```

Tenga en cuenta que un componente React **no puede devolver varios elementos React, pero una sola expresión JSX puede tener varios hijos** . Entonces, si quieres que un componente genere varias cosas, puedes envolverlo en un `div` como el ejemplo anterior.

Expresiones de JavaScript

Puede pasar cualquier expresión de JavaScript como hijos, encerrándola dentro de `{}` . Por ejemplo, estas expresiones son equivalentes:

```
<MyComponent>foo</MyComponent>

<MyComponent>{'foo'}</MyComponent>
```

Esto suele ser útil para representar una lista de expresiones JSX de longitud arbitraria. Por ejemplo, esto hace que una lista HTML:

```
const Item = ({ message }) => (
  <li>{ message }</li>
);
```

```
const TodoList = () => {
  const todos = ['finish doc', 'submit review', 'wait stackoverflow review'];
  return (
    <ul>
      { todos.map(message => (<Item key={message} message={message} />)) }
    </ul>
  );
};
```

Tenga en cuenta que las expresiones de JavaScript se pueden mezclar con otros tipos de niños.

Funciones como niños

Normalmente, las expresiones de JavaScript insertadas en JSX se evaluarán como una cadena, un elemento React o una lista de esas cosas. Sin embargo, `props.children` funciona igual que cualquier otro apoyo, ya que puede pasar cualquier tipo de datos, no solo los tipos que React sabe cómo renderizar. Por ejemplo, si tiene un componente personalizado, podría hacer que tome una devolución de llamada como `props.children` :

```
const ListOfTenThings = () => (
  <Repeat numTimes={10}>
    {(index) => <div key={index}>This is item {index} in the list</div>}
  </Repeat>
);

// Calls the children callback numTimes to produce a repeated component
const Repeat = ({ numTimes, children }) => {
  let items = [];
  for (let i = 0; i < numTimes; i++) {
    items.push(children(i));
  }
  return <div>{items}</div>;
};
```

Los niños pasados a un componente personalizado pueden ser cualquier cosa, siempre que ese componente los transforme en algo que React pueda entender antes de renderizar. Este uso no es común, pero funciona si desea ampliar lo que JSX es capaz de hacer.

Valores ignorados

Tenga en cuenta que `false` , `null` , `undefined` y `true` son hijos válidos. Pero simplemente no se renderizan. Estas expresiones JSX se representarán todas a la misma cosa:

```
<MyComponent />

<MyComponent></MyComponent>
```

```
<MyComponent>{false}</MyComponent>

<MyComponent>{null}</MyComponent>

<MyComponent>{true}</MyComponent>
```

Esto es extremadamente útil para renderizar condicionalmente los elementos React. Este JSX solo muestra si `showHeader` es verdadero:

```
<div>
  {showHeader && <Header />}
  <Content />
</div>
```

Una advertencia importante es que React sigue procesando algunos valores "falsos", como el número `0` . Por ejemplo, este código no se comportará como cabría esperar porque se imprimirá `0` cuando `props.messages` es una matriz vacía:

```
<div>
  {props.messages.length &&
    <MessageList messages={props.messages} />
  }
</div>
```

Un enfoque para solucionar esto es asegurarse de que la expresión antes de `&&` siempre sea booleana:

```
<div>
  {props.messages.length > 0 &&
    <MessageList messages={props.messages} />
  }
</div>
```

Por último, recuerde que si desea que aparezca un valor como `false` , `true` , `null` o `undefined` en la salida, primero debe convertirlo en una cadena:

```
<div>
  My JavaScript variable is {String(myVariable)}.
</div>
```

Lea JSX en línea: https://riptutorial.com/es/reactjs/topic/8027/jsx

Capítulo 18: Reaccionar con redux

Introducción

Redux se ha convertido en el statu quo para administrar el estado de nivel de aplicación en el front-end en estos días, y los que trabajan en "aplicaciones a gran escala" a menudo confían en ello. Este tema cubre por qué y cómo debe usar la biblioteca de administración de estado, Redux, en sus aplicaciones React.

Observaciones

Si bien la arquitectura basada en componentes de React es fantástica para dividir la aplicación en pequeñas piezas encapsuladas y modulares, presenta algunos desafíos para administrar el estado de la aplicación en su totalidad. El momento de usar Redux es cuando necesita mostrar los mismos datos en más de un componente o página (también conocida como ruta). En ese momento, ya no puede almacenar los datos en variables locales para un componente u otro, y el envío de mensajes entre componentes se convierte rápidamente en un desastre. Con Redux, todos sus componentes se están suscribiendo a los mismos datos compartidos en la tienda y, por lo tanto, el estado se puede reflejar fácilmente de manera consistente en toda la aplicación.

Examples

Usando Connect

Crear una tienda de Redux con *createStore* .

```
import { createStore } from 'redux'
import todoApp from './reducers'
let store = createStore(todoApp, { inistialStateVariable: "derp"})
```

Use *conectar* para conectar el componente a la tienda Redux y tire de los accesorios de la tienda al componente.

```
import { connect } from 'react-redux'

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList
```

Defina acciones que permitan a sus componentes enviar mensajes al almacén de Redux.

```
/*
 * action types
```

```
 */

export const ADD_TODO = 'ADD_TODO'

export function addTodo(text) {
  return { type: ADD_TODO, text }
}
```

Maneje estos mensajes y cree un nuevo estado para la tienda en funciones reductoras.

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    default:
      return state
  }
}
```

Lea Reaccionar con redux en línea: https://riptutorial.com/es/reactjs/topic/10856/reaccionar-con-redux

Capítulo 19: Reaccionar enrutamiento

Examples

Ejemplo del archivo Routes.js, seguido del uso de Router Link en el componente

Coloque un archivo como el siguiente en su directorio de nivel superior. Define qué componentes representar para qué rutas

```
import React from 'react';
import { Route, IndexRoute } from 'react-router';
import New from './containers/new-post';
import Show from './containers/show';

import Index from './containers/home';
import App from './components/app';

export default(
  <Route path="/" component={App}>
    <IndexRoute component={Index} />
    <Route path="posts/new" component={New} />
    <Route path="posts/:id" component={Show} />

  </Route>
);
```

Ahora en su index.js de nivel superior, que es su punto de entrada a la aplicación, solo necesita renderizar este componente de Enrutador así:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Router, browserHistory } from 'react-router';
// import the routes component we created in routes.js
import routes from './routes';


// entry point
ReactDOM.render(
    <Router history={browserHistory} routes={routes} />
  , document.getElementById('main'));
```

Ahora es simplemente una cuestión de usar las etiquetas `Link` lugar de `<a>` toda la aplicación. El uso de Link se comunicará con React Router para cambiar la ruta de React Router al enlace especificado, que a su vez generará el componente correcto tal como se define en route.js

```
import React from 'react';
import { Link } from 'react-router';
```

```
export default function PostButton(props) {
  return (
    <Link to={`posts/${props.postId}`}>
      <div className="post-button" >
        {props.title}
        <span>{props.tags}</span>
      </div>
    </Link>
  );
}
```

Reaccionar enrutamiento asíncrono

```
import React from 'react';
import { Route, IndexRoute } from 'react-router';

import Index from './containers/home';
import App from './components/app';

//for single Component lazy load use this
const ContactComponent = () => {
  return {
      getComponent: (location, callback)=> {
        require.ensure([], require => {
          callback(null, require('./components/Contact')["default"]);
        }, 'Contact');
      }
    }
};

//for multiple componnets
 const groupedComponents = (pageName) => {
  return {
      getComponent: (location, callback)=> {
        require.ensure([], require => {
          switch(pageName){
            case 'about' :
                callback(null, require( "./components/about" )["default"]);
                    break ;
            case 'tos' :
                callback(null, require( "./components/tos" )["default"]);
                    break ;
          }
        }, "groupedComponents");
      }
    }
};
export default(
  <Route path="/" component={App}>
    <IndexRoute component={Index} />
    <Route path="/contact" {...ContactComponent()} />
    <Route path="/about" {...groupedComponents('about')} />
    <Route path="/tos" {...groupedComponents('tos')} />
  </Route>
);
```

Lea Reaccionar enrutamiento en línea: https://riptutorial.com/es/reactjs/topic/6096/reaccionar-enrutamiento

Capítulo 20: Reaccionar formas

Examples

Componentes controlados

Un componente controlado está vinculado a un valor y sus cambios se manejan en el código mediante devoluciones de llamada basadas en eventos.

