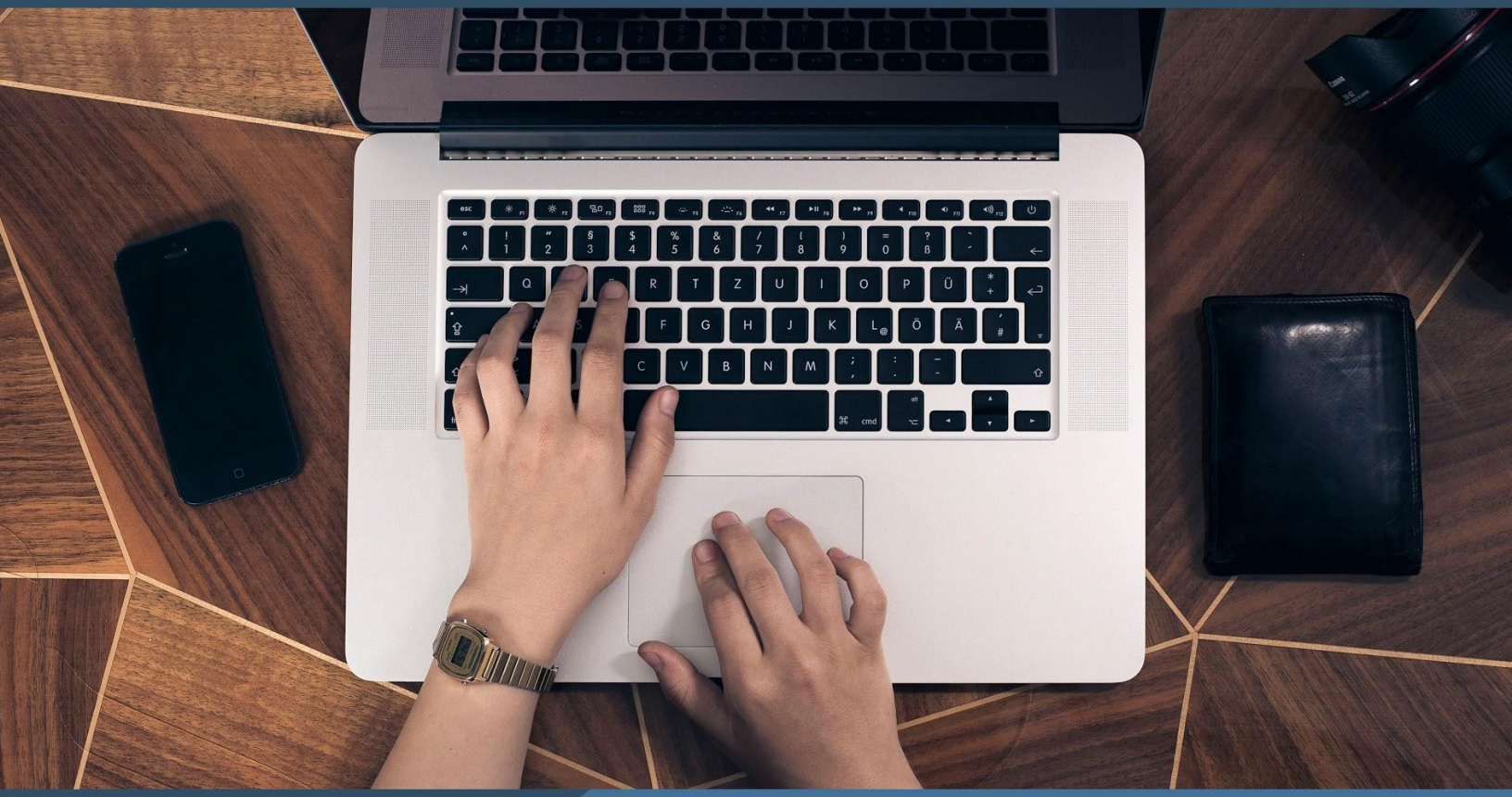

TU PRIMERA APP CON REACT.JS EN 10 MINUTOS

CARLOSAZAUSTRE.ES

¡BIENVENID@!

¡Hola! Muchas gracias por descargarte esta guía con la que espero sea tu primera andadura con React.js

Por si no me conoces, me presento. Soy Carlos Azaustre y llevo más de 6 años dedicado al desarrollo web y los últimos centrado en el Frontend y en JavaScript en general.

En el último año he cambiado mi stack favorito por React.js, una librería que me encanta por su simpleza y por el gran eco-sistema que hay alrededor: Webpack, React Native, Redux, GraphQL, etc..

React es solo la puerta de entrada y quiero abrirte paso a este mundo con esta guía.

