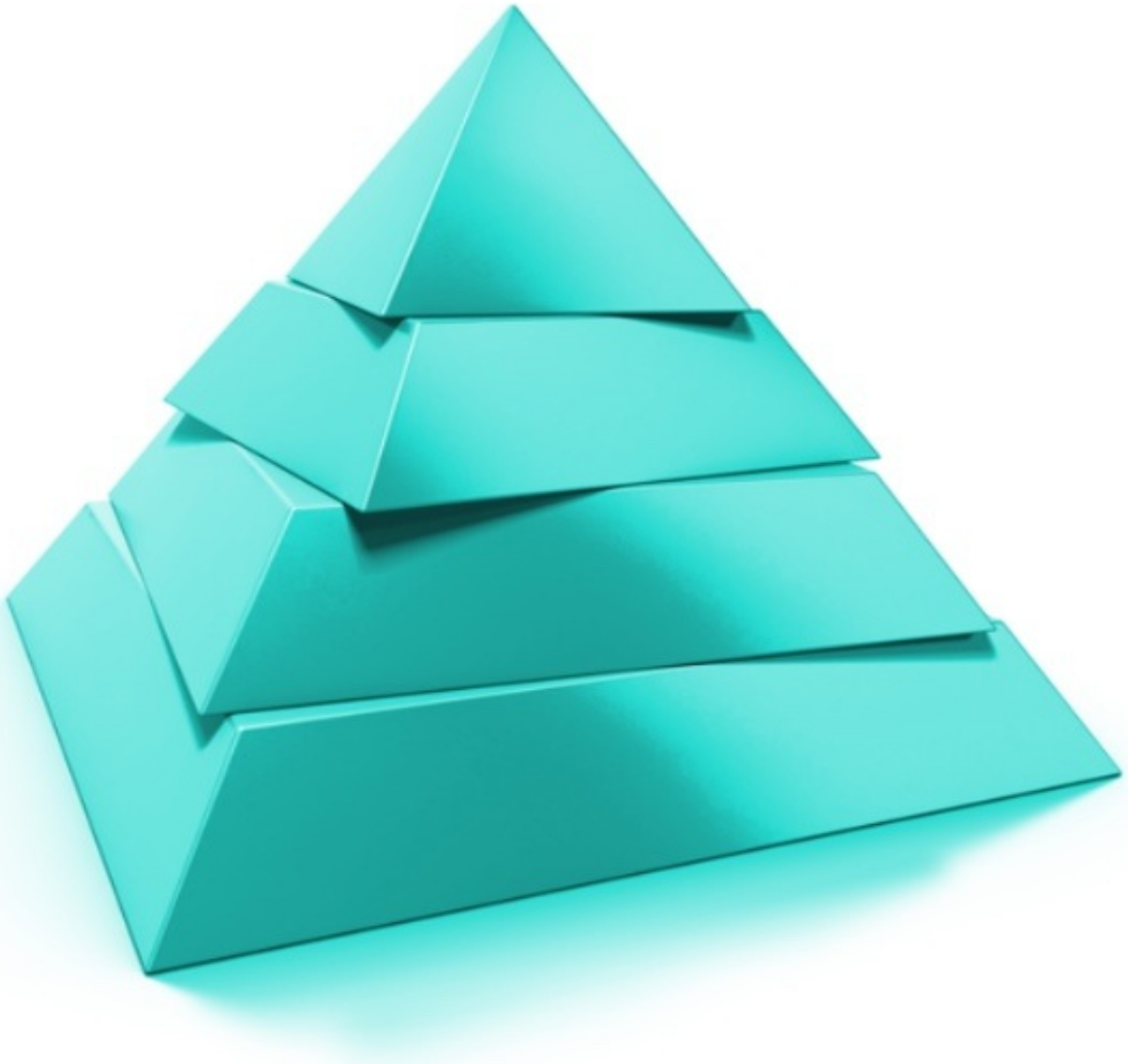
arquitecturajava

Desarrollo Web con React.js

Cecilio Álvarez Caules



www.arquitecturajava.com

Cecilio Álvarez Caules es Oracle Enterprise Architect, Sun Certified Business Component Developer, Sun Certified Web Component Developer y Sun Certified Java Programmer. Es además Microsoft Certified Solution Developer, Microsoft Certified Enterprise Developer y Microsoft Certified Trainer. Trabaja como Arquitecto, Consultor y Trainer desde hace más de 15 años para distintas empresas del sector.

