

# React JS

## Apuntes para Profesionales

### Chapter 3: Using ReactJS with TypeScript

#### Section 3.1: ReactJS component written in TypeScript

Actually you can use ReactJS components in TypeScript as in Facebook's example. Just replace to:

```
//HelloMessage.tsx
var HelloMessage = React.createClass({
  render: function() {
    return <div>Hello {this.props.name}</div>;
  }
});
ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

But in order to make full use of TypeScript's main feature (static type checking) should be:

1) convert React.createClass example to ES6 Class:

```
//HelloMessage.tsx
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

2) next add Props and State interfaces:

```
interface HelloMessageProps {
  name: string;
}
interface HelloMessageState {
  //empty in our case
}
class HelloMessage extends React.Component<HelloMessageProps, HelloMessageState> {
  constructor() {
    super();
  }
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

Now TypeScript will display an error if the programmer forgets to pass props defined in the interface.

#### Section 3.2: Installation and Setup

To use typescript with react in a node project, you must first have a project directory initialized with npm:

install the directory with npm: `npm init`

### Chapter 10: React Routing

#### Section 10.1: Example Routes.js file, followed by use of Router Link in component

Place a file like the following in your top level directory. It defines which components to render for which paths:

```
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>
);
```

Now in your top level index.js that is your entry point to the app, you need only render this Router component like so:

```
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('root')
);
```

Now it is simply a matter of using Link instead of <a> tags throughout your application. Using Link will communicate with React Router to change the React Router route to the specified link, which will in turn render the correct component as defined in routes.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>
  );
}
```

ReactJS Notes for Professionals

ReactJS Notes for Professionals

### Chapter 14: React AJAX call

#### Section 14.1: HTTP GET request

Sometimes a component needs to render some data from a remote endpoint (e.g. a REST API). A [ReactJS tutorial](#) is to make such calls in components using method.

Here is an example, using [superagent](#) as AJAX helper:

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

class App extends React.Component {
  constructor() {
    super();
    this.state = {}
  }
  componentDidMount() {
    request
      .get('/posts')
      .query({ range: 'new' })
      .query({ order: 'desc' })
      .set('Accept', 'application/json')
      .end((err, resp) => {
        if (!err) {
          this.setState({posts: resp.text})
        }
      })
  }
  render() {
    return (
      <div>{this.state.posts} </div>
    );
  }
}
```

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

A request can be initiated by invoking the appropriate method in the request object, then calling .end() to send the request. Setting header fields is simple, invoke .set() with a field name and value.

The .query() method accepts objects, which when used with the GET method will form a query-string. The following will produce the path /search/query=foo&range=1..5&order=desc.

POST requests

```
request.post('/posts')
  .set('Content-Type', 'application/json')
  .send({name: 'foo', 'post': 'bar'})
  .end(callback);
```

See [superagent docs](#) for more details.

ReactJS Notes for Professionals

Traducido por:

**ortegag**

**100+ páginas**

de consejos y trucos profesionales

# Contenidos

<b>Acerca de .....</b>	<b>1</b>
<b>Capítulo 1: Introducción a React .....</b>	<b>2</b>
Sección 1.1: ¿Qué es ReactJS? .....	2
Sección 1.2: Instalación o configuración.....	3
Sección 1.3: Hello World con funciones sin estado.....	4
Sección 1.4: Fundamentos absolutos de la creación de componentes reutilizables .....	5
Sección 1.5: Create React App.....	6
Sección 1.6: Hello World .....	7
Sección 1.7: Componente Hello World.....	8
<b>Capítulo 2: Componentes .....</b>	<b>10</b>
Sección 2.1: Creación de componentes.....	10
Sección 2.2: Componente básico.....	12
Sección 2.3: Componentes anidados .....	13
Sección 2.4: Props.....	16
Sección 2.5: Estados de los componentes - Interfaz de usuario dinámica .....	17
Sección 2.6: Variaciones de los componentes funcionales sin estado .....	18
Sección 2.7: trampas de setState .....	19
<b>Capítulo 3: Uso de ReactJS con TypeScript .....</b>	<b>21</b>
Sección 3.1: Componente ReactJS escrito en TypeScript.....	21
Sección 3.2: Instalación y configuración .....	21
Sección 3.3: Componentes React sin estado en TypeScript.....	22
Sección 3.4: Componentes sin estado ni propiedades .....	23
<b>Capítulo 4: Estados en React .....</b>	<b>24</b>
Sección 4.1: Estado básico.....	24
Sección 4.2: Antipatrón común .....	24
Sección 4.3: setState().....	25
Sección 4.4: Estado, eventos y controles gestionados.....	27
<b>Capítulo 5: Props en React .....</b>	<b>28</b>
Sección 5.1: Introducción .....	28
Sección 5.2: Props por defecto .....	28
Sección 5.3: PropTypes.....	29
Sección 5.4: Transmisión de puntales mediante operador de propagación.....	30
Sección 5.5: Props.children y composición de componentes.....	31
Sección 5.6: Detección del tipo de componentes infantiles.....	31
<b>Capítulo 6: Ciclo de vida de los componentes React .....</b>	<b>33</b>
Sección 6.1: Creación de componentes.....	33
Sección 6.2: Extracción de componentes.....	35

Sección 6.3: Actualización de componentes .....	36
Sección 6.4: Llamada al método del ciclo de vida en diferentes estados .....	37
Sección 6.5: Contenedor de componentes React .....	37
<b>Capítulo 7: Formularios y entradas de usuario .....</b>	<b>39</b>
Sección 7.1: Componentes controlados .....	39
Sección 7.2: Componentes no controlados .....	39
<b>Capítulo 8: React Boilerplate [React + Babel + Webpack] .....</b>	<b>41</b>
Sección 8.1: proyecto react-starter .....	41
Sección 8.2: Puesta en marcha del proyecto .....	42
<b>Capítulo 9: Uso de ReactJS con jQuery .....</b>	<b>45</b>
Sección 9.1: ReactJS con jQuery .....	45
<b>Capítulo 10: Enrutamiento de React .....</b>	<b>47</b>
Sección 10.1: Ejemplo de archivo Routes.js, seguido del uso de Router Link en el componente .....	47
Sección 10.2: Enrutamiento React Async .....	48
<b>Capítulo 11: Comunicación entre componentes .....</b>	<b>49</b>
Sección 11.1: Comunicación entre componentes funcionales sin estado .....	49
<b>Capítulo 12: Cómo configurar un entorno básico de webpack, react y babel .....</b>	<b>51</b>
Sección 12.1: Cómo construir un pipeline para un “Hola mundo” personalizado con imágenes .....	51
<b>Capítulo 13: React.createClass vs extends React.Component .....</b>	<b>56</b>
Sección 13.1: Crear componente React .....	56
Sección 13.2: Contexto “this” .....	56
Sección 13.3: Declarar Props y PropTypes por defecto .....	58
Sección 13.4: Mixins .....	60
Sección 13.5: Establecer estado inicial .....	61
Sección 13.6: ES6/Reacciona la palabra clave «this» con ajax para obtener datos del servidor .....	62
<b>Capítulo 14: Llamada AJAX de React .....</b>	<b>63</b>
Sección 14.1: Solicitud HTTP GET .....	63
Sección 14.2: Petición HTTP GET y bucle de datos .....	63
Sección 14.3: Ajax en React sin librerías de terceros - también conocido como VanillaJS .....	65
<b>Capítulo 15: Comunicación entre componentes .....</b>	<b>66</b>
Sección 15.1: Componentes de hijo a padre .....	66
Sección 15.2: Componentes no relacionados .....	66
Sección 15.3: Componentes de padre a hijo .....	67
<b>Capítulo 16: Componentes funcionales sin estado .....</b>	<b>68</b>
Sección 16.1: Componentes funcionales sin estado .....	68
<b>Capítulo 17: Rendimiento .....</b>	<b>71</b>
Sección 17.1: Medición del rendimiento con ReactJS .....	71
Sección 17.2: Algoritmo diff de React .....	71

Sección 17.3: Conceptos básicos - DOM HTML frente a DOM virtual .....	72
Sección 17.4: Trucos y consejos .....	72
<b>Capítulo 18: Introducción al renderizado del lado del servidor .....</b>	<b>74</b>
Sección 18.1: Componentes de renderizado.....	74
<b>Capítulo 19: Configuración del entorno React .....</b>	<b>75</b>
Sección 19.1: Componente simple de React .....	75
Sección 19.2: Instalar todas las dependencias.....	75
Sección 19.3: Configurar webpack.....	75
Sección 19.4: Configurar babel.....	75
Sección 19.5: Archivo HTML para utilizar el componente react.....	76
Sección 19.6: Transpile y empaquete su componente.....	76
<b>Capítulo 20: Uso de React con Flow .....</b>	<b>77</b>
Sección 20.1: Uso de Flow para comprobar los tipos de accesorios de los componentes funcionales sin estado .....	77
Sección 20.2: Uso de Flow para comprobar los tipos de accesorios.....	77
<b>Capítulo 21: JSX .....</b>	<b>78</b>
Sección 21.1: Props en JSX.....	78
Sección 21.2: Hijos en JSX.....	79
<b>Capítulo 22: Formularios de React .....</b>	<b>82</b>
Sección 22.1: Componentes controlados.....	82
<b>Capítulo 23: Soluciones de interfaz de usuario .....</b>	<b>84</b>
Sección 23.1: Panel básico.....	84
Sección 23.2: Panel.....	84
Sección 23.3: Tab.....	85
Sección 23.4: PanelGroup .....	85
Sección 23.5: Ejemplo de vista con PanelGroups.....	86
<b>Capítulo 24: Utilización de ReactJS en Flux .....</b>	<b>89</b>
Sección 24.1: Flujo de datos.....	89
<b>Capítulo 25: Instalación de React, Webpack y TypeScript .....</b>	<b>90</b>
Sección 25.1: webpack.config.js.....	90
Sección 25.2: tsconfig.json .....	90
Sección 25.3: Mi primer componente.....	91
<b>Capítulo 26: Cómo y por qué usar llaves en React .....</b>	<b>92</b>
Sección 26.1: Ejemplo básico .....	92
<b>Capítulo 27: Llaves en React .....</b>	<b>93</b>
Sección 27.1: Utilizar el id de un elemento .....	93
Sección 27.2: Utilización del índice de array .....	94
<b>Capítulo 28: Componentes de orden superior .....</b>	<b>95</b>

Sección 28.1: Componente de orden superior que comprueba la autenticación.....	95
Sección 28.2: Componente simple de orden superior.....	96
<b>Capítulo 29: React con Redux .....</b>	<b>97</b>
Sección 29.1: Utilizar Connect.....	97
<b>Apéndice A: Instalación.....</b>	<b>98</b>
Sección A.1: Configuración sencilla .....	98
Sección A.2: Uso de webpack-dev-server .....	100
<b>Apéndice B: Herramientas para React .....</b>	<b>101</b>
Sección B.1: Enlaces .....	101
<b>Créditos .....</b>	<b>102</b>

## Acerca de

Este libro ha sido traducido por [rortegag.com](https://rortegag.com)

Si desea descargar el libro original, puede descargarlo desde:

<https://goalkicker.com/ReactJSBook/>

Si desea contribuir con una donación, hazlo desde:

<https://www.buymeacoffee.com/GoalKickerBooks>

Por favor, siéntase libre de compartir este PDF con cualquier persona de forma gratuita, la última versión de este libro se puede descargar desde:

<https://goalkicker.com/ReactJSBook/>

Este libro React JS Apuntes para Profesionales está compilado a partir de la [Documentación de Stack Overflow](#), el contenido está escrito por la hermosa gente de Stack Overflow. El contenido del texto está liberado bajo Creative Commons BY-SA, ver los créditos al final de este libro quién contribuyó a los distintos capítulos. Las imágenes pueden ser copyright de sus respectivos propietarios a menos que se especifique lo contrario.

Este es un libro no oficial gratuito creado con fines educativos y no está afiliado con los grupo(s) o empresa(s) oficiales de React JS ni Stack Overflow. Todas las marcas comerciales y marcas registradas son propiedad de sus respectivos propietarios de la empresa.

No se garantiza que la información presentada en este libro sea correcta ni exacta. Utilícelo bajo su propia responsabilidad.

Envíe sus comentarios y correcciones a [web@petercv.com](mailto:web@petercv.com)

# Capítulo 1: Introducción a React

Versión	Fecha de publicación
<a href="#">0.3.0</a>	29-05-2013
<a href="#">0.4.0</a>	17-07-2013
<a href="#">0.5.0</a>	16-10-2013
<a href="#">0.8.0</a>	19-12-2013
<a href="#">0.9.0</a>	20-02-2014
<a href="#">0.10.0</a>	21-03-2014
<a href="#">0.11.0</a>	17-07-2014
<a href="#">0.12.0</a>	28-10-2014
<a href="#">0.13.0</a>	10-03-2015
<a href="#">0.14.0</a>	07-10-2015
<a href="#">15.0.0</a>	07-04-2016
<a href="#">15.1.0</a>	20-05-2016
<a href="#">15.2.0</a>	01-07-2016
<a href="#">15.2.1</a>	08-07-2016
<a href="#">15.3.0</a>	29-07-2016
<a href="#">15.3.1</a>	19-08-2016
<a href="#">15.3.2</a>	19-09-2016
<a href="#">15.4.0</a>	16-11-2016
<a href="#">15.4.1</a>	23-11-2016
<a href="#">15.4.2</a>	06-01-2017
<a href="#">15.5.0</a>	07-04-2017
<a href="#">15.6.0</a>	13-06-2017
<a href="#">15.6.1</a>	14-06-2017
<a href="#">15.6.2</a>	25-09-2017
<a href="#">16.0.0</a>	26-09-2017
<a href="#">16.1.0</a>	09-11-2017
<a href="#">16.1.1</a>	13-11-2017
<a href="#">16.3.0</a>	29-03-2018
<a href="#">16.3.1</a>	03-04-2018
<a href="#">16.3.2</a>	16-04-2018

## Sección 1.1: ¿Qué es ReactJS?

ReactJS es una librería front-end de código abierto basada en componentes, responsable únicamente de la **capa de visualización** de la aplicación. Su mantenimiento corre a cargo de Facebook.

ReactJS utiliza un mecanismo basado en el DOM virtual para rellenar los datos (vistas) en el DOM HTML. El DOM virtual funciona rápido gracias al hecho de que sólo cambia elementos individuales del DOM en lugar de recargar el DOM completo cada vez.

Una aplicación React se compone de varios **componentes**, cada uno de los cuales es responsable de producir un pequeño fragmento de HTML reutilizable. Los componentes pueden anidarse dentro de otros componentes para permitir la construcción de aplicaciones complejas 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 abierta en ese momento.

React nos permite escribir componentes utilizando un lenguaje específico del dominio llamado JSX. JSX nos permite escribir nuestros componentes usando HTML, mientras mezclamos eventos JavaScript. React convertirá internamente esto en un DOM virtual y, en última instancia, mostrará nuestro HTML.

React *"reacciona"* a los cambios de estado en sus componentes de forma rápida y automática para volver a renderizar los componentes en el DOM 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 directamente en el DOM del navegador, React puede actuar con rapidez y sólo añadir, actualizar y eliminar componentes que hayan cambiado desde que se produjo el último ciclo de renderización.

## Sección 1.2: Instalación o configuración

ReactJS es una biblioteca JavaScript contenida en un único archivo `react-<version>.js` que puede incluirse en cualquier página HTML. También es habitual instalar la biblioteca React DOM `react-dom-<version>.js` junto con el archivo principal de 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">
      // Utilice el código JavaScript de react aquí o en un archivo aparte
    </script>
  </body>
</html>
```