```
class CustomForm extends React.Component {
constructor() {
    super();
    this.state = {
        person: {
            firstName: '',
            lastName: ''
        }
    }
}

handleChange(event) {
    let person = this.state.person;
    person[event.target.name] = event.target.value;
    this.setState({person});
}

render() {
    return (
        <form>
            <input
            type="text"
            name="firstName"
            value={this.state.firstName}
            onChange={this.handleChange.bind(this)} />

            <input
            type="text"
            name="lastName"
            value={this.state.lastName}
            onChange={this.handleChange.bind(this)} />
        </form>
    )
}
```

}

En este ejemplo, inicializamos el estado con un objeto persona vacío. Luego vinculamos los valores de las 2 entradas a las claves individuales del objeto persona. Luego, a medida que el usuario escribe, `handleChange` cada valor en la función `handleChange` . Dado que los valores de los componentes están vinculados al estado, podemos redirigirlos a los tipos de usuario llamando a `setState()` .

NOTA: No llamar a `setState()` cuando se trata con componentes controlados, hará que el usuario

escriba, pero no verá la entrada porque React solo presenta los cambios cuando se le indica que lo haga.

También es importante tener en cuenta que los nombres de las entradas son los mismos que los nombres de las teclas en el objeto persona. Esto nos permite capturar el valor en forma de diccionario como se ve aquí.

```
handleChange(event) {
    let person = this.state.person;
    person[event.target.name] = event.target.value;
    this.setState({person});
}
```

`person[event.target.name]` es el mismo es `person.firstName || person.lastName` . Por supuesto, esto dependería de la entrada que se está escribiendo actualmente. Ya que no sabemos dónde escribirá el usuario, usar un diccionario y hacer coincidir los nombres de entrada con los nombres de las teclas, nos permite capturar la entrada del usuario no importa desde donde se llama el onChange.

Lea Reaccionar formas en línea: https://riptutorial.com/es/reactjs/topic/8047/reaccionar-formas

Capítulo 21: Reaccionar herramientas

Examples

Campo de golf

Lugares para encontrar React componentes y bibliotecas;

- Catálogo de componentes React
- JS.coach

Lea Reaccionar herramientas en línea: https://riptutorial.com/es/reactjs/topic/6595/reaccionar-herramientas

Capítulo 22: Reaccionar llamada AJAX

Examples

Solicitud HTTP GET

A veces, un componente necesita generar algunos datos desde un punto final remoto (por ejemplo, una API REST). Una práctica estándar es hacer tales llamadas en el método `componentDidMount` .

Aquí hay un ejemplo, usando superagente como ayudante de AJAX:

```
import React from 'react'
import request from 'superagent'

class App extends React.Component {
  constructor () {
    super()
    this.state = {}
  }
  componentDidMount () {
    request
      .get('/search')
      .query({ query: 'Manny' })
      .query({ range: '1..5' })
      .query({ order: 'desc' })
      .set('API-Key', 'foobar')
      .set('Accept', 'application/json')
      .end((err, resp) => {
        if (!err) {
          this.setState({someData: resp.text})
        }
      })
  },
  render() {
    return (
      <div>{this.state.someData || 'waiting for response...'}</div>
    )
  }
}

React.render(<App />, document.getElementById('root'))
```

Se puede iniciar una solicitud invocando el método apropiado en el objeto de `request` , luego llamando a `.end()` para enviar la solicitud. Establecer campos de encabezado es simple, invoque `.set()` con un nombre y valor de campo.

El método `.query()` acepta objetos, que cuando se usan con el método GET formarán una cadena de consulta. Lo siguiente producirá la ruta `/search?query=Manny&range=1..5&order=desc` .

Solicitudes **POST**

```
request.post('/user')
  .set('Content-Type', 'application/json')
  .send('{"name":"tj","pet":"tobi"}')
  .end(callback)
```

Ver documentos Superagent para más detalles.

Ajax en Reaccionar sin una biblioteca de terceros, también conocido como VanillaJS

Lo siguiente funcionaría en IE9 +

```
import React from 'react'

class App extends React.Component {
  constructor () {
    super()
    this.state = {someData: null}
  }
  componentDidMount () {
    var request = new XMLHttpRequest();
    request.open('GET', '/my/url', true);

    request.onload = () => {
      if (request.status >= 200 && request.status < 400) {
        // Success!
        this.setState({someData: request.responseText})
      } else {
        // We reached our target server, but it returned an error
        // Possibly handle the error by changing your state.
      }
    };

    request.onerror = () => {
      // There was a connection error of some sort.
      // Possibly handle the error by changing your state.
    };

    request.send();
  },
  render() {
    return (
      <div>{this.state.someData || 'waiting for response...'}</div>
    )
  }
}

React.render(<App />, document.getElementById('root'))
```

Solicitud HTTP GET y bucle a través de los datos

El siguiente ejemplo muestra cómo un conjunto de datos obtenidos de una fuente remota se puede representar en un componente.

Hacemos una solicitud AJAX usando `fetch` , que está integrada en la mayoría de los navegadores. Utilice un polyfill de `fetch` en producción para admitir navegadores más antiguos.

También se puede utilizar cualquier otra biblioteca para hacer peticiones (por ejemplo axios , SuperAgent , o incluso llanura Javascript).

Establecemos los datos que recibimos como estado de componente, por lo que podemos acceder a ellos dentro del método de procesamiento. Allí, recorremos los datos utilizando el `map` . No se olvide de agregar siempre un atributo `key` único (o prop) al elemento en bucle, que es importante para el rendimiento de representación de React.

```
import React from 'react';

class Users extends React.Component {
  constructor() {
    super();
    this.state = { users: [] };
  }

  componentDidMount() {
    fetch('/api/users')
      .then(response => response.json())
      .then(json => this.setState({ users: json.data }));
  }

  render() {
    return (
      <div>
        <h1>Users</h1>
        {
          this.state.users.length == 0
            ? 'Loading users...'
            : this.state.users.map(user => (
              <figure key={user.id}>
                <img src={user.avatar} />
                <figcaption>
                  {user.name}
                </figcaption>
              </figure>
            ))
        }
      </div>
    );
  }
}

ReactDOM.render(<Users />, document.getElementById('root'));
```

Ejemplo de trabajo en JSBin .

Lea Reaccionar llamada AJAX en línea: https://riptutorial.com/es/reactjs/topic/6432/reaccionar-llamada-ajax

Capítulo 23: React Boilerplate [React + Babel + Webpack]

Examples

Configurando el proyecto

Necesita Node Package Manager para instalar las dependencias del proyecto. Descargue el nodo para su sistema operativo desde Nodejs.org . Node Package Manager viene con nodo.

También puede usar Node Version Manager para administrar mejor sus versiones de nodo y npm. Es ideal para probar su proyecto en diferentes versiones de nodos. Sin embargo, no se recomienda para el entorno de producción.

Una vez que haya instalado el nodo en su sistema, siga adelante e instale algunos paquetes esenciales para explotar su primer proyecto React utilizando Babel y Webpack.

Antes de que realmente comencemos a golpear comandos en el terminal. Eche un vistazo a para qué se utilizan Babel y Webpack .

Puede iniciar su proyecto ejecutando `npm init` en su terminal. Siga la configuración inicial. Después de eso, ejecute los siguientes comandos en su terminal-

Dependencias:

```
npm install react react-dom --save
```

Dev Dependecies:

```
npm install babel-core babel-loader babel-preset-es2015 babel-preset-react babel-preset-stage-0
webpack webpack-dev-server react-hot-loader --save-dev
```

Dependencias de desarrollo opcional:

```
npm install eslint eslint-plugin-react babel-eslint --save-dev
```

Puede hacer referencia a esta muestra de package.json

Cree `.babelrc` en la raíz de su proyecto con los siguientes contenidos:

```
{
  "presets": ["es2015", "stage-0", "react"]
}
```

Opcionalmente, cree `.eslintrc` en la raíz de su proyecto con los siguientes contenidos:

```
{
  "ecmaFeatures": {
    "jsx": true,
```

```
    "modules": true
  },
  "env": {
    "browser": true,
    "node": true
  },
  "parser": "babel-eslint",
  "rules": {
    "quotes": [2, "single"],
    "strict": [2, "never"],
  },
  "plugins": [
    "react"
  ]
}
```

Cree un archivo `.gitignore` para evitar cargar archivos generados en su repositorio git.