¿QUÉ VAS A ENCONTRAR EN ESTA GUÍA?

En esta guía voy a enseñarte rápidamente como desarrollar con React.js una webapp muy sencilla sin meterme en workflows de trabajo, librerías de terceros ni configuración de editores de texto.

Vamos a “codear” directamente en el navegador para que veas como funcionan los principios básicos de React.

¡Vamos a ello!

PASO 1: Librerías a utilizar

Primero de todo vamos a abrir en el navegador (preferiblemente Chrome) la página JSBin que te provee de un editor online en el que escribir HTML, CSS y JavaScript. Utilizaremos esto para desarrollar nuestra app.

En la parte HTML es dónde vamos a insertar nuestros componentes de React y también, por medio de tags `<script>` añadiremos las librerías que vamos a necesitar.

Las librerías que vamos a emplear son `react.js` evidentemente y `react-dom.js` que nos permite manejar el DOM del documento, ya que que React puede ser usado también desde el servidor, pero en este caso nos vamos a centrar en el cliente.

Y en este caso, vamos a utilizar también `babel.js` que nos va a permitir utilizar toda la nueva sintaxis y funcionalidades que trae ECMAScript 6 (o ES2015).

Esta librería sólo debe utilizarse en un entorno de desarrollo. Para producción lo ideal es que utilices Babel como *transpilador* y mediante empaquetadores como *Webpack* o *Browserify*, crear un fichero final listo para producción.

Por si no conoces Babel, como digo, es un *transpilador* que convierte el código JavaScript que escribimos y que aún no es soportado por los navegadores, a una versión que si conozcan. Además incluye plugins que permiten transformar JSX (El lenguaje que utilizaremos en React para crear componentes) a JavaScript sin que tengas que preocuparte por nada.

Por tanto, nuestro fichero HTML tendrá que incluir los siguientes scripts antes del cierre del `</body>` incluyendo las librerías que estarán alojadas en un CDN:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.3.2/react.min.js" />
<script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.3.2/react-dom.min.js" />
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-standalone/6.17.0/babel.min.js"/>
```

Seguidos de estos scripts, vamos a añadir un link a nuestro fichero app.js que es donde tendremos toda la lógica de la aplicación. Como este fichero contendrá código en ES2015, en lugar de ser un script del tipo `text/javascript` es de tipo `text/babel` y así la librería de *babel* lo sabrá leer para que el navegador lo pueda ejecutar

```
  <script type="text/babel" src="app.js" />
```

Además debemos incluir al menos un `<div>` con un ID, para poderlo llamar después con *React-DOM*

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

En la siguiente página tienes el código *HTML* completo, para que no te pierdas.

Ésta es la pinta que tendría el fichero index.html y que ya no volveremos a tocar

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Mi primera App React</title>
  </head>
  <body>
    <div id="app"></div>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.3.2/react.min.js"/>
   <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.3.2/react-dom.min.js"/>
   <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-standalone/6.17.0/babel.min.js"/>
    <script type="text/babel" src="app.js"></script>
  </body>
</html>
```

PASO 2: Componente principal

Ahora que tenemos el HTML listo, vamos a pasar al código JavaScript. En la pestaña de JavaScript, vamos a elegir la opción *JSX (React)* para que todo funcione correctamente.

El componente que vamos a crear se llamará `<App>` y tiene la siguiente pinta:

```
class App extends React.Component {
    render () {
        return (<h1>Mi primera App</h1>);
    }
}

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

Creamos una clase de ES2015 con el nombre “App” y que herede o extienda de `React.Component.` Anteriormente se utilizaba la función `React.createClass` para crear componentes, pero con la nueva versión de JavaScript, es más correcto y cómodo utilizar la sintaxis de *Clase*.

Dentro del método `render` de la clase, estamos escribiendo código JSX, muy parecido a HTML pero no lo es. Esto simplemente indica que representación visual tendrá éste componente en el DOM.

Con `ReactDOM.render`, le estamos pasando como primer argumento el componente que vamos a renderizar, y como segundo, el elemento HTML del DOM donde lo vamos a insertar.

Si realizas esos cambios, verás en la pestaña de *output* el resultado que muestra un título con H1: Mi primera App.

PASO 3: Añadir componentes Hijo

No vamos a meter toda la lógica de nuestra app en el mismo componente. Lo bueno de React y la programación orientada a componentes es que nos permite modularizar en pequeños pedazo nuestra aplicación.

Por tanto, vamos a crear más componentes y anidarlos unos dentro de otros.

La app que vamos a desarrollar como ejemplo, va a mostrar un listado de empleados, y para cada uno de ellos utilizaremos un componente reutilizable que lo represente