Para obtener los archivos JavaScript, vaya a [la página de instalación](#) de la documentación oficial de React.

React también es compatible con la [sintaxis JSX](#). JSX es una extensión creada por Facebook que añade sintaxis XML a JavaScript. Para utilizar JSX es necesario incluir la biblioteca Babel y cambiar `<script type="text/javascript">` por `<script type="text/babel">` para traducir JSX a 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">
      // Utilice el código react JSX aquí o en un archivo separado
    </script>
  </body>
</html>
```

### Instalación a través de npm

También puedes instalar React usando [npm](#) haciendo lo siguiente:

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

Para utilizar React en tu proyecto JavaScript, puedes hacer lo siguiente:

```
var React = require('react');
var ReactDOM = require('react-dom');
```

```
ReactDOM.render(<App />, ...);
```

### Instalación a través de Yarn

Facebook lanzó su propio gestor de paquetes llamado [Yarn](#), que también se puede utilizar para instalar React. Después de instalar Yarn sólo tienes que ejecutar este comando:

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

A continuación, puede utilizar React en su proyecto exactamente de la misma manera que si hubiera instalado React a través de npm.



## Sección 1.3: Hello World con funciones sin estado

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

**Por ejemplo:**

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

Como se 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 denomina *efecto secundario*. Ya que el componente afecta a otras cosas.

Por lo tanto, se aconseja utilizar componentes sin estado más a menudo, ya que están *libres de efectos secundarios* y crearán siempre el mismo comportamiento. Eso es lo que quieres en tus aplicaciones, porque un estado fluctuante es el peor escenario para un programa mantenible.

El tipo más básico de componente react es el que no tiene estado. Los componentes React que son puras funciones de sus `props`(propiedades) y no requieren ninguna gestión de estado interno pueden ser escritos como simples funciones JavaScript. Se dice que estos son `Stateless Functional Components` porque son una función sólo de `props`, sin tener ningún `state` para realizar un seguimiento.

He aquí un ejemplo sencillo para ilustrar el concepto de componente funcional sin estado:

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

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

ReactDOM.render(<MyComponent name="Arun" />, element);
// Esto renderizara <h1>Hello, Arun!</h1>
```

Observe que todo lo que hace este componente es mostrar un elemento `h1` que contiene el `name` prop. Este componente no guarda ningún estado. Aquí hay un ejemplo 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 gestionar el estado, React tiene más espacio para optimizaciones. La implementación es limpia, pero hasta ahora [no se han implementado optimizaciones de este tipo para componentes sin estado](#).

## Sección 1.4: Fundamentos absolutos de la creación de componentes reutilizables

### Componentes y props

Como React sólo se ocupa de la vista de una aplicación, la mayor parte del desarrollo en React será la creación de componentes. Un componente representa una porción de la vista de tu aplicación. “Props” son simplemente los atributos utilizados en un nodo JSX (por ejemplo, `<SomeComponent someProp="some prop's value" />`), y son la forma principal en que nuestra aplicación interactúa con nuestros componentes. En el snippet 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 simples funciones - toman la entrada en forma de “props”, y producen la salida como marcado. Muchos componentes simples van un paso más allá, convirtiéndose en “Funciones Puras”, lo que significa que no producen efectos secundarios, y son idempotentes (dado un conjunto de entradas, el componente siempre producirá la misma salida). Este objetivo se puede cumplir formalmente mediante la creación de componentes como funciones, en lugar de “clases”. Hay tres maneras de crear un componente React:

- **Componentes funcionales («Stateless»)**

```
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} />  
  );  
}
```

Todos los ejemplos anteriores producirán un marcado idéntico.

Los componentes funcionales no pueden tener “estado”. Así que, si su componente necesita tener un estado, entonces opte por componentes basados en clases. Consulte Creación de componentes para obtener más información.

Como nota final, las props de React son inmutables una vez que han sido pasadas, lo que significa que no pueden ser modificadas desde dentro de un componente. Si el padre de un componente cambia el valor de una

prop, React se encarga de reemplazar las props antiguas por las nuevas, el componente se renderizará usando los nuevos valores.

Ver [Thinking In React](#) y [Reusable Components](#) para profundizar en la relación de los props con los componentes.

## Sección 1.5: Create React App

[create-react-app](#) (**¡IMPORTANTE! create-react-app está obsoleto, visita [la página oficial de React para generar una aplicación de React desde cero](#)**) es un generador de aplicaciones React creado por Facebook. Proporciona un entorno de desarrollo configurado para facilitar su uso con una configuración mínima, incluyendo:

- Transpilación ES6 y JSX
- Servidor de desarrollo con recarga de módulos en caliente
- Linting de código
- Prefixificación automática de CSS
- Script de compilación con agrupación de JS, CSS e imágenes, y mapas de fuentes
- Marco de pruebas Jest

### Instalación

Primero, instala create-react-app globalmente con el gestor de paquetes node (npm).

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

A continuación, ejecute el generador en el 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 intencionadamente no configurable por defecto. Si se requiere un uso no predeterminado, por ejemplo, para utilizar un lenguaje CSS compilado como Sass, entonces se puede utilizar el comando eject.

```
npm run eject
```

Esto permite editar todos los archivos de configuración. N.B. este es un proceso irreversible.

### Alternativas

Las plantillas de React alternativas incluyen:

- [enclave](#)
- [nwb](#)
- [motion](#)
- [rackt-cli](#)
- [budō](#)
- [rwb](#)
- [quik](#)
- [sagui](#)
- [roc](#)

### Crear una aplicación React

Para crear una aplicación lista para producción, ejecute el siguiente comando

```
npm run build
```

## Sección 1.6: Hello World

### Sin JSX

He aquí un ejemplo básico que utiliza la API principal de React para crear un elemento React y la API DOM de React para renderizar el elemento React en el navegador.

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

    <!-- Incluir las bibliotecas React y ReactDOM -->
    <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">

      // crear un elemento React rElement
      var rElement = React.createElement('h1', null, 'Hello, world!');

      // dElement es un contenedor DOM
      var dElement = document.getElementById('example');

      // renderizar el elemento React en el contenedor DOM
      ReactDOM.render(rElement, dElement);

    </script>
  </body>
</html>
```

### Con JSX

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

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

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

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

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

Así que finalmente el ejemplo anterior se convierte en:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>
    <!-- Incluir las bibliotecas React y ReactDOM -->
    <script src="https://fb.me/react-15.2.1.js"></script>
    <script src="https://fb.me/react-dom-15.2.1.js"></script>
    <!-- Incluir la biblioteca Babel -->
    <script src="https://npmcdn.com/babel-core@5.8.38/browser.min.js"></script>
  </head>
  <body>
    <div id="example"></div>
    <script type="text/babel">
      // crear un elemento React rElement usando JSX
      var rElement = <h1>Hello, world!</h1>;

      // dElement es un contenedor DOM
      var dElement = document.getElementById('example');

      // renderizar el elemento React en el contenedor DOM
      ReactDOM.render(rElement, dElement);
    </script>
  </body>
</html>
```

## Sección 1.7: Componente Hello World

Un componente React puede definirse como una clase ES6 que extiende la clase base `React.Component`. En su forma mínima, un componente *debe* definir un método de renderizado que especifique cómo se renderiza el componente en el DOM. El método `render` devuelve nodos React, que pueden ser definidos usando sintaxis JSX como etiquetas 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 con el fin de especificar algunos valores que el componente no puede conocer por sí mismo; una propiedad también puede contener una función que puede ser llamada por el componente después de que ocurran ciertos eventos - por ejemplo, un botón podría recibir una función para su propiedad `onClick` y llamarla cada vez que se haga clic sobre él. Al escribir un componente, se puede acceder a sus propiedades a través del objeto `props` del 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 renderizar una cadena arbitraria pasada a la prop `name` por su padre. Tenga en cuenta que un componente no puede modificar los `props` que recibe.

Un componente puede ser renderizado dentro de cualquier otro componente, o directamente en el DOM si es el componente superior, usando `ReactDOM.render` y proporcionándole tanto el componente como el Nodo DOM donde quieres que se renderice 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'))
```

Ahora ya sabes cómo crear un componente básico y aceptar accesorios. Vamos a dar un paso más e introducir el `state`.

A modo de demostración, hagamos que nuestra aplicación `Hello World` muestre sólo el nombre de pila si se da un nombre completo.

```
import React from 'react'

class Hello extends React.Component {

  constructor(props){

    // Ya que estamos extendiendo el constructor por defecto,
    // manejaremos primero las actividades por defecto.
    super(props);

    // Extraer el nombre de pila de la prop
    let firstName = this.props.name.split(" ")[0];

    // En el constructor, no dudes en modificar la propiedad
    // propiedad state en el contexto actual.
    this.state = {
      name: firstName
    }

    // Mira mama, ¡no se requiere coma en las definiciones de clase basadas en JSX!