```
node_modules
npm-debug.log
.DS_Store
dist
```

Cree el archivo `webpack.config.js` con los siguientes contenidos mínimos.

```
var path = require('path');
var webpack = require('webpack');

module.exports = {
  devtool: 'eval',
  entry: [
    'webpack-dev-server/client?http://localhost:3000',
    'webpack/hot/only-dev-server',
    './src/index'
  ],
  output: {
    path: path.join(__dirname, 'dist'),
    filename: 'bundle.js',
    publicPath: '/static/'
  },
  plugins: [
    new webpack.HotModuleReplacementPlugin()
  ],
  module: {
    loaders: [{
      test: /\.js$/,
      loaders: ['react-hot', 'babel'],
      include: path.join(__dirname, 'src')
    }]
  }
};
```

Y finalmente, cree un archivo `sever.js` para poder ejecutar `npm start` , con los siguientes contenidos:

```
var webpack = require('webpack');
var WebpackDevServer = require('webpack-dev-server');
```

```
var config = require('./webpack.config');

new WebpackDevServer(webpack(config), {
  publicPath: config.output.publicPath,
  hot: true,
  historyApiFallback: true
}).listen(3000, 'localhost', function (err, result) {
  if (err) {
    return console.log(err);
  }

  console.log('Serving your awesome project at http://localhost:3000/');
});
```

Cree el archivo `src/app.js` para ver cómo su proyecto React hace algo.

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

export default class App extends Component {
  render() {
    return (
      <h1>Hello, world.</h1>
    );
  }
}
```

Ejecute `node server.js` o `npm start` en el terminal, si ha definido lo que significa `start` en su `package.json`

proyecto react-starter

Sobre este proyecto

Este es un proyecto simple. Esta publicación te guiará para configurar el entorno para ReactJs + Webpack + Bable.

Empecemos

necesitaremos el administrador de paquetes de nodos para encender el servidor Express y administrar las dependencias a lo largo del proyecto. Si es nuevo en el administrador de paquetes de nodos, puede consultar aquí . Nota: aquí se requiere la instalación del administrador de paquetes de nodos.

Cree una carpeta con un nombre adecuado y navegue en ella desde la terminal o por GUI. Luego, vaya a la terminal y escriba `npm init` Esto creará un archivo package.json. Nada de miedo, le hará algunas preguntas como el nombre de su proyecto, versión, descripción, punto de entrada, repositorio git, autor, licencia, etc. Aquí, el punto de entrada es importante porque el nodo lo buscará inicialmente cuando ejecute el proyecto. Al final, le pedirá que verifique la información que proporciona. Puede escribir *sí* o modificarlo. Bueno, eso es todo, nuestro archivo *package.json* está listo.

Configuración del servidor Express ejecutar *npm install express @ 4 --save* . Esta es todas las dependencias que necesitamos para este proyecto. Aquí el indicador de guardado es importante,

sin él, el archivo *package.js* no se actualizará. La tarea principal de *package.json* es almacenar la lista de dependencias. Agregará la versión expresa 4. Su *package.json* se verá como
`"dependencies": { "express": "^4.13.4", ..... },`

Después de la descarga completa, puede ver que hay una carpeta y subcarpeta de *node_modules* de nuestras dependencias. Ahora en la raíz del proyecto crea un nuevo archivo *server.js* . Ahora estamos configurando el servidor express. Voy a pasar todo el código y lo explicaré más tarde.

```
var express = require('express');
// Create our app
var app = express();

app.use(express.static('public'));

app.listen(3000, function () {
  console.log('Express server is using port:3000');
});
```

var express = require ('express'); Esto te dará el acceso de toda la API expresa.

var app = express (); Llamaré a express library como función. *app.use ();* Deja que la agregue la funcionalidad a tu aplicación express. *app.use (express.static ('public'));* Especificará el nombre de la carpeta que será expuesta en nuestro servidor web. *app.listen (puerto, función () {})* aquí será nuestro puerto *3000* y la función a la que llamamos verificará que el servidor web esté funcionando correctamente. Eso es todo lo que el servidor Express está configurado.

Ahora vaya a nuestro proyecto y cree una nueva carpeta pública y cree el archivo *index.html* . *index.html* es el archivo predeterminado para su aplicación y Express Server buscará este archivo. El *index.html* es un archivo html simple que parece

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="UTF-8"/>
</head>

<body>
  <h1>hello World</h1>
</body>

</html>
```

Y vaya a la ruta del proyecto a través del terminal y escriba *node server.js* . Luego verá * console.log ('El servidor Express está usando el puerto: 3000'); *.

Vaya al navegador y escriba *http: // localhost: 3000* en la barra de navegación, verá *Hola Mundo* .

Ahora ve dentro de la carpeta pública y crea un nuevo archivo *app.jsx* . JSX es un paso de preprocesador que agrega sintaxis XML a tu JavaScript. Definitivamente puedes usar React sin JSX, pero JSX hace que React sea mucho más elegante. Aquí está el código de ejemplo para *app.jsx*

```
ReactDOM.render(
  <h1>Hello World!!!</h1>,
  document.getElementById('app')
);
```

Ahora ve a *index.html* y modifica el código, debería tener este aspecto

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="UTF-8"/>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23
/browser.min.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.7/react.js">
</script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.7/react-dom.js">    </script>
</head>

<body>
  <div id="app"></div>

  <script type="text/babel" src="app.jsx"></script>
</body>

</html>
```

Con esto en su lugar, todo está listo, espero que lo encuentre simple.

Lea React Boilerplate [React + Babel + Webpack] en línea:
https://riptutorial.com/es/reactjs/topic/5969/react-boilerplate--react-plus-babel-plus-webpack-

Capítulo 24: React Component Lifecycle

Introducción

Los métodos de ciclo de vida deben usarse para ejecutar código e interactuar con su componente en diferentes puntos de la vida de los componentes. Estos métodos se basan en un componente de montaje, actualización y desmontaje.

Examples

Creación de componentes

Cuando se crea un componente React, se llama una serie de funciones:

- Si está utilizando `React.createClass` (ES5), se llaman 5 funciones definidas por el usuario
- Si está utilizando la `class Component extends React.Component` (ES6), se llaman 3 funciones definidas por el usuario

`getDefaultProps()` (solo ES5)

Este es el **primer** método llamado.

Los valores de prop devueltos por esta función se utilizarán como valores predeterminados si no se definen cuando se crea una instancia del componente.

En el siguiente ejemplo, `this.props.name` se predeterminará a `Bob` si no se especifica lo contrario:

```
getDefaultProps() {
  return {
    initialCount: 0,
    name: 'Bob'
  };
}
```

`getInitialState()` (solo ES5)

Este es el **segundo** método llamado.

El valor de retorno de `getInitialState()` define el estado inicial del componente React. El marco React llamará a esta función y asignará el valor de retorno a `this.state` .