```
class Empleado extends React.Component {
     render () {
          return (<li>{this.props.nombre - this.props.email}</li>);
     }
}
```

Hemos creado una nueva clase “Empleado” que representará al componente y su método `render` devuelve un tag `<li>` con dos variables: `this.props.nombre` y `this.props.email.` Estas variables son propiedades (o `props`) y se reciben del componente padre (Ahora lo veremos). Es la forma de pasar información de componentes “padres” a componentes “hijos”

PASO 4: Anidando componentes

Ya tenemos un componente `<Empleado />` con dos propiedades *email* y *nombre*. Ahora vamos a crear otro componente: `<ListaEmpleados />` cuya función será renderizar una lista de componentes `<Empleado />`:

```
class ListaEmpleados extends React.Component {
     render () {
          return (
            <ul>
              {this.props.empleados.map(empleado => {
                return (
                  <Empleado
                    key={empleado.id}
                    nombre={empleado.nombre}
                    email={empleado.email}
                  />
                );
               })}
             </ul>
          );
     }
}
```

Este componente parece complicado, pero no lo es. Te explico.

Lo único que hacemos es devolver un tag `<ul>` (una lista) y con el método `map` de JavaScript recorremos un array `this.props.empleados`, es decir un array `empleados` que vamos a recibir por las propiedades desde el padre.

La sintaxis `this.props.empleados.map(empleado => { ... })` es una *arrow function* de ES2015, viene a ser lo mismo que si escribiéramos: `this.props.empleados(function (empleado) { ... })` pero de una forma más limpia

Y ese array cuando se recorre, devuelve por cada ítem, un componente `<Empleado>` al que le pasamos la información (nombre y email) por propiedades.

En React, si estamos mostrando una lista, es conveniente que cada componente item, tenga una propiedad `key` con un valor, para poderlo identificar y React lo pueda renderizar de forma más eficiente.

Ahora vamos a modificar el componente App, para añadir ese array de empleados y pasárselo al componente `<ListaEmpleados>.`

Para ello vamos a hacer uso del estado de los componentes en React.

PASO 5: Manejando el estado

Los componentes en React, además de tener propiedades, pueden tener estado.

El estado es un objeto inmutable que tiene la particularidad de que si es modificado, el método `render` de ese componente se dispara de nuevo, forzando así un *re-renderizado* del componente en el navegador con la nueva información recibida.

Por tanto, vamos a modificar un poco el componente *App.js* para que tenga estado y su estado inicial contenga el array de objetos empleado.

Las clases de JavaScript tienen un método `constructor()` dónde se inicializa el componente. Como los componentes "heredan" de `React.Component` tenemos que invocar el método `super()` para que también se llame al constructor del padre.

Para que el estado tenga repercusión en el renderizado, debemos modificar el método `render`, y a la propiedad `empleados` pasarle el valor de `this.state.empleados`.

El componente `<App>` modificado sería así:

```
class App extends React.Component {
    constructor() {
        super();
        this.state = {
            empleados: [
                {id: 1, nombre: "Pepe", email: "pepe@correo.com"},
                {id: 2, nombre: "Paco", email: "paco@email.es"}
            ]
        };
    }

    render () {
        return (
            <ListaEmpleados empleados={this.state.empleados} />
        );
    }
}
```

¿Para que nos sirve utilizar en el método *render* `this.state.empleados`? Imagina que quieres añadir nuevos "empleados" a la lista. Al tener vinculado el estado con el *render*, cada vez que se modifique ese estado (añadiendo un nuevo empleado) el componente se va a "repintar" y aparecerán los nuevos datos en el navegador.

PASO 5: Propagación de eventos

Así pues, vamos a implementar esa funcionalidad paso a paso. Primero vamos a modificar el componente `<ListaEmpleados>` para añadirle un formulario que recoja estos datos, tal que así:

```
class ListaEmpleados extends React.Component {
     render () {
          return (
               <div>
                 <ul>
                   ...
                 </ul>
                 <form onSubmit={this.props.onAddEmployee}>
                    <input type="text" placeholder="Nombre" name="nombre"
                    <input type="email" placeholder="Email" name="email" />
                    <input type="submit" value="Guardar" />
                 </form>
               </div>
          );
     }
}
```

Lo que hemos hecho es añadir un elemento `<form>` que contiene dos `<input>` uno para recoger el nombre y otro para recoger el email. Por último un *input* de tipo *submit* que disparará el evento `onSubmit` que tiene `<form>` y que proporciona React con JSX.

Si te fijas, `onSubmit` llama a una función `this.props.onAddEmployee`. Esto quiere decir que la función que realmente maneja este evento se lo estamos proporcionando a través de las propiedades del componente, y por tanto será el componente padre el que la gestione.

Al contrario que los datos, que fluyen de padres a hijos a través de las propiedades, los eventos fluyen hacia arriba. Son disparados por un hijo y el padre puede recogerlo y así modificar lo que sea necesario.

Hacemos esto porque el formulario se encuentra en el componente `<ListaEmpleados>` pero el array donde están los datos se encuentra en `<App>` y es de esta manera como podemos comunicarnos.

Por tanto, lo que toca ahora es modificar el componente <App> para que recoja ese evento, lo maneje y cambie el estado:

```
class App extends React.Component {
    constructor() {...}
     render () {
        return(
           <ListaEmpleados
              empleados={this.state.empleados}
              onAddEmployee={this.handleOnAddEmployee.bind(this)}
           />);
     }
}
```

He añadido una nueva propiedad `onAddEmployee` que es la que recibe el componente hijo, y ésta propiedad llama a una función de `<App>` que ahora implementaremos llamada `handleOnAddEmployee`. A esta función le añadimos el método `bind` y le pasamos el objeto `this`.

Esto lo hacemos porque dentro de `handleOnAddEmployee` vamos a usar `this` y si no está "bindeado" no podremos usarlo correctamente.

Pasamos ahora a desarrollar la función `handleOnAddEmployee`. Esta función va a recibir por parámetros un objeto `event`, el cuál trae la información que tengan los *input* del formulario.

Por tanto, primero usaremos el método `preventDefault()` para evitar que la página se recargue por ser el comportamiento por defecto de un formulario.

Después recogeremos el valor del campo `nombre` y del `email`, y actualizaremos el estado.

Para actualizar el estado puedes estar tentado de hacer un `push` al *array* de *empleados*, pero recordemos que el estado debe ser inmutable y la función `push` modifica el *array* desde el que se invoca. Por tanto, vamos a usar otra función, `concat`, que no modifica el *array* si no que devuelve otro nuevo. De esta manera nuestro estado no muta y es más funcional, testeable y fácil de controlar.

En la siguiente página te muestro el código de todo esto:

```
handleOnAddEmployee (event) {
    event.preventDefault();
    let empleado = {
        nombre: event.target.nombre.value,
        email: event.target.email.value
    };
    this.setState({
        empleados: this.state.empleados.concat([empleado])
    });
}
```

Como puedes ver, para actualizar el estado usamos el método `setState` y al objeto "*empleados*", le añadimos el nuevo "*empleado*".

Cuando esto ocurra, como hemos dicho, llamará al método `render` y volverá a *renderizar* el componente `<ListaEmpleados>` con el nuevo empleado.

Para que no tengas dudas, En este enlace te dejo el código completo de la aplicación para que lo pruebes y también en este Gist de Github.

¡CUÉNTAME TU OPINIÓN!

¿Qué te ha parecido? ya conoces el funcionamiento básico de React y te sirve como punto de partida para empezar a profundizar con esta librería.

Ten en cuenta que React es mucho más y se pueden hacer mucho más cosas y más interesantes. Esta guía es tan solo el inicio para que le pierdas el miedo y te animes a probarla.

No te preocupes si no entiendes mucho de esta guía, como todo inicio con una nueva librería, framework o lenguaje, los comienzos cuestan. En los próximos días recibirás algunos emails extra en los que te voy a seguir explicando cosas del mundo React.

Puedes contarme qué te ha parecido, escribiéndome a **hola@carlosazaustre.es**

Si has recibido esta guía por otro medio, te recomiendo que te suscribas a mi lista para recibir los artículos extra.

SOBRE EL AUTOR DE ESTA GUÍA

Me llamo Carlos Azaustre y soy desarrollador web. Soy un amante de JavaScript y actualmente soy CTO en Chefly a la par que comparto conocimiento con la comunidad a través de mi blog y mi canal de Youtube y formo a otros desarrolladores a mejorar sus habilidades con JavaScript.

¡Nos vemos por las redes!

Un saludo,

Carlos
@carlosazaustre

¿TE HA GUSTADO ESTA GUÍA? DEVUÉLVEME EL FAVOR CON UN TWEET

Si te ha gustado esta guía, te voy a pedir el favor de compartirlo con tus amigos en Twitter.

Muchas gracias por ayudarme a difundirlo!

Gracias por la guía sobre #React @carloszaustre. Descarga la tuya y crea tu primera webapp desde aquí https://carlosazaustre.es/guia-react

Twitéalo