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

  }

}

export default Hello
```

**Nota:** Cada componente puede tener su propio estado o aceptar el estado de su padre como prop.

[Codepen Enlace al ejemplo.](#)

# Capítulo 2: Componentes

## Sección 2.1: Creación de componentes

Se trata de una ampliación del Ejemplo Básico:

### Estructura básica

```
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 estado (en el sentido React de la palabra).

En tal caso, algunas personas consideran preferible utilizar componentes funcionales sin estado, que se basan en [funciones de flecha ES6](#).

### Componentes funcionales sin estado

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

Tienen 2 características principales:

1. Cuando se renderizan reciben un objeto con todos los props que se pasaron
2. Deben devolver el JSX a renderizar

*// Cuando uses JSX dentro de un módulo debes importar React*

```
import React from 'react';
import PropTypes from 'prop-types';
```

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

*// los componentes de flecha también pueden tener validación de accesorios*

```
FirstComponent.propTypes = {
  name: PropTypes.string.isRequired,
}
```

*// Para utilizar FirstComponent en otro archivo, debe exponerse mediante una llamada a la exportación:*

```
export default FirstComponent;
```



## Componentes con estado

A diferencia de los componentes “sin estado” mostrados anteriormente, los componentes “con estado” tienen un objeto de estado que puede actualizarse con el método `setState`. El estado debe ser inicializado en el `constructor` antes de que pueda ser establecido:

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

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

    this.state = {
      toggle: true
    };

    // Esto es para vincular el contexto al pasar onClick como una devolución de llamada
    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>
    );
  }
}
```

Al extender un componente con `PureComponent` en lugar de `Component`, se implementará automáticamente el método del ciclo de vida `shouldComponentUpdate()` con una comparación superficial de propiedad y estado. Esto mantiene el rendimiento de su aplicación reduciendo la cantidad de renderizaciones innecesarias que se producen. Esto asume que tus componentes son 'Puros' y siempre renderizan la misma salida con el mismo estado y propiedades de entrada.

## 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');
  }

  /* El componente de orden superior toma otro componente como parámetro y luego lo renderiza
  con accesorios adicionales */
  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 se desea compartir la lógica entre varios componentes, independientemente de lo diferentes que sean.

## Sección 2.2: Componente básico

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 independiente:

**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 />,
  // Tenga en cuenta que esto es lo mismo que la variable que almacenó anteriormente
  document.getElementById('content')
);
```

Obtendrá el siguiente resultado (observe lo que hay dentro de `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>
```

## Sección 2.3: Componentes anidados

Gran parte de la potencia de ReactJS es su capacidad para permitir el anidamiento de componentes. Tomemos los dos componentes siguientes:

```
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 hacer referencia 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 /> // Que se definió anteriormente y puede reutilizarse
        <CommentForm /> // Lo mismo digo.
      </div>
    );
  }
});
```

El anidamiento posterior puede hacerse de tres maneras, cada una de las cuales tiene sus propios lugares de utilización.

### 1. Anidar sin utilizar hijos

*(continuación del anterior)*

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

Es el estilo en el que A compone B y B compone C.

#### Pros

- Separación fácil y rápida de los elementos de la interfaz de usuario
- Facilidad para inyectar propiedades a los hijos en función del estado del componente padre.

#### Contras

- Menor visibilidad de la arquitectura de composición
- Menor reutilización

#### Bueno si

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

## 2. Anidamiento mediante hijos

(continuación del anterior)

```
var CommentBox = react.createClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList>
          <ListTitle/> // hijo
        </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 padres.

### Pros

- Mejor gestión del ciclo de vida de los componentes
- Mejor visibilidad de la arquitectura de composición
- Mejor reutilización

### Contras

- Inyectar propiedades puede llegar a ser un poco caro
- Menos flexibilidad y potencia en los componentes infantiles

### Bueno si

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

B renderizaría C usando `this.props.children`, y no hay una forma estructurada para que B sepa para qué son esos hijos. Por lo tanto, B puede enriquecer los componentes hijos dando propiedades adicionales hacia abajo, pero si B necesita saber exactamente lo que son, #3 podría ser una mejor opción.

## 3. Anidamiento mediante accesorios

(continuación del anterior)

```
var CommentBox = react.createClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList title={ListTitle}/> // propiedad
        <CommentForm />
      </div>
    );
  }
});
```

Es el estilo en el que A compone B y B ofrece una opción para que A le pase algo para componer con un fin específico. Composición más estructurada.

### Pros

- La composición como característica
- Validación sencilla
- Mayor compatibilidad

## Contras

- Inyectar propiedades puede llegar a ser un poco caro
- Menos flexibilidad y potencia en los componentes infantiles

## Bueno si

- B tiene características específicas definidas para componer algo
- B sólo debe saber cómo renderizar, no qué renderizar

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

## Sección 2.4: Props

Los `props` son una forma de pasar información a un componente React, pueden ser de cualquier tipo incluyendo funciones - a veces llamadas callbacks.

En JSX los props se pasan con la sintaxis de atributo

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

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

```
// La función render dentro de MyComponent
render() {
  return (
    <span>The user's ID is {this.props.userID}</span>
  )
}
```

Es importante definir todos los accesorios, sus tipos y, en su caso, su valor por defecto:

```
// definido en la parte inferior de 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 la prop `someObject` es opcional, pero la prop `userID` es obligatoria. Si no proporcionas `userID` a `MyComponent`, en tiempo de ejecución el motor React mostrará una consola advirtiéndote de que no se ha proporcionado la prop requerida. Ten cuidado, esta advertencia sólo se muestra en la versión de desarrollo de la librería React, la versión de producción no mostrará ninguna advertencia.

El uso de `defaultProps` permite simplificar

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

a

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

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

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

En el ejemplo anterior, `this.props.someObject` es `undefined` y por lo tanto la comprobación de `someKey` arrojará un error y el código se romperá. Con el uso de `defaultProps` puede utilizar con seguridad la comprobación anterior.

## Sección 2.5: Estados de los componentes - Interfaz de usuario dinámica

Supongamos que queremos tener el siguiente comportamiento - Tenemos un encabezado (digamos 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 muy sencillo e intuitivo utilizando estados de componentes y sentencias `if else`. (Explicación del código más abajo).

```
// He utilizado elementos ReactBootstrap. Pero el código funciona con elementos html regulares
también
var Button = ReactBootstrap.Button;
var Form = ReactBootstrap.Form;
var FormGroup = ReactBootstrap.FormGroup;
var FormControl = ReactBootstrap.FormControl;

var Comment = react.createClass({
  getInitialState: function() {
    return {show: false, newTitle: ''};
  },
  handleTitleSubmit: function() {
    // código para manejar el envío del cuadro de entrada - por ejemplo, emitir una petición
    // ajax para cambiar el nombre en la base de datos
  },
  handleTitleChange: function(e) {
    // para cambiar el nombre en el cuadro de entrada del formulario. newTitle se inicializa
    // como cadena de caracteres vacía. Necesitamos actualizarlo con la cadena actualmente
    // introducida por el usuario en el formulario
    this.setState({newTitle: e.target.value});
  },
  changeComponent: function() {
    // esto activa la variable show que se utiliza para la interfaz de usuario dinámica
    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`. Basado en la variable de estado `show`, puede ser o bien un elemento `Form` o un elemento `Button`. React permite el anidamiento de componentes.

Así que podemos añadir un elemento `{clickableTitle}` en la función `render`. Busca la variable `clickableTitle`. Basándose en el valor `'this.state.show'`, muestra el elemento correspondiente.

## Sección 2.6: Variaciones de los componentes funcionales sin estado

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

// una línea
const Language = ({language}) => <li>{language}</li>

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

/**
 * Si hay más de una línea.
 * Tenga en cuenta que los corchetes redondos son opcionales aquí,
 * Sin embargo, es mejor usarlos para facilitar la lectura.
 */
const LanguagesList = ({languages}) => {
  <ul>
    {languages.map(language => <Language language={language} />)}
  </ul>
}

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

/**
 * Esta sintaxis se utiliza si hay más trabajo además de sólo la presentación JSX
 * Por ejemplo, algunas manipulaciones de datos deben hacerse.
 * Tenga en cuenta que se requieren corchetes después de retorno,
 * De lo contrario return no devolverá nada (undefined)
 */
const LanguageSection = ({header, languages}) => {
  // trabaja
  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](#) encontrará un ejemplo práctico.

## Sección 2.7: dificultades de setState

Debe tener cuidado cuando utilice `setState` en un contexto asíncrono. Por ejemplo, podrías intentar llamar a `setState` en la devolución de llamada de una solicitud `get`:

```
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 causar problemas - si el callback es llamado después de que el `Component` es desmontado, entonces `this.setState` no será una función. Siempre que este sea el caso, usted debe tener cuidado para asegurarse de que su uso de `setState` es cancelable.

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

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

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

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

    // Cancelar la petición xhr, para que nunca se llame a la devolución de llamada
    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 `async` se guarda como un estado. En el `componentWillUnmount` se realiza toda su limpieza - incluyendo la cancelación de la petición XHR.

También puedes hacer algo más complejo. En este ejemplo, estoy creando una función 'stateSetter' que acepta el objeto `this` como argumento y previene `this.setState` cuando la función `cancel` ha sido llamada:

```

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 `setState` que hemos creado.

# Capítulo 3: Uso de ReactJS con TypeScript

## Sección 3.1: Componente ReactJS escrito en TypeScript

En realidad, puedes usar los componentes de ReactJS en Typescript como en el ejemplo de facebook. Solo reemplaza la extensión del archivo 'jsx' por '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 principal característica de Typescript (comprobación estática de tipos) se deben hacer un par de cosas:

### 1) convertir el ejemplo 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, añade las interfaces Props y State:

```
interface IHelloMessageProps {
  name:string;
}

interface IHelloMessageState {
  // vacío en nuestro caso
}

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 olvida pasar props. O si añade accesorios que no están definidos en la interfaz.

## Sección 3.2: Instalación y configuración

Para usar typescript con react en un proyecto node, primero debes tener un directorio de proyecto inicializado con npm. Para inicializar el directorio con npm `init`

### Instalación mediante npm o yarn

Puedes instalar React usando [npm](#) haciendo lo siguiente:

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

Facebook lanzó su propio gestor de paquetes llamado [Yarn](#), que también se puede utilizar para instalar React. Después de instalar Yarn sólo tienes que ejecutar este comando:

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

A continuación, puede utilizar React en su proyecto exactamente de la misma manera que si hubiera instalado React a través de `npm`.

### Instalación de definiciones de tipo react en Typescript 2.0+

Para compilar tu código usando typescript, añade/instala archivos de definición de tipos usando `npm` o `yarn`.

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

o, utilizando `yarn`

```
yarn add --dev @types/react @types/react-dom
```

### Instalación de definiciones de tipo react en versiones antiguas de Typescript

Tiene que utilizar un paquete independiente llamado [tsd](#)

```
tsd install react react-dom --save
```

### Añadir o modificar la configuración de Typescript

Para utilizar [JSX](#), un lenguaje que mezcla javascript con `html/xml`, hay que cambiar la configuración del compilador de typescript. En el archivo de configuración typescript del proyecto (normalmente llamado `tsconfig.json`), tendrás que añadir la opción JSX como:

```
"compilerOptions": {
  "jsx": "react"
},
```

Esa opción del compilador básicamente le dice al compilador typescript que traduzca las etiquetas JSX en código a llamadas a funciones javascript.

Para evitar que el compilador typescript convierta JSX en llamadas a funciones javascript sin formato, utilice

```
"compilerOptions": {
  "jsx": "preserve"
},
```

## Sección 3.3: Componentes React sin estado en TypeScript

Los componentes de React que son funciones puras de sus props y no requieren ningún estado interno se pueden escribir como funciones de JavaScript en lugar de utilizar la sintaxis de clase estándar, como:

```
import React from 'react'

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

Lo mismo puede conseguirse en Typescript utilizando 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 lo tanto, cualquiera de los dos puede ser utilizado.

## Sección 3.4: Componentes sin estado ni propiedades

El componente react más simple sin estado y sin propiedades puede escribirse como:

```
import * as React from 'react';
```

```
const Greeter = () => <span>Hello, World!</span>
```

Ese componente, sin embargo, no puede acceder a `this.props` ya que typescript no puede saber si es un componente react. Para acceder a sus props, usa:

```
import * as React from 'react';
```

```
const Greeter: React.SFC<{}> = props => () => <span>Hello, World!</span>
```

Aunque el componente no tenga propiedades definidas explícitamente, 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 es en la sencilla creación de plantillas de páginas. A continuación, se muestra un ejemplo de componente de página simple, suponiendo que ya hay componentes hipotéticos `Container`, `NavTop` y `NavBottom` 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` puede ser utilizado posteriormente por cualquier otra página real como plantilla base.