En el siguiente ejemplo, `this.state.count` se inicializará con el valor de `this.props.initialCount` :

```
getInitialState() {
```

```
  return {
    count : this.props.initialCount
  };
}
```

`componentWillMount()` (ES5 y ES6)

Este es el **tercer** método llamado.

Esta función se puede utilizar para realizar cambios finales en el componente antes de que se agregue al DOM.

```
componentWillMount() {
  ...
}
```

`render()` (ES5 y ES6)

Este es el **cuarto** método llamado.

La función `render()` debe ser una función pura del estado y los elementos del componente. Devuelve un único elemento que representa el componente durante el proceso de representación y debe ser una representación de un componente DOM nativo (por ejemplo, `<p />`) o un componente compuesto. Si no se debe representar nada, puede devolver `null` o `undefined` .

Esta función se recuperará después de cualquier cambio en las propiedades o el estado del componente.

```
render() {
  return (
    <div>
      Hello, {this.props.name}!
    </div>
  );
}
```

`componentDidMount()` (ES5 y ES6)

Este es el **quinto** método llamado.

El componente se ha montado y ahora puede acceder a los nodos DOM del componente, por ejemplo, a través de `refs` .

Este método debe ser utilizado para:

- Preparando temporizadores

- Recuperacion de datos
- Añadiendo oyentes de eventos
- Manipulando elementos DOM

```
componentDidMount() {
  ...
}
```

Sintaxis de ES6

Si el componente se define utilizando la sintaxis de clase ES6, las funciones `getDefaultProps()` y `getInitialState()` no se pueden usar.

En su lugar, declaramos nuestros `defaultProps` como una propiedad estática en la clase y declaramos la forma de estado y el estado inicial en el constructor de nuestra clase. Ambos se establecen en la instancia de la clase en el momento de la construcción, antes de que se llame a cualquier otra función de ciclo de vida React.

El siguiente ejemplo demuestra este enfoque alternativo:

```
class MyReactClass extends React.Component {
  constructor(props){
    super(props);

    this.state = {
      count: this.props.initialCount
    };
  }

  upCount() {
    this.setState((prevState) => ({
      count: prevState.count + 1
    }));
  }

  render() {
    return (
      <div>
        Hello, {this.props.name}!<br />
        You clicked the button {this.state.count} times.<br />
        <button onClick={this.upCount}>Click here!</button>
      </div>
    );
  }
}

MyReactClass.defaultProps = {
  name: 'Bob',
  initialCount: 0
};
```

Reemplazo de `getDefaultProps()`

Los valores predeterminados para las propiedades del componente se especifican al establecer la propiedad `defaultProps` de la clase:

```
MyReactClass.defaultProps = {
  name: 'Bob',
  initialCount: 0
};
```

Sustitución de `getInitialState()`

La forma idiomática de configurar el estado inicial del componente es establecer `this.state` en el constructor:

```
constructor(props){
  super(props);

  this.state = {
    count: this.props.initialCount
  };
}
```

Actualización de componentes

`componentWillReceiveProps(nextProps)`

Esta es la **primera función llamada sobre cambios de propiedades** .

Cuando **las propiedades del componente cambian** , React llamará a esta función con las **nuevas propiedades** . Puede acceder a los accesorios antiguos con *this.props* y a los nuevos accesorios con *nextProps* .

Con estas variables, puede realizar algunas operaciones de comparación entre accesorios antiguos y nuevos, o llamar a la función debido a un cambio de propiedad, etc.

```
componentWillReceiveProps(nextProps){
  if (nextProps.initialCount && nextProps.initialCount > this.state.count){
    this.setState({
      count : nextProps.initialCount
    });
  }
}
```

`shouldComponentUpdate(nextProps, nextState)`

Esta es la **segunda función llamada sobre cambios de propiedades y la primera sobre cambios de estado** .

De forma predeterminada, si otro componente / su componente cambia una propiedad / un estado de su componente, **React** generará una nueva versión de su componente. En este caso, esta función siempre devuelve true.

Puede anular esta función y **elegir con mayor precisión si su componente debe actualizarse o no** .

Esta función se utiliza principalmente para la **optimización** .

En caso de que la función devuelva **falso** , la **actualización de la tubería se detiene de inmediato** .

```
componentShouldUpdate(nextProps, nextState){
  return this.props.name !== nextProps.name ||
    this.state.count !== nextState.count;
}
```

componentWillUpdate(nextProps, nextState)

Esta función funciona como `componentWillMount()` . **Los cambios no están en DOM** , por lo que puede hacer algunos cambios justo antes de que se realice la actualización.

/! \: no puede usar **this.setState ()** .

```
componentWillUpdate(nextProps, nextState){}
```

render()

Hay algunos cambios, así que vuelva a renderizar el componente.

componentDidUpdate(prevProps, prevState)

Lo mismo que `componentDidMount()` : **DOM se actualiza** , por lo que puede hacer algún trabajo en el DOM aquí.

```
componentDidUpdate(prevProps, prevState){}
```

Eliminación de componentes

componentWillUnmount()

Se llama a este método **antes de** que se desmonte un componente del DOM.

Es un buen lugar para realizar operaciones de limpieza como:

- Eliminando oyentes de eventos.
- Temporizadores de compensación.
- Detener los enchufes.
- Limpieza de estados redux.

```
componentWillUnmount(){
  ...
```

```
}
```

Un ejemplo de eliminación de la escucha de eventos adjunta en `componentWillUnMount`

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

export default class SideMenu extends Component {

  constructor(props) {
    super(props);
    this.state = {
        ...
      };
    this.openMenu = this.openMenu.bind(this);
    this.closeMenu = this.closeMenu.bind(this);
  }

  componentDidMount() {
    document.addEventListener("click", this.closeMenu);
  }

  componentWillUnmount() {
    document.removeEventListener("click", this.closeMenu);
  }

  openMenu() {
    ...
  }

  closeMenu() {
    ...
  }

  render() {
    return (
      <div>
        <a
          href      = "javascript:void(0)"
          className = "closebtn"
          onClick   = {this.closeMenu}
        >
          x
        </a>
        <div>
          Some other structure
        </div>
      </div>
    );
  }
}
```

React Component Container

Al crear una aplicación React, a menudo es conveniente dividir los componentes en función de su responsabilidad principal, en los componentes de Presentación y Contenedor.
Los componentes de presentación solo se ocupan de mostrar datos: pueden considerarse como, y con frecuencia, se implementan como funciones que convierten un modelo en una vista.

Típicamente no mantienen ningún estado interno. Los componentes del contenedor se ocupan de la gestión de datos. Esto puede hacerse internamente a través de su propio estado o actuando como intermediarios con una biblioteca de administración del estado como Redux. El componente contenedor no mostrará directamente los datos, sino que pasará los datos a un componente de presentación.

```
// Container component
import React, { Component } from 'react';
import Api from 'path/to/api';

class CommentsListContainer extends Component {
    constructor() {
        super();
        // Set initial state
        this.state = { comments: [] }
    }

    componentDidMount() {
        // Make API call and update state with returned comments
        Api.getComments().then(comments => this.setState({ comments }));
    }

    render() {
        // Pass our state comments to the presentational component
        return (
            <CommentsList comments={this.state.comments} />;
        );
    }
}

// Presentational Component
const CommentsList = ({ comments }) => (
    <div>
        {comments.map(comment => (
            <div>{comment}</div>
        )}
    </div>
);

CommentsList.propTypes = {
    comments: React.PropTypes.arrayOf(React.PropTypes.string)
}
```

Método de ciclo de vida llamado en diferentes estados.

Este ejemplo sirve como complemento a otros ejemplos que hablan sobre cómo usar los métodos del ciclo de vida y cuándo se llamará el método.

Este ejemplo resume qué métodos (componentWillMount, componentWillReceiveProps, etc.) se llamarán y en qué secuencia será diferente para un componente **en diferentes estados** :

Cuando se inicializa un componente:

1. getDefaultProps
2. getInitialState
3. componentWillMount

4. hacer
5. componentDidMount

Cuando un componente ha cambiado de estado:

1. shouldComponentUpdate
2. componentWillUpdate
3. hacer
4. componentDidUpdate

Cuando un componente tiene accesorios cambiados:

1. componentWillReceiveProps
2. shouldComponentUpdate
3. componentWillUpdate
4. hacer
5. componentDidUpdate

Cuando se desmonta un componente:

1. componentWillUnmount

Lea React Component Lifecycle en línea: https://riptutorial.com/es/reactjs/topic/2750/react-component-lifecycle

Capítulo 25: React.createClass vs extiende React.Component

Sintaxis

- Caso 1: React.createClass ({})
- Caso 2: la clase MyComponent extiende React.Component {}

Observaciones

`React.createClass` quedó en desuso en v15.5 y se espera que se elimine en v16 . Hay un paquete de reemplazo para aquellos que todavía lo requieren. Los ejemplos que lo usan deben ser actualizados.

Examples

Crear Componente React

Exploremos las diferencias de sintaxis comparando dos ejemplos de código.

React.createClass (en desuso)

Aquí tenemos una `const` con una clase React asignada, con la función de `render` continuación para completar una definición de componente base típica.