# Capítulo 4: Estados en React

## Sección 4.1: Estado básico

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

En el siguiente ejemplo estamos definiendo algún estado inicial en la función `constructor` de nuestro componente y hacemos uso de él en la función `render`.

```
class ExampleComponent extends React.Component {
  constructor(props){
    super(props);
    // Configurar nuestro estado inicial
    this.state = {
      greeting: 'Hiya Buddy!'
    };
  }

  render() {
    // Podemos acceder a la propiedad greeting a través de this.state
    return(
      <div>{this.state.greeting}</div>
    );
  }
}
```

## Sección 4.2: Antipatrón común

No debes guardar `props` en el `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` prop se guarda en estado y luego se modifica. En su lugar, elija guardar los cambios en un estado y, a continuación, construya la ruta completa utilizando tanto el `state` como las `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 único componente, y sólo un componente. Es responsabilidad de este componente almacenar los datos dentro de su estado, y distribuir los datos a otros componentes a través de `props`.

En el primer ejemplo, tanto la clase `MyComponent` como su padre están manteniendo `'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 se hace cada vez más difícil seguir el flujo de datos a través de nuestra aplicación. Contrastando esto con el segundo ejemplo - `url` sólo se mantiene en el estado del componente padre, y se utiliza como `prop` en `MyComponent` - por lo tanto, mantenemos una única fuente de verdad.

## Sección 4.3: `setState()`

La principal forma de actualizar la interfaz de usuario de las aplicaciones React es mediante una llamada a la función `setState()`. Esta función realizará una *fusión superficial* entre el nuevo estado que proporciones y el estado anterior, y activará una nueva renderización de tu componente y todos los descendientes.

### Parámetros

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

### Uso:

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

## Uso de `setState()` con un objeto como updater

```
//
// Un ejemplo de componente estilo ES6, actualizando el estado con un simple click de botón.
// También demuestra dónde se puede establecer el estado directamente y dónde se debe usar
// setState.
//
class Greeting extends React.Component {
  constructor(props) {
    super(props);
    this.click = this.click.bind(this);
    // Establecer estado inicial (SÓLO PERMITIDO EN 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>
    );
  }
}
```

## Uso de `setState()` como función como updater

```
//
// Esto se utiliza más a menudo cuando se desea comprobar o hacer uso
// del estado anterior antes de actualizar cualquier valor.
//
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 múltiples llamadas pueden ser agrupadas por React y ejecutadas a la vez, y es el enfoque preferido cuando se usan props 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 ser agrupadas por React usando `Object.assign()`, resultando en que el contador se incremente en 1 en lugar de 3.

El enfoque funcional también puede utilizarse para trasladar la lógica de establecimiento de estados fuera de los componentes. Esto permite aislar y reutilizar la lógica de estado.

```
// Fuera de la clase componente, potencialmente en otro archivo/módulo
function incrementCounter(previousState, currentProps) {
  return {
    counter: previousState.counter + 1
  };
}
// Dentro del componente
this.setState(incrementCounter);
```

## Llamada a `setState()` con un objeto y una función callback

```
//  
// 'Hi There' will be logged to the console after setState completes  
//  
this.setState({ name: 'John Doe' }, console.log('Hi there'));
```

## Sección 4.4: Estado, eventos y controles gestionados

Este es un ejemplo de un componente React con un campo de entrada “gestionado”. Cada vez que el valor del campo de entrada cambia, se llama a un manejador de eventos que actualiza el estado del componente con el nuevo valor del campo de entrada. La llamada a `setState` en el manejador de eventos desencadenará una llamada a `render` actualizando el 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` será llamado y así
- `render` será llamado

Después de escribir un carácter en el campo de entrada, ¿qué elementos del DOM cambian?

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

Puedes experimentar más con esto aquí para encontrar la respuesta

# Capítulo 5: Props en React

## Sección 5.1: Introducción

`props` se utilizan para pasar datos y métodos de un componente padre a un componente hijo.

### Cosas interesantes sobre los `props`

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

### Ejemplo básico

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

## Sección 5.2: Props por defecto

`defaultProps` le permite establecer valores por defecto, o fallback, para los `props` de sus componentes. `defaultProps` es útil cuando llama a componentes desde diferentes vistas con `props` fijos, pero en algunas vistas necesita 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` será almacenado en caché y utilizado para asegurar que `this.props.randomObject` tendrá un valor si no fue especificado por el componente padre.

## Sección 5.3: PropTypes

`propTypes` le permite especificar qué accesorios necesita su componente y de qué tipo deben ser. Tu componente funcionará sin establecer `propTypes`, pero es una buena práctica definirlos ya que hará que tu componente sea más legible, actuará como documentación para otros desarrolladores que estén leyendo tu componente, y durante el desarrollo, React te advertirá si intentas establecer un prop que es de un tipo diferente a la definición que has establecido para él.

Algunos `propTypes` primitivos y de uso común 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 `propTypes`, entonces esa `prop` debe ser suministrada al crear la instancia de ese componente. Si no se proporcionan los `propTypes` requeridos, no se podrá 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 accesorios más compleja

Del mismo modo, `PropTypes` permite especificar validaciones más complejas

#### Validar un objeto

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

```

#### Validación de matrices de objetos

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

```

## Sección 5.4: Transmisión de puntales mediante operador de propagación

En lugar de

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

Donde cada propiedad necesita ser pasada como un único valor `prop` podrías usar el operador de propagación `...` soportado para arrays en ES6 para pasar todos tus valores. El componente tendrá ahora este aspecto.

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

Recuerda que las propiedades del objeto que pasas se copian en los `props` 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 se puede utilizar el operador `spread` para pasar sólo partes de `props` a los componentes hijos, entonces se puede utilizar la sintaxis de desestructuración de `props` de nuevo.

Es muy útil cuando los niños componentes necesitan muchos `props`, pero no quieren pasarlos uno a 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'
```

## Sección 5.5: Props.children y composición de componentes

Los componentes “hijos” de un componente están disponibles en una prop especial, `props.children`. Esta prop es muy útil para “componer” componentes juntos, y puede hacer que el marcado JSX sea más intuitivo o refleje 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 cuando utilicemos el componente posteriormente:

```
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 un array, React proporciona funciones de utilidad para ellos como [React.Children](#). Consideremos en el ejemplo anterior si hubiéramos querido envolver 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 props de la etiqueta `<p>` hija - debido a que los props son inmutables, estos valores no se pueden cambiar directamente. En su lugar, se debe utilizar un clon sin estos props.

Además, al añadir elementos en bucles, tenga en cuenta cómo React [reconcilia los elementos hijos durante una nueva renderización](#), y considere seriamente la inclusión de una clave global única en los elementos hijos añadidos en un bucle.

## Sección 5.6: Detección del tipo de componentes infantiles

A veces es muy útil conocer el tipo de componente hijo cuando se itera a través de ellos. Para iterar a través de los componentes hijos se puede utilizar la función `React.Children.map`:



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

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

# Capítulo 6: Ciclo de vida de los componentes React

Los métodos del ciclo de vida se utilizan para ejecutar código e interactuar con el componente en diferentes momentos de su vida. Estos métodos se basan en el montaje, actualización y desmontaje de un componente.

## Sección 6.1: Creación de componentes

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

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

### `getDefaultProps()` (sólo ES5)

Este es el **primer** método llamado.

Los valores prop devueltos por esta función se utilizarán como valores por defecto si no están definidos al instanciar el componente.

En el siguiente ejemplo, `this.props.name` será por defecto Bob si no se especifica lo contrario:

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

### `getInitialState()` (sólo ES5)

Este es el **segundo** método llamado.

El valor devuelto por `getInitialState()` define el estado inicial del componente React. El framework 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 puede utilizarse para realizar cambios finales en el componente antes de que se añada al DOM.

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

### `render()` (ES5 y ES6)

Este es el **cuarto** método llamado.

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

Esta función se recuperará después de cualquier cambio en los accesorios 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 utilizarse para:

- Preparación de temporizadores
- Obtención de datos
- Añadir escuchadores de eventos
- Manipulación de elementos DOM

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

### **Sintaxis ES6**

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

En su lugar, declaramos nuestro **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 llamar a cualquier otra función del ciclo de vida de 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  
};
```

### Sustitución de `getDefaultProps()`

Los valores por defecto para los `props` del componente se especifican estableciendo la propiedad `defaultProps` de la clase:

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

### Sustitución de `getInitialState()`

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

```
constructor(props) {  
  super(props);  
  
  this.state = {  
    count: this.props.initialCount  
  };  
}
```

## Sección 6.2: Extracción de componentes

### `componentWillUnmount()`

Este método se ejecuta **antes** de desmontar un componente del DOM.

Es un buen lugar para realizar operaciones de limpieza como:

- Eliminar escuchadores de eventos.
- Borrar temporizadores.
- Detener sockets.
- Limpieza de estados redux.

```
componentWillUnmount(){  
  ...  
}
```

Ejemplo de eliminación de un receptor de eventos adjunto 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>
        );
    }
}

```

## Sección 6.3: Actualización de componentes

### componentWillReceiveProps(nextProps)

Esta es la **primera función a la que se llama cuando cambian las propiedades**.

Cuando las **propiedades del componente cambien**, React llamará a esta función con las **nuevas propiedades**. Puedes acceder a las propiedades antiguas con `this.props` y a las nuevas con `nextProps`.

Con estas variables, puedes hacer algunas operaciones de comparación entre `props` viejos y nuevos, o llamar a una función porque una propiedad cambia, etc.

```

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

```

### shouldComponentUpdate(nextProps, nextState)

Esta es la **segunda función que se llama cuando cambian las propiedades y la primera cuando cambia el estado**.

Por defecto, si otro componente / tu componente cambia una propiedad / un estado de tu componente, **React renderizará** una nueva versión de tu 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 sobre todo para la **optimización**.

Si la función devuelve `false`, el **proceso de actualización se detiene inmediatamente**.

```

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 la actualización se llevará a cabo.

⚠: No se puede utilizar `this.state()`.

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

### `render()`

Hay algunos cambios, así que vuelve 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){}
```

## Sección 6.4: Llamada al método del ciclo de vida en diferentes estados

Este ejemplo sirve de complemento a otros ejemplos que hablan de cómo utilizar los métodos del ciclo de vida y cuándo se llamará al método.

Este ejemplo resume Qué métodos (`componentWillMount`, `componentWillReceiveProps`, etc) serán llamados y en qué secuencia será diferente para un componente en diferentes estados:

#### Cuando se inicializa un componente:

1. `getDefaultProps`
2. `getInitialState`
3. `componentWillMount`
4. `render`
5. `componentDidMount`

#### Cuando se cambia el estado de un componente:

1. `shouldComponentUpdate`
2. `componentWillUpdate`
3. `render`
4. `componentDidUpdate`

#### Cuando un componente tiene accesorios modificados:

1. `componentWillReceiveProps`
2. `shouldComponentUpdate`
3. `componentWillUpdate`
4. `render`
5. `componentDidUpdate`

#### Cuando se desmonta un componente:

1. `componentWillUnmount`

## Sección 6.5: Contenedor de componentes React

Al crear una aplicación React, a menudo es conveniente dividir los componentes en función de su responsabilidad principal, en componentes de presentación y componentes contenedores.

Los componentes de presentación sólo se ocupan de mostrar los datos - pueden ser considerados como, y a menudo se implementan como, funciones que convierten un modelo en una vista. Normalmente no mantienen

ningún estado interno. Los componentes contenedores se ocupan de gestionar los datos. Esto puede hacerse internamente a través de su propio estado, o actuando como intermediarios con una biblioteca de gestión de estados como Redux. El componente contenedor no mostrará directamente los datos, sino que los pasará a un componente de presentación.

```
// Componente contenedor
import React, { Component } from 'react';
import Api from 'path/to/api';

class CommentsListContainer extends Component {
  constructor() {
    super();
    // Establecer el estado inicial
    this.state = { comments: [] }
  }

  componentDidMount() {
    // Hacer una llamada a la API y actualizar el estado con los comentarios devueltos
    Api.getComments().then(comments => this.setState({ comments }));
  }

  render() {
    // Pasar nuestros comentarios de estado al componente de presentación
    return (
      <CommentsList comments={this.state.comments} />;
    );
  }
}

// Componente de presentación
const CommentsList = ({ comments }) => (
  <div>
    {comments.map(comment => (
      <div>{comment}</div>
    ))}
  </div>
);

CommentsList.propTypes = {
  comments: React.PropTypes.arrayOf(React.PropTypes.string)
}
```

# Capítulo 7: Formularios y entradas de usuario

## Sección 7.1: Componentes controlados

Los componentes de formulario controlados se definen con una propiedad de `value`. El valor de las entradas controladas es gestionado por React, las entradas del usuario no tendrán ninguna influencia directa en la entrada renderizada. En su lugar, un cambio en la propiedad `value` necesita 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 muestra cómo la propiedad `value` define el valor actual de la entrada y el manejador del evento `onChange` actualiza el estado del componente con la entrada del usuario.

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

## Sección 7.2: Componentes no controlados

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



```

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 manejador de eventos `onChange`, igual que para los componentes controlados. Sin embargo, en lugar de una propiedad `value`, se proporciona una propiedad `defaultValue`. Ésta determina el valor inicial de la entrada durante la primera renderización. Cualquier cambio posterior en el estado del componente no se refleja automáticamente en el valor de la entrada; si esto es necesario, se debe utilizar un componente controlado.

# Capítulo 8: React Boilerplate [React + Babel + Webpack]

## Sección 8.1: proyecto react-starter

### Acerca de este proyecto

Se trata de un proyecto sencillo. Este post le guiará para configurar el entorno para ReactJs + Webpack + Babel.

**Empezaremos** necesitando el gestor de paquetes `node` para iniciar el servidor `express` y gestionar las dependencias en todo el proyecto. Si eres nuevo en el gestor de paquetes `node`, puedes comprobarlo [aquí](#).

Nota: Instalar `node package manager` es necesario aquí.

Cree una carpeta con el nombre adecuado y navegue en ella desde el terminal o por GUI. Entonces ir 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 Node lo buscará inicialmente cuando ejecute el proyecto. Al final te pedirá que verifiques la información que has proporcionado. Puedes escribir si o modificarla. Bueno, eso es todo, nuestro archivo `package.json` está listo.

**Configuración del servidor Express** ejecutar `npm install express@4 --save`. Estas son todas las dependencias que necesitamos para este proyecto. Aquí la bandera `save` es importante, sin ella el archivo `package.json` no se actualizará. La tarea principal de `package.json` es almacenar la lista de dependencias. Se añadirá la versión `express 4`. Tu `package.json` se verá como `"dependencies": { "express": "^4.13.4",`  
`..... },`

Después de la descarga completa se puede ver que hay `node_modules` carpeta y subcarpeta de nuestras dependencias. Ahora en la raíz del proyecto crear un nuevo archivo `server.js` archivo. Ahora estamos configurando el servidor `express`. Voy a pasar todo el código y explicarlo más adelante.

```
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 acceso a todo el api de `express`.

`var app = express();` llamará a la librería `express` como función. `app.use();` permitirá añadir 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(port, function() {});` aquí nuestro puerto será `3000` y la función a la que estamos llamando verificará que nuestro servidor web se está ejecutando correctamente. Eso es todo servidor `express` está configurado.

Ahora ve a nuestro proyecto y crea una nueva carpeta `public` y crea el archivo `index.html`. `index.html` es el archivo por defecto para tu aplicación y el servidor `Express` buscará este archivo. El `index.html` es un simple archivo html que se parece a

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8"/>
  </head>
  <body>
    <h1>hello World</h1>
  </body>
</html>
```

Y ve a la ruta del proyecto a través del terminal y escribe `node server.js`. Entonces verás `* console.log('Express server is using port:3000');*`.

Vaya al navegador y escriba <http://localhost:3000> en la barra de navegación y verá hola Mundo.

Ahora ve dentro de la carpeta pública y crea un nuevo archivo `app.jsx`. JSX es un paso del preprocesador que añade sintaxis XML a tu JavaScript. Puedes usar React sin JSX pero JSX hace React 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 verse así

```
<!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 ya está todo hecho, espero que os resulte sencillo.

## Sección 8.2: Puesta en marcha del proyecto

Necesitas Node Package Manager para instalar las dependencias del proyecto. Descarga node para tu sistema operativo desde [Nodejs.org](https://nodejs.org). Node Package Manager viene con node.

También puedes usar [Node Version Manager](#) para gestionar mejor tus versiones de node y npm. Es ideal para probar tu proyecto en diferentes versiones de node. Sin embargo, no se recomienda para entornos de producción.

Una vez que hayas instalado node en tu sistema, sigue adelante e instala algunos paquetes esenciales para lanzar tu primer proyecto React usando Babel y Webpack.

Antes de empezar a pulsar comandos en el terminal. Echa un vistazo a lo que Babel y Webpack se utilizan para.

Puedes iniciar tu proyecto ejecutando `npm init` en tu 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 opcionales:

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

Puede consultar este `package.json` de ejemplo

Cree `.babelrc` en la raíz de su proyecto con el siguiente contenido:

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

Opcionalmente cree `.eslintrc` en la raíz de su proyecto con el siguiente contenido:

```
{
  "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 subir los archivos generados a su repositorio `git`.

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

Crea el archivo `webpack.config.js` con el siguiente contenido mínimo.

```
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, por último, crear un archivo `server.js` para poder ejecutar `npm start`, con el siguiente contenido:

```
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/');
});
```

Crea el archivo `src/app.js` para ver tu proyecto React hacer algo.

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

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

Ejecuta `node server.js` o `npm start` en el terminal, si has definido lo que significa `start` en tu `package.json`

# Capítulo 9: Uso de ReactJS con jQuery

## Sección 9.1: ReactJS con jQuery

En primer lugar, hay que importar la librería `jquery`. También tenemos que importar `findDOMNode` ya que vamos a manipular el dom. Y obviamente también vamos a importar `React`.

```
import React from 'react';
import { findDOMNode } from 'react-dom';
import $ from 'jquery';
```

Estamos estableciendo una función de flecha `handleToggle` que se disparará cuando se haga clic en un icono. Sólo estamos mostrando y ocultando un `div` con una referencia de nombre `'toggle'` `onClick` sobre un icono.

```
handleToggle = () => {
  const el = findDOMNode(this.refs.toggle);
  $(el).slideToggle();
};
```

Establezcamos ahora el nombre de la 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>
    );
  }
}
```

```

        <div className="ellipsis-click" onClick={this.handleToggle}>
          <i className="fa-ellipsis-h" />
        </div>
      </div>
    );
  }
}

```

¡Hemos terminado! Esta es la forma, cómo podemos utilizar **jQuery en** componente **React**.

# Capítulo 10: Enrutamiento de React

## Sección 10.1: Ejemplo de 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 renderizar 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 tu `index.js` de nivel superior que es tu punto de entrada a la aplicación, sólo necesitas renderizar este componente Router así:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Router, browserHistory } from 'react-router';
// importar el componente routes que creamos en routes.js
import routes from './routes';

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

Ahora es simplemente cuestión de usar `Link` en lugar de etiquetas `<a>` en toda tu aplicación. El uso de `Link` se comunicará con React Router para cambiar la ruta de React Router al enlace especificado, que a su vez renderizará el componente correcto como se define en `routes.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>
  );
}
```



## Sección 10.2: Enrutamiento React Async

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

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

// para la carga diferida de un único componente, utilice lo siguiente
const ContactComponent = () => {
  return {
    getComponent: (location, callback)=> {
      require.ensure([], require => {
        callback(null, require('./components/Contact')["default"]);
      }, 'Contact');
    }
  };
};

// para múltiples componentes
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>
);
```

# Capítulo 11: Comunicación entre componentes

## Sección 11.1: 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 auto-representación de nuestros componentes funcionales, y por supuesto `React` y `ReactDOM`.

Puede ver la [demo completa](#) aquí

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

- **UserInputForm:** Este componente muestra un campo de entrada y cuando el valor del campo cambia, llama al método `inputChange` en `props` (que es proporcionado por el componente padre) y si los datos son proporcionados también, muestra eso en el campo de entrada.
- **UserDashboard:** Este componente muestra un mensaje simple y también anida el componente `UserInputForm`, también pasa el método `inputChange` al componente `UserInputForm`, el componente `UserInputForm` a su vez hace uso de este método para comunicarse con el componente padre.
  - **UserDashboardConnected:** Este componente simplemente envuelve el componente `UserDashboard` utilizando el método `connect` de `ReactRedux`, esto nos facilita la gestión del estado del componente y la actualización del componente cuando el estado cambia.
- **App:** Este componente sólo renderiza 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')
);

```

[JS Bin URL](#)

# Capítulo 12: Cómo configurar un entorno básico de webpack, react y babel

## Sección 12.1: Cómo construir un pipeline para un “Hola mundo” personalizado con imágenes

### Paso 1: Instalar Node.js

El proceso de construcción que va a construir se basa en Node.js por lo que debe asegurarse en primer lugar de que lo tiene instalado. Para obtener instrucciones sobre cómo instalar Node.js puedes consultar la documentación de SO aquí.

### Paso 2: Inicializar el proyecto como módulo de node

Abra la carpeta de su proyecto en la línea de comandos y utilice el siguiente comando:

```
npm init
```

Para los propósitos de este ejemplo usted puede sentirse libre de tomar los valores predeterminados o si desea obtener más información sobre lo que todo esto significa que usted puede comprobar fuera de este SO doc en la creación de la configuración del paquete.

### 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
```

A continuación, para las dependencias dev ejecutar este comando:

```
npm install --save-dev babel-core babel-preset-react babel-preset-es2015 webpack babel-loader  
cssloader 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 tu proyecto, si prefieres añadirlo como una dependencia entonces eso funcionará también, yo no. Aquí está el comando a ejecutar:

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

### Paso 4: Añada un archivo .babelrc a la raíz de su proyecto

Esto configurará babel para utilizar los preajustes que acaba de instalar. Su archivo .babelrc debe tener este aspecto:

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

### Paso 5: Configurar la estructura de directorios del proyecto

Configure una estructura de directorios 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 estar ya en los pasos anteriores, sólo los he incluido para que puedas ver dónde encajan.

## Paso 6: Rellenar el proyecto con los archivos del proyecto Hola Mundo

Esto no es realmente importante para el proceso de construcción de una tubería por lo que sólo le dará el código para estos y se puede copiar y pegar en ellos:

### 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">
          
        </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 sustituirla por cualquier imagen que desee. Es simplemente para demostrar que también podemos agrupar imágenes. Si proporcionas tu propia imagen y le pones un nombre diferente, tendrás que actualizar `HelloWorldComponent.jsx` para reflejar los cambios. Del mismo modo, si eliges una imagen con una extensión de archivo diferente, tendrás que modificar la propiedad `test` del cargador de imágenes en `webpack.config.js` con la regex adecuada para que coincida con tu 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 7: Crear la configuración de webpack

Crema un archivo llamado `webpack.config.js` en la raíz de tu proyecto y copia 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$/i,
        exclude: /node_modules/,
        loader: 'babel-loader'
      },
      {
        test: /\.css$/,
        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 8: Crear tareas npm para su pipeline

Para ello tendrás que añadir dos propiedades a la clave `scripts` del JSON definido en el archivo `package.json` en la raíz de tu proyecto. Haz que tu clave `scripts` tenga este aspecto:

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

La secuencia de comandos de `test` ya habrá estado allí y usted puede elegir si desea mantenerlo o no, no es importante para este ejemplo.

## Paso 9: Utilizar el pipeline

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

```
npm run build
```

Esto empaquetará la pequeña aplicación que has construido y la colocará en el directorio `dist/` que creará en la raíz de la carpeta de tu proyecto.

Si ejecutas el comando

```
npm start
```

Entonces la aplicación que has construido será servida en tu navegador web por defecto dentro de una instancia del servidor webpack dev.



# Capítulo 13: React.createClass vs extends React.Component

## Sección 13.1: Crear componente React

Exploremos las diferencias sintácticas comparando dos ejemplos de código.

### React.createClass (obsoleto)

Aquí tenemos una `const` con una clase React asignada, con la función `render` a continuación para completar la definición típica de un componente base.