```
import React from 'react';

const MyComponent = React.createClass({
  render() {
    return (
      <div></div>
    );
  }
});

export default MyComponent;
```

React.Component

Tomemos la definición anterior de React.createClass y la convertimos para usar una clase ES6.

```
import React from 'react';
```

```
class MyComponent extends React.Component {
  render() {
    return (
      <div></div>
    );
  }
}

export default MyComponent;
```

En este ejemplo ahora estamos usando clases de ES6. Para los cambios de React, ahora creamos una clase llamada **MyComponent** y la extendemos desde React.Component en lugar de acceder a React.createClass directamente. De esta manera, usamos menos repetitivo y más JavaScript.

PS: Normalmente, esto se usaría con algo como Babel para compilar el ES6 a ES5 para que funcione en otros navegadores.

Declare Props y PropTypes por defecto

Hay cambios importantes en la forma en que usamos y declaramos los accesorios predeterminados y sus tipos.

React.createClass

En esta versión, la propiedad `propTypes` es un Objeto en el que podemos declarar el tipo para cada prop. La propiedad `getDefaultProps` es una función que devuelve un objeto para crear los accesorios iniciales.

```
import React from 'react';

const MyComponent = React.createClass({
  propTypes: {
    name: React.PropTypes.string,
    position: React.PropTypes.number
  },
  getDefaultProps() {
    return {
      name: 'Home',
      position: 1
    };
  },
  render() {
    return (
      <div></div>
    );
  }
});

export default MyComponent;
```

React.Component

Esta versión utiliza `propTypes` como una propiedad en la clase **MyComponent** real en lugar de una propiedad como parte del objeto de definición `createClass` .

`getDefaultProps` ahora ha cambiado a solo una propiedad Object en la clase llamada defaultProps, ya que ya no es una función "obtener", es solo un objeto. Evita más repeticiones de React, esto es simplemente JavaScript.

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div></div>
    );
  }
}
MyComponent.propTypes = {
  name: React.PropTypes.string,
  position: React.PropTypes.number
};
MyComponent.defaultProps = {
  name: 'Home',
  position: 1
};

export default MyComponent;
```

Además, hay otra sintaxis para `propTypes` y `defaultProps` . Este es un acceso directo si su compilación tiene activados los inicializadores de propiedades de ES7:

```
import React from 'react';

class MyComponent extends React.Component {
  static propTypes = {
    name: React.PropTypes.string,
    position: React.PropTypes.number
  };
  static defaultProps = {
    name: 'Home',
    position: 1
  };
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div></div>
    );
  }
}
```

```
export default MyComponent;
```

Establecer estado inicial

Hay cambios en cómo estamos estableciendo los estados iniciales.

React.createClass

Tenemos una función `getInitialState` , que simplemente devuelve un objeto de estados iniciales.

```
import React from 'react';

const MyComponent = React.createClass({
  getInitialState () {
    return {
      activePage: 1
    };
  },
  render() {
    return (
      <div></div>
    );
  }
});

export default MyComponent;
```

React.Component

En esta versión, declaramos todo el estado como una **propiedad de inicialización** simple **en el constructor** , en lugar de usar la función `getInitialState` . Se siente menos impulsado por "React API" ya que esto es simplemente JavaScript.

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      activePage: 1
    };
  }
  render() {
    return (
      <div></div>
    );
  }
}

export default MyComponent;
```

Mixins

Podemos usar `mixins` solo con la forma React.createClass.

React.createClass

En esta versión, podemos agregar `mixins` a los componentes utilizando la propiedad mixins que toma una matriz de mixins disponibles. Estos luego extienden la clase de componente.

```
import React from 'react';

var MyMixin = {
  doSomething() {

  }
};
const MyComponent = React.createClass({
  mixins: [MyMixin],
  handleClick() {
    this.doSomething(); // invoke mixin's method
  },
  render() {
    return (
      <button onClick={this.handleClick}>Do Something</button>
    );
  }
});

export default MyComponent;
```

React.Component

Los mixins de React no son compatibles cuando se usan componentes React escritos en ES6. Además, no tendrán soporte para las clases de ES6 en React. La razón es que se consideran perjudiciales .

"este contexto

El uso de React.createClass vinculará automáticamente `this` contexto (valores) correctamente, pero ese no es el caso cuando se usan clases de ES6.

React.createClass

Tenga en cuenta la declaración `onClick` con el método `this.handleClick` enlazado. Cuando se llame a este método, React aplicará el contexto de ejecución correcto al `handleClick` .

```
import React from 'react';
```

```
const MyComponent = React.createClass({
  handleClick() {
    console.log(this); // the React Component instance
  },
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
});

export default MyComponent;
```

React.Component

Con las clases de ES6 `this` es `null` por defecto, las propiedades de la clase no se unen automáticamente a la instancia de la clase React (componente).

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  handleClick() {
    console.log(this); // null
  }
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default MyComponent;
```

Hay algunas formas en que podríamos atar el derecho a `this` contexto.

Caso 1: Enlace en línea:

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  handleClick() {
    console.log(this); // the React Component instance
  }
  render() {
    return (
      <div onClick={this.handleClick.bind(this)}></div>
```

```
    );
  }
}

export default MyComponent;
```

Caso 2: Enlace en el constructor de la clase.

Otro enfoque es cambiar el contexto de `this.handleClick` dentro del `constructor` . De esta manera evitamos la repetición en línea. Considerado por muchos como un mejor enfoque que evita tocar JSX en absoluto:

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    console.log(this); // the React Component instance
  }
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default MyComponent;
```

Caso 3: utilizar la función anónima de ES6

También puede utilizar la función anónima de ES6 sin tener que enlazar explícitamente:

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  handleClick = () => {
    console.log(this); // the React Component instance
  }
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default MyComponent;
```

ES6 / Reacciona la palabra clave "this" con ajax para obtener datos del

servidor

```
import React from 'react';

class SearchEs6 extends React.Component{
    constructor(props) {
        super(props);
        this.state = {
            searchResults: []
        };
    }

    showResults(response){
        this.setState({
            searchResults: response.results
        })
    }

    search(url){
        $.ajax({
            type: "GET",
            dataType: 'jsonp',
            url: url,
            success: (data) => {
                this.showResults(data);
            },
            error: (xhr, status, err) => {
                console.error(url, status, err.toString());
            }
        });
    }

    render() {
        return (
            <div>
                <SearchBox search={this.search.bind(this)} />
                <Results searchResults={this.state.searchResults} />
            </div>
        );
    }
}
```

Lea React.createClass vs extiende React.Component en línea:
https://riptutorial.com/es/reactjs/topic/6371/react-createclass-vs-extiende-react-component

Capítulo 26: Soluciones de interfaz de usuario

Introducción

Digamos que nos inspiramos en algunas ideas de las interfaces de usuario modernas utilizadas en los programas y las convertimos en componentes React. En eso consiste el tema " **Soluciones de interfaz de usuario** ". Se atribuye la atribución.

Examples

Panel básico