```
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 convirtámosla 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 ES6. Para los cambios en React, ahora creamos una clase llamada `MyComponent` y extendemos desde `React.Component` en lugar de acceder directamente a `React.createClass`. De esta manera, usamos menos React boilerplate y más JavaScript.

PD: Típicamente esto se usaría con algo como Babel para compilar el ES6 a ES5 para que funcione en otros navegadores.

## Sección 13.2: Contexto “this”

Usando `React.createClass` se enlazaría automáticamente `this` contexto (valores) correctamente, pero ese no es el caso cuando se usan clases ES6.

### React.createClass

Observe la declaración `onClick` con el método `this.handleClick` vinculado. Cuando se llame a este método, React aplicará el contexto de ejecución correcto a `handleClick`.

```
import React from 'react';

const MyComponent = React.createClass({
  handleClick() {
    console.log(this); // la instancia de React Component
  },
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
});

export default MyComponent;
```

## React.Component

Con las clases ES6 **this** es **null** por defecto, las propiedades de la clase no se vinculan 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 varias formas de vincular el derecho a **this** contexto.

### Caso 1: Encuadernar en línea:

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  handleClick() {
    console.log(this); // la instancia de React Component
  }
  render() {
    return (
      <div onClick={this.handleClick.bind(this)}></div>
    );
  }
}

export default MyComponent;
```

## Caso 2: Bind en el constructor de la clase

Otro enfoque es cambiar el contexto de `this.handleClick` dentro del constructor. De esta forma 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); // la instancia de React Component
  }
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default MyComponent;
```

## Caso 3: Utilizar una función anónima ES6

También puede utilizar la función anónima ES6 sin tener que enlazar explícitamente:

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  handleClick = () => {
    console.log(this); // la instancia de React Component
  }
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default MyComponent;
```

## Sección 13.3: Declarar Props y PropTypes por defecto

Hay cambios importantes en la forma de utilizar y declarar los accesorios por defecto 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 las props 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 de la definición `createClass` Objeto.

El `getDefaultProps` ahora ha cambiado a sólo una propiedad `Object` en la clase llamada `defaultProps`, ya que ya no es una función "get", es sólo un `Object`. Evita más React boilerplate, 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 atajo si tu build tiene activados los inicializadores de propiedades 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;
```

## Sección 13.4: Mixins

Podemos usar `mixins` sólo con el método `React.createClass`.

### `React.createClass`

En esta versión podemos añadir `mixins` a los componentes utilizando la propiedad `mixins` que toma un array de `mixins` disponibles. Éstos extienden la clase del 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 están soportados cuando se utilizan componentes React escritos en ES6. Además, no tendrán soporte para clases ES6 en React. La razón es que se [consideran perjudiciales](#).

## Sección 13.5: Establecer estado inicial

Hay cambios en la forma de establecer 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 simple **propiedad de inicialización en el constructor**, en lugar de utilizar la función `getInitialState`. Se siente menos "React API" impulsado ya que esto es sólo 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;
```

## Sección 13.6: 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>
    );
  }
}
```

# Capítulo 14: Llamada AJAX de React

## Sección 14.1: Solicitud HTTP GET

A veces, un componente necesita renderizar algunos datos de 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 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'))
```

Una petición puede iniciarse invocando el método apropiado en el objeto `request`, y luego llamando a `.end()` para enviar la petición. Establecer campos de cabecera es sencillo, invoca `.set()` con un nombre de campo y un valor.

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

### Peticiones POST

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

Consulte la [documentación de Superagent](#) para obtener más información.

## Sección 14.2: Petición HTTP GET y bucle de datos

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



Hacemos una petición AJAX usando [fetch](#), que está integrado en la mayoría de los navegadores. Utiliza un [fetch polyfill](#) en producción para soportar navegadores antiguos. También puede utilizar cualquier otra biblioteca para hacer peticiones (por ejemplo, [axios](#), [SuperAgent](#), o incluso simple Javascript).

Establecemos los datos que recibimos como estado del componente, para poder acceder a ellos dentro del método de renderizado. Allí, hacemos un bucle a través de los datos usando [map](#). No olvides añadir siempre un [atributo key](#) único (o prop) al elemento en bucle, lo cual es importante para el rendimiento de renderizado 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.](#)

## Sección 14.3: Ajax en React sin librerías 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) {
        // ¡Éxito!
        this.setState({someData: request.responseText})
      } else {
        // Llegamos a nuestro servidor de destino, pero devolvió un error
        // Posiblemente maneje el error cambiando su estado.
      }
    };

    request.onerror = () => {
      // Hubo un error de conexión de algún tipo.
      // Posiblemente maneje el error cambiando su estado.
    };

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

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

# Capítulo 15: Comunicación entre componentes

## Sección 15.1: Componentes de hijo a padre

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

En este ejemplo, cambiaremos el estado `Parent` pasando una función al componente `Child` e invocando esa función dentro del componente `Child`.

```
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;
```

Observe que el método `outputEvent` de `Parent` (que cambia el estado del `Parent`) es invocado por el evento `onClick` del botón de `Child`.

## Sección 15.2: Componentes no relacionados

La única forma si tus componentes no tienen una relación padre-hijo (o están relacionados, pero demasiado lejos como un nieto-nieto) es tener algún tipo de señal a la que un componente se suscriba, y en la que el otro escriba.

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

Hay al menos 3 patrones para hacerlo. Puede encontrar una [comparación aquí](#).

He aquí un breve resumen:

- Patrón 1: **Emisor/objetivo/despachador de eventos**: los oyentes necesitan hacer referencia a la fuente para suscribirse.
  - para suscribirse: `otherObject.addEventListener('click', () => { alert('click!'); });`
  - para enviar: `this.dispatchEvent('click');`
- Patrón 2: **Publicar/Suscribir**: no se necesita una referencia específica a la fuente que desencadena el evento, hay un objeto global accesible en todas partes que gestiona todos los eventos.
  - para suscribirse: `globalBroadcaster.subscribe('click', () => { alert('click!'); });`
  - para enviar: `globalBroadcaster.publish('click');`
- Patrón 3: **Señales**: similar a Emisor de Eventos/Objetivo/Dispatcher pero aquí no se utilizan cadenas aleatorias. Cada objeto que pueda emitir eventos necesita tener una propiedad específica con ese nombre. De esta forma, sabes exactamente qué eventos puede emitir un objeto.
  - para suscribirse: `otherObject.clicked.add( () => { alert('click!'); });`
  - para enviar: `this.clicked.dispatch();`

## Sección 15.3: Componentes de padre a hijo

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

Puedes **pasar props a componentes hijos**. En este ejemplo el mensaje es el prop que pasamos al componente hijo, el nombre mensaje es elegido arbitrariamente, puedes llamarlo como quieras.

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

En resumen, tienes un componente (padre) que renderiza a otro (hijo) y le pasa unos props.

# Capítulo 16: Componentes funcionales sin estado

## Sección 16.1: Componentes funcionales sin estado

Los componentes 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 podíamos crear un componente React con estado usando `React.Component` (en ES6), o `React.createClass` (en ES5), independientemente de si requiere algún estado para gestionar datos o no.

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

Por ejemplo:

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

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

Los componentes funcionales sin estado suelen centrarse en la interfaz de usuario; el estado debe ser gestionado por componentes «contenedores» de nivel superior, o a través de Flux/Redux, etc. Los componentes funcionales sin estado no admiten métodos de estado o ciclo de vida.

Ventajas:

1. Sin gastos generales de clase
2. No hay que preocuparse por la palabra clave `this`
3. Fácil de escribir y fácil de entender
4. No hay que preocuparse por la gestión de los valores de estado
5. Mejora del rendimiento

**Resumen:** Si estás escribiendo un componente React que no requiere estado y te gustaría crear una interfaz de usuario reutilizable, en lugar de crear un componente React estándar puedes escribirlo como un **componente funcional sin estado**.

**Pongamos un ejemplo sencillo:**

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 interfaz de usuario para registrar y buscar usuarios. Fíjate en que es un componente React típico que incluye estado, interfaz de usuario y código de comportamiento. Los datos de la lista de usuarios registrados se almacenan en la variable `state`, pero nuestra `List` reutilizable (que se muestra a continuación) encapsula el código de interfaz de usuario 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>
    );
  }
}
```

Por último, nuestro **componente funcional sin estado** `List`, que se utiliza para mostrar tanto la lista de usuarios registrados como los resultados de la búsqueda, pero sin mantener ningún estado por sí mismo.

`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>

# Capítulo 17: Rendimiento

## Sección 17.1: Medición del rendimiento con ReactJS

**No se puede mejorar algo que no se puede medir.** Para mejorar el rendimiento de los componentes React, debes ser capaz de medirlo. ReactJS proporciona herramientas *adicionales* para medir el rendimiento. Importa el módulo `react-addons-perf` para medir el rendimiento.

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

Puede utilizar los siguientes métodos del módulo `Perf` importado:

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

El más importante y que necesitará la mayoría de las veces es `Perf.printWasted()`, que le proporciona una representación tabular del tiempo perdido por cada componente.

(index)	Owner > component	Wasted time (ms)	Instances
0	"Todos > TodoItem"	102.76999999977124	1000

Total time: 132.71 ms react-with-addons.js:9900

Puede anotar la columna **Tiempo perdido** en la tabla y mejorar el rendimiento del componente utilizando la sección **Consejos y trucos** anterior.

Consulte la [Guía oficial de React](#) y el excelente artículo de [Benchling Engg. sobre el rendimiento de React](#).

## Sección 17.2: Algoritmo diff de React

Generar el número mínimo de operaciones para transformar un árbol en otro tiene una complejidad del orden de  $O(n^3)$  donde  $n$  es el número de nodos del árbol. React se basa en dos supuestos 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 los 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 diferentes tipos.
  - for example, `<div>...</div>` is different from `<span>...</span>`
2. Cuando 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 de entender** si quiere optimizar el rendimiento

Si [dos nodos] no son del mismo tipo, React ni siquiera va a intentar que coincidan. Simplemente eliminará el primero del DOM e insertará el segundo.



He aquí por qué

Es muy improbable que un elemento vaya a generar un DOM que se parezca a lo que generaría a. En lugar de perder tiempo tratando de hacer coincidir esas dos estructuras, React simplemente reconstruye el árbol desde cero.

## Sección 17.3: Conceptos básicos - DOM HTML frente a DOM virtual

### HTML DOM es caro

Cada página web se representa internamente como un árbol de objetos. Esta representación se denomina *Modelo de Objetos del Documento*. Además, es una interfaz neutra 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, añadir o eliminar elementos HTML.

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

### El DOM virtual es una solución

Así que al equipo de React se le ocurrió la idea de abstraer el *DOM HTML* y crear su propio *DOM Virtual* para calcular el número mínimo de operaciones que necesitamos aplicar sobre el *DOM 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 **DOM virtual**. Cada vez que cambia el estado de la aplicación, estos son los pasos que realiza React para optimizar el rendimiento.

1. Generar un nuevo *DOM virtual* que represente el nuevo estado de nuestra aplicación
2. Comparar el antiguo DOM virtual (que representa el DOM HTML actual) con el nuevo DOM virtual
3. Basándose en 2. encontrar el número mínimo de operaciones para transformar el antiguo DOM virtual (que representa el actual DOM HTML) en el nuevo DOM virtual.
  - para saber más sobre eso - lea React's Diff Algorithm
4. Una vez encontradas estas operaciones, se convierten en sus operaciones *HTML DOM* equivalentes
  - recuerde, el *DOM virtual* es sólo una abstracción del *DOM HTML* y existe una relación isomórfica entre ellos
5. Ahora, el número mínimo de operaciones que se han encontrado y transferido a sus operaciones *HTML DOM* equivalentes se aplican directamente en el *HTML DOM* de la aplicación, lo que ahorra tiempo de modificar el *HTML DOM* innecesariamente.

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

## Sección 17.4: Trucos y consejos

Cuando dos nodos no son del mismo tipo, React no intenta emparejarlos - simplemente elimina el primer nodo del DOM e inserta el segundo. Por eso el primer consejo dice

1. Si se ve alternando entre dos clases de componentes con un resultado muy similar, es posible que desee que sea la misma clase.
2. Utilice `shouldComponentUpdate` para evitar que el componente se vuelva a renderizar, si sabe que no va a cambiar, por ejemplo

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

# Capítulo 18: Introducción al renderizado del lado del servidor

## Sección 18.1: Componentes de renderizado

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

### `renderToString`

Esto renderizará componentes React a HTML en el servidor. Esta función también añadirá propiedades `data-react-` a los elementos HTML para que React en el cliente no tenga que renderizar los elementos de nuevo.