```
import React from 'react';

class Pane extends React.Component {
    constructor(props) {
        super(props);
    }

    render() {
        return React.createElement(
            'section', this.props
        );
    }
}
```

Panel

```
import React from 'react';

class Panel extends React.Component {
    constructor(props) {
        super(props);
    }

    render(...elements) {
        var props = Object.assign({
            className: this.props.active ? 'active' : '',
            tabIndex: -1
        }, this.props);

        var css = this.css();
        if (css != '') {
            elements.unshift(React.createElement(
                'style', null,
                css
            ));
        }

        return React.createElement(
```

```
            'div', props,
            ...elements
        );
    }

    static title() {
        return '';
    }
    static css() {
        return '';
    }
}
```

Las principales diferencias con respecto al panel simple son:

- el panel tiene el foco, por ejemplo, cuando se llama mediante un script o se hace clic con el mouse;
- el panel tiene un método de `title` estático por componente, por lo que puede ser extendido por otro componente del panel con un `title` anulado (la razón aquí es que la función puede ser llamada nuevamente en la representación para fines de localización, pero en los límites de este `title` ejemplo no tiene sentido) ;
- puede contener una hoja de estilo individual declarada en el método estático `css` (puede cargar previamente el contenido del archivo desde `PANEL.css`).

Lengüeta

```
import React from 'react';

class Tab extends React.Component {
    constructor(props) {
        super(props);
    }

    render() {
        var props = Object.assign({
            className: this.props.active ? 'active' : ''
        }, this.props);
        return React.createElement(
            'li', props,
            React.createElement(
                'span', props,
                props.panelClass.title()
            )
        );
    }
}
```

`panelClass` propiedad `panelClass` de la instancia de `Tab` debe contener la clase de *panel* utilizada para la descripción.

Grupo de paneles

```
import React from 'react';
import Tab from './Tab.js';
```

```
class PanelGroup extends React.Component {
    constructor(props) {
        super(props);
        this.setState({
            panels: props.panels
        });
    }

    render() {
        this.tabSet = [];
        this.panelSet = [];
        for (let panelData of this.state.panels) {
            var tabIsActive = this.state.activeTab == panelData.name;
            this.tabSet.push(React.createElement(
                Tab, {
                    name: panelData.name,
                    active: tabIsActive,
                    panelClass: panelData.class,
                    onMouseDown: () => this.openTab(panelData.name)
                }
            ));
            this.panelSet.push(React.createElement(
                panelData.class, {
                    id: panelData.name,
                    active: tabIsActive,
                    ref: tabIsActive ? 'activePanel' : null
                }
            ));
        }
        return React.createElement(
            'div', { className: 'PanelGroup' },
            React.createElement(
                'nav', null,
                React.createElement(
                    'ul', null,
                    ...this.tabSet
                )
            ),
            ...this.panelSet
        );
    }

    openTab(name) {
        this.setState({ activeTab: name });
        this.findDOMNode(this.refs.activePanel).focus();
    }
}
```

`panels` propiedad de `panels` de la instancia de `PanelGroup` debe contener una matriz con objetos. Cada objeto allí declara datos importantes sobre los paneles:

- **`name`** : identificador del panel utilizado por el script del controlador;
- **`class`** - **`class`** panel.

No olvide configurar la propiedad `activeTab` al nombre de la pestaña necesaria.

Aclaración

Cuando se pestaña hacia abajo, el panel se necesita es cada nombre de la clase `active` el elemento DOM (que significa que va a ser visible) y se centra ahora.

Vista de ejemplo con `PanelGroup`s

```
import React from 'react';
import Pane from './components/Pane.js';
import Panel from './components/Panel.js';
import PanelGroup from './components/PanelGroup.js';

class MainView extends React.Component {
    constructor(props) {
        super(props);
    }

    render() {
        return React.createElement(
            'main', null,
            React.createElement(
                Pane, { id: 'common' },
                React.createElement(
                    PanelGroup, {
                        panels: [
                            {
                                name: 'console',
                                panelClass: ConsolePanel
                            },
                            {
                                name: 'figures',
                                panelClass: FiguresPanel
                            }
                        ],
                        activeTab: 'console'
                    }
                )
            ),
            React.createElement(
                Pane, { id: 'side' },
                React.createElement(
                    PanelGroup, {
                        panels: [
                            {
                                name: 'properties',
                                panelClass: PropertiesPanel
                            }
                        ],
                        activeTab: 'properties'
                    }
                )
            )
        );
    }
}

class ConsolePanel extends Panel {
    constructor(props) {
        super(props);
    }
```

```
    static title() {
        return 'Console';
    }
}

class FiguresPanel extends Panel {
    constructor(props) {
        super(props);
    }

    static title() {
        return 'Figures';
    }
}

class PropertiesPanel extends Panel {
    constructor(props) {
        super(props);
    }

    static title() {
        return 'Properties';
    }
}
```

Lea Soluciones de interfaz de usuario en línea:
https://riptutorial.com/es/reactjs/topic/8112/soluciones-de-interfaz-de-usuario

Capítulo 27: Teclas en reaccionar

Introducción

Las claves en reaccionar se utilizan para identificar una lista de elementos DOM de la misma jerarquía internamente.

Entonces, si está iterando sobre una matriz para mostrar una lista de elementos li, cada uno de los elementos li necesita un identificador único especificado por la propiedad clave. Por lo general, este puede ser el ID de su elemento de base de datos o el índice de la matriz.

Observaciones

Por lo general, no se recomienda usar el índice de matriz como clave cuando la matriz va a cambiar con el tiempo. De los documentos de React:

> Como último recurso, puede pasar el índice del elemento en la matriz como una clave. Esto puede funcionar bien si los artículos nunca se reordenan, pero los pedidos serán lentos.

Un buen ejemplo de esto: https://medium.com/@robinpokorny/index-as-a-key-is-an-anti-pattern-e0349aece318

Examples

Usando el id de un elemento

Aquí tenemos una lista de elementos pendientes que se pasan a los accesorios de nuestro componente.

Cada elemento tiene una propiedad de texto e id. Imagine que la propiedad id proviene de un almacén de datos de back-end y es un valor numérico único:

```
todos = [
  {
    id: 1,
    text: 'value 1'
  },
  {
    id: 2,
    text: 'value 2'
  },
  {
    id: 3,
    text: 'value 3'
  },
  {
    id: 4,
```

```
    text: 'value 4'
  },
];
```

Establecemos el atributo clave de cada elemento de la lista iterada en `todo-${todo.id}` para que reaccione puede identificarlo internamente:

```
render() {
  const { todos } = this.props;
  return (
    <ul>
      { todos.map((todo) =>
        <li key={ `todo-${todo.id}` }>
          { todo.text }
        </li>
      }
    </ul>
  );
}
```

Usando el índice de matriz

Si no tiene identificadores de base de datos únicos a la mano, también podría usar el índice numérico de su matriz de la siguiente manera:

```
render() {
  const { todos } = this.props;
  return (
    <ul>
      { todos.map((todo, index) =>
        <li key={ `todo-${index}` }>
          { todo.text }
        </li>
      }
    </ul>
  );
}
```

Lea Teclas en reaccionar en línea: https://riptutorial.com/es/reactjs/topic/9805/teclas-en-reaccionar

Capítulo 28: Usando React con Flujo

Introducción

Cómo utilizar el comprobador de tipo de flujo para verificar los tipos en los componentes React.

Observaciones

Flujo | Reaccionar

Examples

Uso de Flow para verificar tipos de accesorios de componentes funcionales sin estado

```
type Props = {
  posts: Array<Article>,
  dispatch: Function,
  children: ReactElement
}

const AppContainer =
  ({ posts, dispatch, children }: Props) => (
    <div className="main-app">
      <Header {...{ posts, dispatch }} />
      {children}
    </div>
  )
```

Usando Flow para verificar los tipos de accesorios

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

type Props = {
  posts: Array<Article>,
  dispatch: Function,
  children: ReactElement
}

class Posts extends Component {
  props: Props;

  render () {
    // rest of the code goes here
  }
}
```

Lea Usando React con Flujo en línea: https://riptutorial.com/es/reactjs/topic/7918/usando-react-con-flujo

Capítulo 29: Usando ReactJS con jQuery

Examples

Reacciona con jQuery

En primer lugar, tienes que importar la librería jquery. También necesitamos importar findDOmNode ya que vamos a manipular el dominio. Y obviamente también estamos importando React.

```
import React from 'react';
import { findDOMNode } from 'react-dom';
import $ from 'jquery';
```

Estamos configurando una función de flecha 'handleToggle' que se activará cuando se haga clic en un icono. Solo mostramos y escondemos un div con un nombre de referencia 'alternar' enHaga clic sobre un ícono.

```
handleToggle = () => {
    const el = findDOMNode(this.refs.toggle);
    $(el).slideToggle();
};
```

Ahora vamos a establecer el nombre de referencia 'toggle'

```
<ul className="profile-info additional-profile-info-list" ref="toggle">
  <li>
    <span className="info-email">Office Email</span>   me@shuvohabib.com
  </li>
</ul>
```

El elemento div donde dispararemos el 'handleToggle' en onClick.

```
 <div className="ellipsis-click" onClick={this.handleToggle}>
   <i className="fa-ellipsis-h"/>
 </div>
```

Vamos a revisar el código completo a continuación, cómo se ve.

```
import React from 'react';
import { findDOMNode } from 'react-dom';
import $ from 'jquery';

export default class FullDesc extends React.Component {
    constructor() {
        super();
    }

    handleToggle = () => {
        const el = findDOMNode(this.refs.toggle);
```

```
        $(el).slideToggle();
    };

    render() {
        return (
            <div className="long-desc">
                <ul className="profile-info">
                    <li>
                        <span className="info-title">User Name : </span> Shuvo Habib
                    </li>
                </ul>

                <ul className="profile-info additional-profile-info-list" ref="toggle">
                    <li>
                        <span className="info-email">Office Email</span> me@shuvohabib.com
                    </li>
                </ul>

                <div className="ellipsis-click" onClick={this.handleToggle}>
                    <i className="fa-ellipsis-h"/>
                </div>
            </div>
        );
    }
}
```

¡Hemos terminado! Así es, cómo podemos usar **jQuery en el** componente **React** .

Lea Usando ReactJS con jQuery en línea: https://riptutorial.com/es/reactjs/topic/6009/usando-reactjs-con-jquery

Capítulo 30: Usando ReactJS en forma de flujo

Introducción

Resulta muy útil utilizar el enfoque de Flux, cuando se planea que crezca su aplicación con ReactJS en el frontend, debido a estructuras limitadas y un poco de código nuevo para hacer que los cambios de estado en el tiempo de ejecución sean más fáciles.

Observaciones

Flux es la arquitectura de la aplicación que usa Facebook para crear aplicaciones web del lado del cliente. Complementa los componentes de vista compostable de React utilizando un flujo de datos unidireccional. Es más un patrón que un marco formal, y puede comenzar a usar Flux inmediatamente sin un montón de código nuevo.

Las aplicaciones de flujo tienen tres partes principales: *el despachador* , *las tiendas* y *las vistas* (componentes React). Estos no deben confundirse con Model-View-Controller. Los controladores existen en una aplicación Flux, pero son vistas de controlador, vistas que a menudo se encuentran en la parte superior de la jerarquía que recuperan datos de las tiendas y pasan estos datos a sus hijos. Además, los creadores de acciones (métodos de ayuda de dispatcher) se utilizan para admitir una API semántica que describe todos los cambios posibles en la aplicación. Puede ser útil pensar en ellos como una cuarta parte del ciclo de actualización de Flux.

Flux evita MVC en favor de un flujo de datos unidireccional. Cuando un usuario interactúa con una vista React, la vista propaga una acción a través de un despachador central, a las diferentes tiendas que contienen los datos y la lógica empresarial de la aplicación, que actualiza todas las vistas afectadas. Esto funciona especialmente bien con el estilo de programación declarativo de React, que permite a la tienda enviar actualizaciones sin especificar cómo hacer la transición de las vistas entre los estados.

Examples

Flujo de datos

> Este es un resumen de la descripción general .

El patrón de flujo asume el uso de flujo de datos unidireccional.

1. **Acción** : objeto simple que describe el `type` acción y otros datos de entrada.
2. **Dispatcher** - controlador de una sola acción y devoluciones de llamada. Imagina que es el eje central de tu aplicación.

3. **Tienda** : contiene el estado y la lógica de la aplicación. Registra la devolución de llamada en el despachador y emite un evento para ver cuando se ha producido un cambio en la capa de datos.

4. **Ver** - Reactivar componente que recibe eventos de cambio y datos de la tienda. Provoca una nueva representación cuando se cambia algo.

    > A partir del flujo de datos de Flux, las vistas también pueden **crear acciones** y pasarlas al distribuidor para las interacciones del usuario.

Revertido

Para hacerlo más claro, podemos empezar desde el final.

- Los diferentes componentes de React (*vistas*) obtienen datos de diferentes tiendas sobre los cambios realizados.

    > Pocos componentes pueden denominarse **vistas de controlador** , ya que proporcionan el código de pegamento para obtener los datos de las tiendas y pasar los datos por la cadena de sus descendientes. Las vistas de controlador representan cualquier sección significativa de la página.

- *Las tiendas se* pueden destacar como devoluciones de llamada que comparan el tipo de acción y otros datos de entrada para la lógica empresarial de su aplicación.

- *Dispatcher* es el receptor de acciones comunes y el contenedor de devoluciones de llamada.

- *Las acciones* no son más que simples objetos con propiedad de `type` requerida.

    > Anteriormente, querrá usar constantes para los tipos de acción y los métodos de ayuda (llamados **creadores de acción**).

Lea Usando ReactJS en forma de flujo en línea:
https://riptutorial.com/es/reactjs/topic/8158/usando-reactjs-en-forma-de-flujo

Capítulo 31: Utilizando ReactJS con Typescript

Examples

Componente ReactJS escrito en Typescript

En realidad, puedes usar los componentes de ReactJS en Typescript como en el ejemplo de facebook. Simplemente reemplaza la extensión del archivo 'jsx' a 'tsx':

```
//helloMessage.tsx:
var HelloMessage = React.createClass({
  render: function() {
    return <div>Hello {this.props.name}</div>;
  }
});
ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

Pero para hacer un uso completo de la característica principal de Typescript (verificación de tipos estática) se debe hacer un par de cosas:

1) convertir el ejemplo de React.createClass a ES6 Class:

```
//helloMessage.tsx:
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

2) a continuación agregue Props y interfaces de estado:

```
interface IHelloMessageProps {
    name:string;
}

interface IHelloMessageState {
  //empty in our case
}

class HelloMessage extends React.Component<IHelloMessageProps, IHelloMessageState> {
  constructor(){
    super();
  }
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
ReactDOM.render(<HelloMessage name="Sebastian" />, mountNode);
```

Ahora Typescript mostrará un error si el programador se olvida de aprobar las propuestas. O si añadieron accesorios que no están definidos en la interfaz.

Componentes de Stateless React en Typescript

Los componentes de React que son funciones puras de sus propiedades y no requieren ningún estado interno se pueden escribir como funciones de JavaScript en lugar de usar la sintaxis de clase estándar, como:

```
import React from 'react'

const HelloWorld = (props) => (
    <h1>Hello, {props.name}!</h1>
);
```

Lo mismo se puede lograr en Typescript usando la clase `React.SFC` :

```
import * as React from 'react';

class GreeterProps {
   name: string
}

const Greeter : React.SFC<GreeterProps> = props =>
    <h1>Hello, {props.name}!</h1>;
```

Tenga en cuenta que, el nombre `React.SFC` es un alias para `React.StatelessComponent` Por `React.StatelessComponent` tanto, cualquiera de los dos puede usarse.