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

### `renderToStaticMarkup`

Esto renderizará componentes React a HTML, pero sin propiedades `data-react-`, no es recomendable usar componentes que serán renderizados en cliente, porque los componentes se re-renderizarán.

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

# Capítulo 19: Configuración del entorno React

## Sección 19.1: Componente simple de React

Queremos poder compilar el siguiente componente y renderizarlo en nuestra página web.

**Nombre de archivo:** `src/index.js`

```
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'));
```

## Sección 19.2: 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
```

## Sección 19.3: Configurar webpack

Crea un archivo `webpack.config.js` en la raíz de tu directorio de trabajo.

**Nombre del 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"}
    ]
  }
}
```

## Sección 19.4: Configurar babel

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

**Nombre de archivo:** `.babelrc`

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

## Sección 19.5: Archivo HTML para utilizar el componente react

Configurar un archivo html simple 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>
```

## Sección 19.6: Transpile y empaquete su componente

Usando `webpack`, puedes empaquetar tu componente:

```
$ webpack
```

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

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

# Capítulo 20: Uso de React con Flow

Cómo usar el [comprobador de tipos Flow](#) para comprobar tipos en componentes React.

## Sección 20.1: Uso de Flow para comprobar los tipos de accesorios de los 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>
)
```

## Sección 20.2: Uso de Flow para comprobar 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 () {
    // el resto del código va aquí
  }
}
```

# Capítulo 21: JSX

## Sección 21.1: Props en JSX

Hay varias maneras diferentes de especificar props en JSX.

### Expresiones de JavaScript

Puede pasar **cualquier expresión JavaScript** como prop, rodeándola con `{}`. Por ejemplo, en este JSX:

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

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

Las sentencias `if` y los bucles `for` no son expresiones en JavaScript, por lo que no pueden utilizarse directamente en JSX.

### Literales de cadena de caracteres

Por supuesto, también puedes pasar cualquier **cadena de caracteres literal** como prop. Estas dos expresiones JSX son equivalentes:

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

Cuando pasas una cadena de caracteres literal, su valor es HTML-unescaped. 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 la información.

### Props Valor por Defecto

Si no pasas ningún valor para una proposición, **por defecto será `true`**. Estas dos expresiones JSX son equivalentes:

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

Sin embargo, el equipo de React dice en sus documentos que **no se recomienda usar este enfoque**, porque puede confundirse con la abreviatura de objetos ES6 `{foo}` que es la abreviatura de `{foo: foo}` en lugar de `{foo: true}`. Dicen que este comportamiento sólo está ahí para que coincida con el comportamiento de HTML.

### Atributos de difusión

Si ya tienes props como objeto, y quieres pasarlo en JSX, puedes usar `...` como operador de extensión para pasar todo el objeto props. Estos dos componentes son equivalentes:

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

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

## Sección 21.2: Hijos en JSX

En las expresiones JSX que contienen tanto una etiqueta de apertura como una etiqueta de cierre, el contenido entre esas etiquetas se pasa como una prop especial: `props.children`. Hay varias formas diferentes de pasar hijos:

### Literales de cadena de caracteres

Puedes poner una cadena de caracteres entre las etiquetas de apertura y cierre y `props.children` será simplemente esa cadena de caracteres. Esto es útil para muchos de los elementos HTML incorporados. Por ejemplo:

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

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

Tenga en cuenta que **el código HTML no está descifrado**, por lo que puede escribir JSX de la misma forma que escribiría 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 aparecen en medio de literales de cadena se condensan en un solo espacio.

### Hijos JSX

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

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

Puedes **mezclar diferentes tipos de hijos, por lo que puedes usar literales de cadena de caracteres junto con hijos JSX**. Esta es otra forma en la que JSX es como HTML, de modo que esto 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>
```

Ten en cuenta que un componente React **no puede devolver múltiples elementos React, pero una única expresión JSX puede tener múltiples hijos**. Así que si quieres que un componente devuelva múltiples cosas puedes envolverlas en un `div` como en el ejemplo anterior.

### Expresiones de JavaScript

Puede pasar cualquier expresión JavaScript como hijo, 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 representa 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 JavaScript pueden mezclarse con otros tipos de hijos.

## Funciones como hijos

Normalmente, las expresiones JavaScript insertadas en JSX se evaluarán a una cadena, un elemento React, o una lista de esas cosas. Sin embargo, `props.children` funciona como cualquier otra prop en el sentido de que puede pasar cualquier tipo de datos, no sólo los que React sabe cómo renderizar. Por ejemplo, si tienes un componente personalizado, puedes hacer que tome una llamada de retorno como `props.children`:

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

// Llama a la llamada de retorno infantil numTimes para producir un componente repetido
const Repeat = ({ numTimes, children }) => {
  let items = [];
  for (let i = 0; i < numTimes; i++) {
    items.push(children(i));
  }
  return <div>{items}</div>;
};
```

Los hijos pasados a un componente personalizado pueden ser cualquier cosa, siempre y cuando ese componente los transforme en algo que React pueda entender antes de renderizarlos. Este uso no es común, pero funciona si quieres estirar 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 muestran. Todas estas expresiones JSX se convertirán en lo mismo:

```
<MyComponent />

<MyComponent></MyComponent>

<MyComponent>{false}</MyComponent>

<MyComponent>{null}</MyComponent>

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

Esto es extremadamente útil para renderizar condicionalmente elementos React. Este JSX solo renderiza un elemento si `showHeader` es `true`:

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

Una advertencia importante es que algunos valores “falsys”, como el número 0, siguen siendo renderizados por React. Por ejemplo, este código no se comportará como cabría esperar porque se imprimirá 0 cuando props.messages sea un array vacío:

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

Una forma de solucionarlo es asegurarse de que la expresión que precede a && sea siempre booleana:

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

Por último, tenga en cuenta que, si desea que un valor como false, true, null o undefined aparezca en la salida, primero tiene que convertirlo en una cadena de caracteres:

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

# Capítulo 22: Formularios de React

## Sección 22.1: Componentes controlados

Un componente controlado se vincula a un valor y sus cambios se gestionan en código mediante llamadas de retorno 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. A continuación, vinculamos los valores de las 2 entradas a las claves individuales del objeto persona. Luego, a medida que el usuario escribe, capturamos cada valor en la función `handleChange`. Dado que los valores de los componentes están vinculados al estado, podemos renderizar a medida que el usuario escribe llamando a `setState()`.

**NOTA:** No llamar a `setState()` cuando se trata de componentes controlados, hará que el usuario escriba, pero no vea la entrada porque React sólo muestra los cambios cuando se le dice que lo haga.

También es importante tener en cuenta que los nombres de las entradas son los mismos que los nombres de las claves del 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 lo mismo es un `person.firstName` || `person.lastName`. Por supuesto, esto dependería de la entrada que se esté escribiendo en ese momento. Dado que no sabemos dónde va a

escribir el usuario, el uso de un diccionario y la correspondencia de los nombres de entrada con los nombres de las claves, nos permite capturar la entrada del usuario sin importar desde dónde se está llamando a `onChange`.

# Capítulo 23: Soluciones de interfaz de usuario

Digamos que nos inspiramos en algunas ideas de interfaces de usuario modernas utilizadas en programas y las convertimos en componentes React. En eso consiste el tema «Soluciones de interfaz de usuario». Se agradece la atribución.

## Sección 23.1: Panel básico

```
import React from 'react';

class Pane extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {
    return React.createElement(
      'section', this.props
    );
  }
}
```

## Sección 23.2: 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 el panel simple son:

- tiene el foco cuando es llamado por el script o pulsado por el ratón;
- el panel tiene el método estático `title` por componente, por lo que puede ser extendido por otro componente de panel con sobreescritura `title` (la razón aquí es que la función puede ser llamada de nuevo al renderizar para propósitos de localización, pero en los límites de este ejemplo `title` no tiene sentido);
- puede contener hojas de estilo individuales declaradas en el método estático `css` (puede precargar el contenido del archivo desde `PANEL.css`).

## Sección 23.3: Tab

```
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()
      )
    );
  }
}
```

La propiedad `panelClass` del `Tab` debe contener la clase del panel utilizado para la descripción.

## Sección 23.4: PanelGroup

```
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();
}
}

```

La propiedad `panels` de la instancia `PanelGroup` debe contener un array con objetos. Cada objeto declara datos importantes sobre los paneles:

- **name** - identificador del panel utilizado por el script controlador;
- **class** - clase del panel.

No olvide establecer la propiedad `activeTab` al nombre de la pestaña necesaria.

### Aclaración

Cuando la pestaña está abajo, el panel necesario está recibiendo el nombre de clase `active` en el elemento DOM (significa que va a ser visible) y está enfocado ahora.

## Sección 23.5: Ejemplo de vista con PanelGroups

```

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';  
  }  
}
```

# Capítulo 24: Utilización de ReactJS en Flux

Viene muy bien utilizar el enfoque Flux, cuando su aplicación con ReactJS en el frontend está previsto que crezca, debido a las estructuras limitadas y un poco de código nuevo para hacer los cambios de estado en tiempo de ejecución más fácil.

## Sección 24.1: Flujo de datos

Se trata de un esbozo de [Panorama general](#).

El patrón Flux asume el uso de un flujo de datos unidireccional.

1. **Acción** - objeto simple que describe el tipo de acción y otros datos de entrada.
2. **Despachador** - receptor de una sola acción y controlador de callbacks. Imagine que es el eje central de su aplicación.
3. **Almacenamiento** - contiene el estado y la lógica de la aplicación. Registra callback en el despachador y emite evento a la vista cuando se ha producido un cambio en la capa de datos.
4. **Vista** - Componente React que recibe eventos de cambio y datos del almacén. Hace que se vuelva a renderizar cuando se cambia algo.

Como en el flujo de datos Flux, las vistas también pueden **crear acciones** y pasarlas al despachador para las interacciones del usuario.

### Revertido

Para que quede más claro, podemos empezar por el final.

- Diferentes componentes de React (*vistas*) obtienen datos de diferentes almacenes sobre los cambios realizados.

Pocos componentes pueden ser llamados **controladores-vista**, porque proporcionan el código de cola para obtener los datos de los almacenes y pasar los datos a lo largo de la cadena de sus descendientes. Las vistas de controlador representan cualquier sección significativa de la página.

- Los *almacenes* pueden ser remarcados como callbacks que comparan el tipo de acción y otros datos de entrada para la lógica de negocio de tu aplicación.
- El *despachador* es el receptor de acciones comunes y el contenedor de callbacks.
- Las *acciones* no son más que simples objetos con la propiedad de tipo requerida.

Anteriormente, querrá utilizar constantes para los tipos de acción y los métodos de ayuda (llamados **creadores de acciones**).

# Capítulo 25: Instalación de React, Webpack y TypeScript

## Sección 25.1: 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 del estándar `entry`, `output` y otras propiedades webpack):

### El loader

Para ello debe crear una regla que compruebe las extensiones de archivo `.ts` y `.tsx`, especificando `ts-loader` como `loader`.

### Resolver las extensiones TS

También necesitas añadir las extensiones `.ts` y `.tsx` en el array `resolve`, o webpack no las verá.

## Sección 25.2: tsconfig.json

Este es un `tsconfig` mínimo para ponerte en marcha.

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

Repasemos las propiedades una por una:

### include

Se trata de una matriz de código fuente. Aquí sólo tenemos una entrada, `src/*`, que especifica que todo lo que está en el directorio `src` debe incluirse en la compilación.

### compilerOptions.target

Especifica que queremos compilar a ES5 objetivo

## compilerOptions.jsx

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

## compilerOptions.allowSyntheticDefaultImports

Práctica propiedad que le permitirá importar módulos de nodos como si fueran módulos ES6, de modo que en lugar de hacer

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

puedes hacer

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

sin ningún error que te diga que React no tiene exportación por defecto.

## Sección 25.3: 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 uses TypeScript con React, una vez que hayas descargado las definiciones de tipo de React DefinitelyTyped (npm `install --save @types/react`), cada componente requerirá que añadas anotaciones de tipo.