Instalación y configuración

Para usar mecanografiado con reacción en un proyecto de nodo, primero debe tener un directorio de proyecto inicializado con npm. Inicializar el directorio con `npm init`

Instalación vía npm o hilo

Puedes instalar React usando npm haciendo lo siguiente:

```
npm install --save react react-dom
```

Facebook lanzó su propio administrador de paquetes llamado Yarn , que también se puede usar para instalar React. Después de instalar Yarn solo necesitas ejecutar este comando:

```
yarn add react react-dom
```

Luego puede usar React en su proyecto exactamente de la misma manera que si hubiera instalado React a través de npm.

Instalando definiciones de tipo de reacción en Typescript 2.0+

Para compilar su código usando mecanografia, agregue / instale archivos de definición de tipo usando npm o hilados.

```
npm install --save-dev @types/react @types/react-dom
```

o, usando hilo

```
yarn add --dev @types/react @types/react-dom
```

Instalar definiciones de tipo de reacción en versiones anteriores de Typescript

Tienes que usar un paquete separado llamado tsd

```
tsd install react react-dom --save
```

Agregando o cambiando la configuración de Typescript

Para usar JSX , un lenguaje que mezcla javascript con html / xml, debe cambiar la configuración del compilador de caracteres. En el archivo de configuración de `tsconfig.json` del proyecto (normalmente denominado `tsconfig.json`), deberá agregar la opción JSX como:

```
"compilerOptions": {
    "jsx": "react"
},
```

Esa opción del compilador básicamente le dice al compilador mecanografiado que traduzca las etiquetas JSX en código a las llamadas de función javascript.

Para evitar que el compilador mecanografiado convierta JSX a llamadas de funciones de JavaScript simples, use

```
"compilerOptions": {
    "jsx": "preserve"
},
```

Componentes sin estado y sin propiedad

El componente de reacción más simple sin un estado y sin propiedades se puede escribir como:

```
import * as React from 'react';

const Greeter = () => <span>Hello, World!</span>
```

Ese componente, sin embargo, no puede acceder a `this.props` ya que mecanografiado no puede saber si es un componente de reacción. Para acceder a sus accesorios, utiliza:

```
import * as React from 'react';

const Greeter: React.SFC<{}> = props => () => <span>Hello, World!</span>
```

Incluso si el componente no tiene propiedades explícitamente definidas, ahora puede acceder a **`props.children`** ya que todos los componentes tienen hijos de forma inherente.

Otro buen uso similar de los componentes sin estado y sin propiedades se encuentra en la plantilla de página simple. El siguiente es un componente simple de la `Page` ejemplos, asumiendo que hay componentes hipotéticos de `Container` , `NavTop` y `NavBottom` ya en el proyecto:

```
import * as React from 'react';

const Page: React.SFC<{}> = props => () =>
    <Container>
        <NavTop />
        {props.children}
        <NavBottom />
    </Container>

const LoginPage: React.SFC<{}> = props => () =>
    <Page>
        Login Pass: <input type="password" />
    </Page>
```

En este ejemplo, el componente `Page` se puede usar posteriormente por cualquier otra página real como plantilla base.

Lea Utilizando ReactJS con Typescript en línea:
https://riptutorial.com/es/reactjs/topic/1419/utilizando-reactjs-con-typescript

Creditos

S. No	Capítulos	Contributors
1	Empezando con React	Adam, Adrián Daraš, Alex, Alex Young, Anuj, Bart Riordan, Cassidy, Community, Daksh Gupta, Dave Kaye, diabolicfreak, DMan, Donald, Everettss, Gianluca Esposito, himanshuIIITian, hyde, Ilya Lyamkin, Inanc Gumus, ivarni, jengeb, jolyonruss, Jon Chan, JordanHendrix, juandemarco, Kaloyan Kosev, Konstantin Grushetsky, Maksim, Marty, MaxPRafferty, Md. Nahiduzzaman Rose, Md.Sifatul Islam, Ming Soon, MMachinegun, Nick Bartlett, orvi, paqash, Prakash, rossipedia, Shabin Hashim, Simplans, Sunny R Gupta, TheShadowbyte, Timo, Tushar Khanna, user2314737
2	Actuación	Aditya Singh, lustoykov, thibmaek
3	Apoyos en reaccionar	Ahmad, Anuj, Danillo Corvalan, Everettss, Faktor 10, Fellow Stranger, hansn, Ilya Lyamkin, Jack7, Jagadish Upadhyay, JimmyLv, MaxPRafferty, QoP, Sergii Bishyr, vintproykt, WitVault , zbynour
4	Cómo configurar un webpack básico, reaccionar y babel.	Bart Riordan, Tien Do, Zac Braddy
5	Cómo y por qué usar llaves en React.	Sammy I.
6	Componentes	akashrajkn, Anuj, Bart Riordan, Bond, Brandon Roberts, Denis Ivanov, Diego V, DMan, Evan Hammer, Everettss, goldbullet, GordyD, hmnzr, Ilya Lyamkin, ivarni, Jagadish Upadhyay, jbmartinez, John Ruddell, jolyonruss, Jon Chan, jonathangoodman, JordanHendrix, justabuzz, k170, Kousha, Kyle Richardson, m_callens, Maayan Glikser, Michael Peyper, Paul Graffam, philpee2, QoP, Radu Brehar, Sai Vikas, sjmarshy , Timo, Vlad Bezden, WooCaSh, Zakaria Ridouh, zurfyx
7	Componentes de orden superior	Dennis Stücken
8	Componentes funcionales sin estado	Adam, Mark Lapierre, Mayank Shukla, Valter Júnior
9	Comunicación Entre	David, Kaloyan Kosev

	Componentes	
10	Comunicar Entre Componentes	Random User
11	Configuración de React Ambiente	ghostffcode, Tien Do
12	Estado en reaccionar	Alex Young, Alexander, Brad Colthurst, Everettss, Kousha, Kyle Richardson, QoP, skav, Timo
13	Formularios y comentarios del usuario	Everettss, Henrik Karlsson, ivarni, Timo
14	Instalación	Rene R, Ruairi O'Brien
15	Instalación de React, Webpack y Typescript.	Aron
16	Introducción a la representación del lado del servidor	Adrián Daraš, MauroPorrasP
17	JSX	Kaloyan Kosev, Ming Soon
18	Reaccionar con redux	Jim
19	Reaccionar enrutamiento	abhirathore2006, Robeen
20	Reaccionar formas	promisified
21	Reaccionar herramientas	brillout
22	Reaccionar llamada AJAX	adamboro, Fabian Schultz, Jason Bourne, lifeiscontent, McGrady, Sunny R Gupta
23	React Boilerplate [React + Babel + Webpack]	Mihir, parlad neupane, Tien Do
24	React Component Lifecycle	Alex Young, Alexg2195, Anuj, Ashari, Everettss, F. Kauder, irrigator, John Ruddell, QoP, Salman Saleem, Saravana, Siddharth, skav, Timo, ultrasamad, Vivian, WitVault
25	React.createClass vs	Kaloyan Kosev, leonardoborges, Michael Peyper, pwolaq,

	extiende React.Component	Qianyue, sqzaman
26	Soluciones de interfaz de usuario	vintproykt
27	Teclas en reaccionar	Dennis Stücken, thibmaek
28	Usando React con Flujo	JimmyLv, lifeiscontent, Rifat, Rory O'Kane
29	Usando ReactJS con jQuery	Kousha, Shuvo Habib
30	Usando ReactJS en forma de flujo	vintproykt
31	Utilizando ReactJS con Typescript	Everettss, John Ruddell, kevgathuku, Leone, Rajab Shakirov