Hazlo así:

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

donde `AppProps` y `AppState` son interfaces (o alias de tipo) para los `props` y `state` de tus componentes respectivamente.

# Capítulo 26: Cómo y por qué usar llaves en React

Siempre que estés renderizando una lista de componentes React, cada componente necesita tener un atributo `key` (clave). La clave puede ser cualquier valor, pero tiene que ser único para esa lista.

Cuando React tiene que representar cambios en una lista de elementos, React simplemente itera sobre ambas listas de hijos al mismo tiempo y genera una mutación siempre que haya una diferencia. Si no hay claves establecidas para los hijos, React escanea cada hijo. En caso contrario, React compara las claves para saber cuáles se han añadido o eliminado de la lista.

## Sección 26.1: Ejemplo básico

Para un componente React sin clases:

```
function SomeComponent(props){
  const ITEMS = ['cat', 'dog', 'rat']

  function getItemList(){
    return ITEMS.map(item => <li key={item}>{item}</li>);
  }

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

Para este ejemplo, el componente anterior se resuelve en:

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

# Capítulo 27: Llaves en React

Las claves en react se utilizan para identificar internamente una lista de elementos DOM de la misma jerarquía.

Así que, si estás iterando sobre un array para mostrar una lista de elementos `li`, cada uno de los elementos `li` necesita un identificador único especificado por la propiedad `key`. Normalmente puede ser el id del elemento de la base de datos o el índice del array.

## Sección 27.1: Utilizar el id de un elemento

Aquí tenemos una lista de tareas que se pasa a los `props` de nuestro componente.

Cada elemento de tareas tiene una propiedad de texto e id. Imagina que la propiedad id proviene de un almacén de datos 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 `key` de cada elemento de lista iterado a `todo-${todo.id}` para que react pueda identificarlo internamente:

```
render() {  
  const { todos } = this.props;  
  return (  
    <ul>  
      { todos.map((todo) =>  
        <li key={ `todo-${todo.id}` }>  
          { todo.text }  
        </li>  
      ) }  
    </ul>  
  );  
}
```

## Sección 27.2: Utilización del índice de array

Si no dispone de identificadores de base de datos únicos, también puede utilizar el índice numérico de su array de la siguiente manera:

```
render() {  
  const { todos } = this.props;  
  return (  
    <ul>  
      { todos.map((todo, index) =>  
        <li key={ `todo-${index}` }>  
          { todo.text }  
        </li>  
      ) }  
    </ul>  
  );  
}
```

# Capítulo 28: Componentes de orden superior

Los componentes de orden superior («HOC» en sus siglas en inglés) son un patrón de diseño de aplicaciones de Reactor que se utiliza para mejorar los componentes con código reutilizable. Permiten añadir 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 nuevo componente con la funcionalidad ampliada.

## Sección 28.1: Componente de orden superior que comprueba la autenticación

Supongamos que tenemos un componente que sólo debe mostrarse si el usuario ha iniciado sesión.

Así que creamos un HOC que comprueba 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;
```



A continuación, sólo tenemos que utilizar este Componente de Orden Superior en nuestros componentes que deben estar ocultos a los 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>
    );
  }
}

// Ahora envuélva MyPrivateComponent con la función requireAuthentication
export default requireAuthentication(MyPrivateComponent);
```

Este ejemplo se describe con más detalle [aquí](#).

## Sección 28.2: Componente simple de orden superior

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 esta COA en su código:

### **MyLoggedComponent.js**

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

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

// Ahora envuélva MyLoggedComponent con la función hocLogger
export default hocLogger(MyLoggedComponent);
```

# Capítulo 29: React con Redux

Redux ha llegado a ser el status quo para la gestión de estado a nivel de aplicación en el front-end en estos días, y los que trabajan en “aplicaciones a gran escala” a menudo juran por él. Este tema cubre por qué y cómo deberías usar la librería de gestión de estado, Redux, en tus aplicaciones React.

## Sección 29.1: Utilizar Connect

Crea una tienda con Redux con `createStore`.

```
import { createStore } from 'redux'
import todoApp from './reducers'
```

```
let store = createStore(todoApp, { initialStateVariable: "derp" })
```

Utiliza `connect` para conectar el componente al almacén Redux y extrae props del almacén al componente.

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

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

```
export default VisibleTodoList
```

Define acciones que permitan a tus componentes enviar mensajes al almacén Redux.

```
/*
 * action types
 */
```

```
export const ADD_TODO = 'ADD_TODO'
```

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

Maneja estos mensajes y crea un nuevo estado para el almacén en las funciones reductoras.

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

# Apéndice A: Instalación

## Sección A.1: Configuración sencilla

### Configurar las carpetas

Este ejemplo supone que el código está en `src/` y la salida en `out/`. Por lo tanto, la estructura de carpetas debería ser similar a

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

### Configuración de los paquetes

Asumiendo un entorno npm configurado, primero necesitamos configurar babel para transpilar el código React a código compatible con es5.

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

El comando anterior le indicará a npm que instale las librerías centrales de babel, así como el módulo loader para su uso con webpack. También instalamos los presets de es6 y react para que babel entienda el código de los módulos JSX y es6. (Puede encontrar más información sobre los presets aquí [Babel presets](#))

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

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

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

Por último, necesitaremos el código real de react

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

### Configuración de webpack

Con las dependencias configuradas 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` (que se supone está en `src/`) y lo convierta en un único archivo `bundle.js` en el directorio `out`.

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

## Probar 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')
)
```

Este archivo normalmente renderizaría un simple `<h1>Hello world!</h1>` Encabezado en la etiqueta html con el id "app", pero por ahora debería ser suficiente con transpilar el código una vez.

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

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

## Sección A.2: Uso de webpack-dev-server

### Configurar

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

### Modificación de `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 se encuentran en

- `output.publicPath` que establece una ruta desde la que se servirá nuestro bundle (ver [archivos de configuración de Webpack](#) para más información)
- `devServer`
  - `contentBase` la ruta base desde la que servir los archivos estáticos (por ejemplo `index.html`)
  - `hot` establece el servidor `webpack-dev` para recargar en caliente cuando se realizan cambios en los archivos en el disco

Y, por último, sólo necesitamos un simple `index.html` 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 `$webpack-dev-server` debería iniciar un servidor http local en el puerto `8080` y al conectarse debería renderizar una página que contenga un `<h1>¡Hola mundo!</h1>`.

# Apéndice B: Herramientas para React

## Sección B.1: Enlaces

Lugares donde encontrar componentes y bibliotecas React;

- [Catálogo de componentes React](#)
- JS.coach

# Créditos

Muchas gracias a todas las personas de Stack Overflow Documentation que ayudaron a proporcionar este contenido, más cambios pueden ser enviados a [web@petercv.com](mailto:web@petercv.com) para que el nuevo contenido sea publicado o actualizado.

## Traductor al español

[rortegag](#)

<a href="#">abhirathore2006</a>	Capítulo 10
<a href="#">Adam</a>	Capítulo 16
<a href="#">Aditya Singh</a>	Capítulo 17
<a href="#">Adrián Daraš</a>	Capítulo 18
<a href="#">Ahmad</a>	Capítulo 5
<a href="#">akashrajkn</a>	Capítulo 2
<a href="#">Alex Young</a>	Capítulos 1 y 4
<a href="#">Alexander</a>	Capítulo 4
<a href="#">Alexg2195</a>	Capítulo 6
<a href="#">Anuj</a>	Capítulos 1, 5 y 6
<a href="#">Aron</a>	Capítulo 25
<a href="#">Bart Riordan</a>	Capítulos 1, 2 y 12
<a href="#">Bond</a>	Capítulo 2
<a href="#">Brad Colthurst</a>	Capítulo 4
<a href="#">Brandon Roberts</a>	Capítulo 2
<a href="#">brillout</a>	Capítulo 31
<a href="#">Chaim Friedman</a>	Capítulo 22
<a href="#">Daksh Gupta</a>	Capítulo 1
<a href="#">Danillo Corvalan</a>	Capítulo 5
<a href="#">David</a>	Capítulo 15
<a href="#">Dennis Stücken</a>	Capítulos 27 y 28
<a href="#">F. Kauder</a>	Capítulo 6
<a href="#">Fabian Schultz</a>	Capítulo 14
<a href="#">Faktor 10</a>	Capítulo 5
<a href="#">ghostffcode</a>	Capítulo 19
<a href="#">Gianluca Esposito</a>	Capítulo 1
<a href="#">goldbullet</a>	Capítulo 2
<a href="#">GordyD</a>	Capítulo 2
<a href="#">hmnzr</a>	Capítulo 2
<a href="#">Inanc Gumus</a>	Capítulo 1
<a href="#">ivarni</a>	Capítulo 2
<a href="#">Jack7</a>	Capítulo 5
<a href="#">Jagadish Upadhyay</a>	Capítulo 5
<a href="#">Jason Bourne</a>	Capítulo 14
<a href="#">Jim</a>	Capítulo 29
<a href="#">JimmyLv</a>	Capítulos 5 y 20
<a href="#">John Ruddell</a>	Capítulos 3 y 6
<a href="#">jolyonruss</a>	Capítulos 1 y 2
<a href="#">Jon Chan</a>	Capítulos 1 y 2
<a href="#">jonathangoodman</a>	Capítulo 2
<a href="#">JordanHendrix</a>	Capítulos 1 y 2
<a href="#">juandemarco</a>	Capítulo 1
<a href="#">justabuzz</a>	Capítulo 2
<a href="#">Kaloyan Kosev</a>	Capítulos 13, 15 y 21
<a href="#">Kousha</a>	Capítulos 2 y 4

<a href="#">leonardoborges</a>	Capítulo 13
<a href="#">Leone</a>	Capítulo 3
<a href="#">Iustoykov</a>	Capítulo 17
<a href="#">Maayan Glikser</a>	Capítulo 2
<a href="#">Mark Lapierre</a>	Capítulo 16
<a href="#">MaxPRafferty</a>	Capítulos 1 y 5
<a href="#">Mayank Shukla</a>	Capítulo 16
<a href="#">McGrady</a>	Capítulo 14
<a href="#">Md Sifatul Islam</a>	Capítulo 1
<a href="#">Md. Nahiduzzaman Rose</a>	Capítulo 1
<a href="#">Michael Peyper</a>	Capítulos 2 y 13
<a href="#">Mihir</a>	Capítulo 8
<a href="#">MMachinegun</a>	Capítulo 1
<a href="#">m_callens</a>	Capítulo 2
<a href="#">Nick Bartlett</a>	Capítulo 1
<a href="#">orvi</a>	Capítulo 1
<a href="#">parlad neupane</a>	Capítulo 8
<a href="#">Qianyue</a>	Capítulo 13
<a href="#">QoP</a>	Capítulos 4, 5 y 6
<a href="#">Rajab Shakirov</a>	Capítulo 3
<a href="#">Dhruv Kumar Jha</a>	Capítulo 11
<a href="#">Rene R</a>	Capítulo 30
<a href="#">Rifat</a>	Capítulo 20
<a href="#">Robeen</a>	Capítulo 10
<a href="#">rossipedia</a>	Capítulo 1
<a href="#">Salman Saleem</a>	Capítulo 6
<a href="#">Sammy I.</a>	Capítulo 26
<a href="#">Sergii Bishyr</a>	Capítulo 5
<a href="#">Shabin Hashim</a>	Capítulo 1
<a href="#">Shuvo Habib</a>	Capítulo 9
<a href="#">Simplans</a>	Capítulo 1
<a href="#">sjmarshy</a>	Capítulo 2
<a href="#">skav</a>	Capítulos 4 y 6
<a href="#">sqzaman</a>	Capítulo 13
<a href="#">Sunny R Gupta</a>	Capítulos 1 y 14
<a href="#">thibmaek</a>	Capítulo 27
<a href="#">Timo</a>	Capítulos 1, 4, 6 y 7
<a href="#">user2314737</a>	Capítulo 1
<a href="#">vintproykt</a>	Capítulos 23 y 24
<a href="#">Vivian</a>	Capítulo 6
<a href="#">Vlad Bezden</a>	Capítulo 2
<a href="#">WitVault</a>	Capítulos 5 y 6
<a href="#">Zac Braddy</a>	Capítulo 12
<a href="#">Zakaria Ridouh</a>	Capítulo 2