Puedes seguirme a través de las siguientes redes

Twitter: @arquitectojava

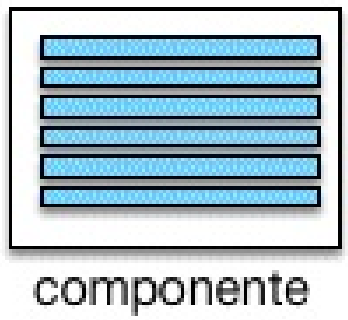
Introducción a React.js

React.js es una librería desarrollada por FaceBook orientada al desarrollo de componentes en **JavaScript**. No es nueva esta idea de desarrollar componentes: anteriormente frameworks como jQuery UI o Angular.js ya habían procedido de esta forma. Entonces, **¿Qué es lo que hace a React.js diferente?**

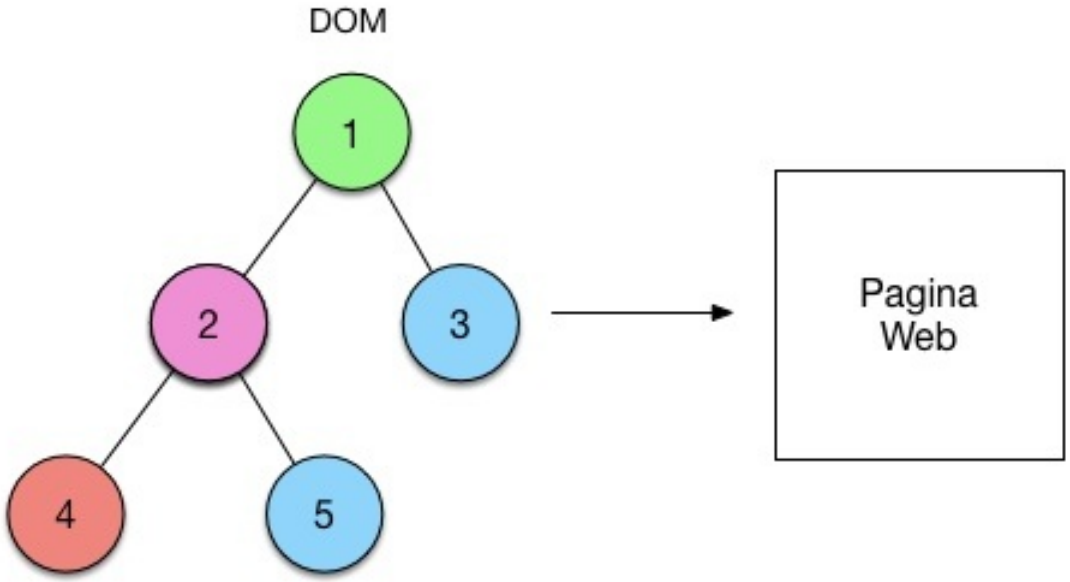
El concepto de Virtual DOM

React esta orientada a desarrollar componentes que permitan construir páginas c**on un gran rendimiento.** Para ello, se apoya en lo que habitualmente se denomina **Virtual DOM. ¿Qué es y como funciona el Virtual DOM?** Para entender esto, hay que comprender previamente dos conceptos:

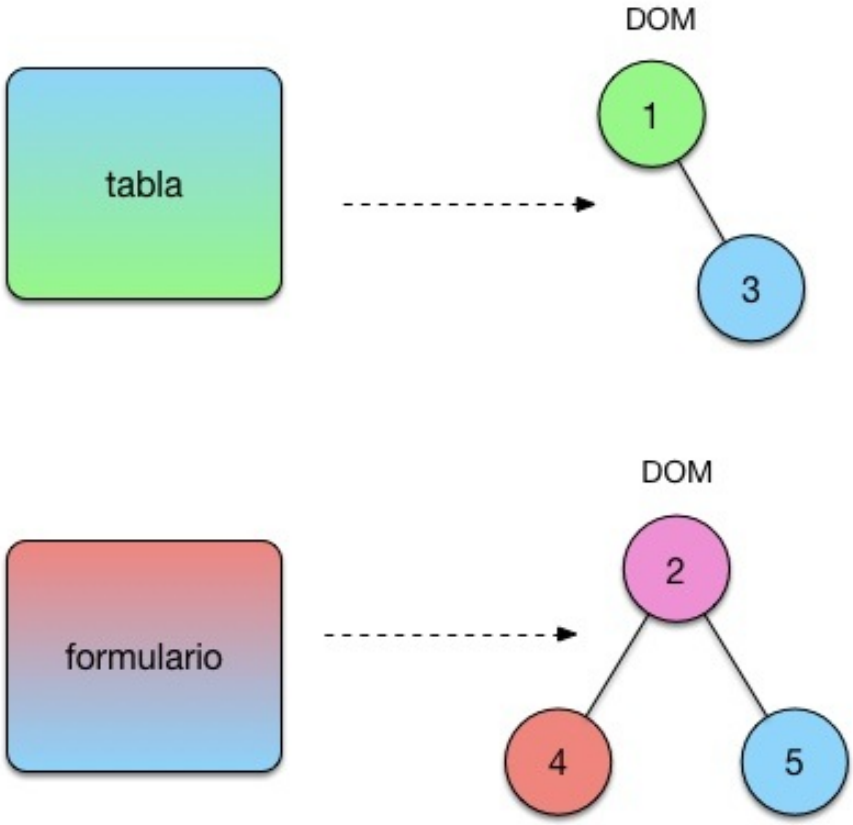
1) **Componente :** Un componente es u**n conjunto de etiquetas relacionadas de forma lógica que se renderiza en una página HTML** y permite su reutilización. Un ejemplo frecuente es un formulario concreto o una tabla.



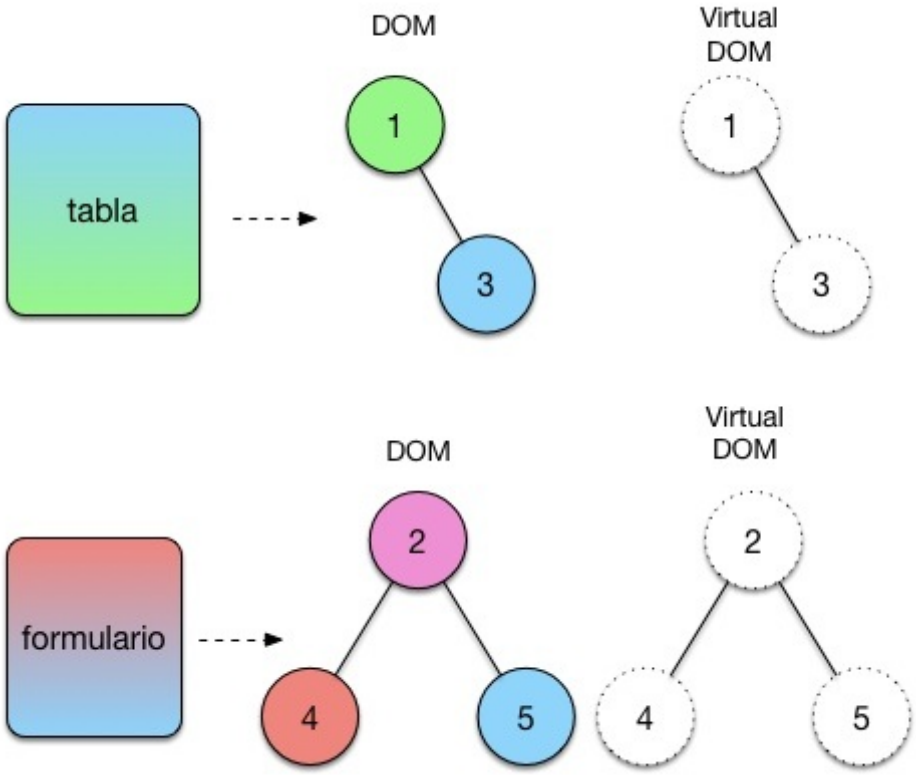
2) **DOM :** DOM o Document Object Model es una estructura (en la memoria) que define todas las etiquetas de una pagina HTML para su posterior renderizado en el navegador.



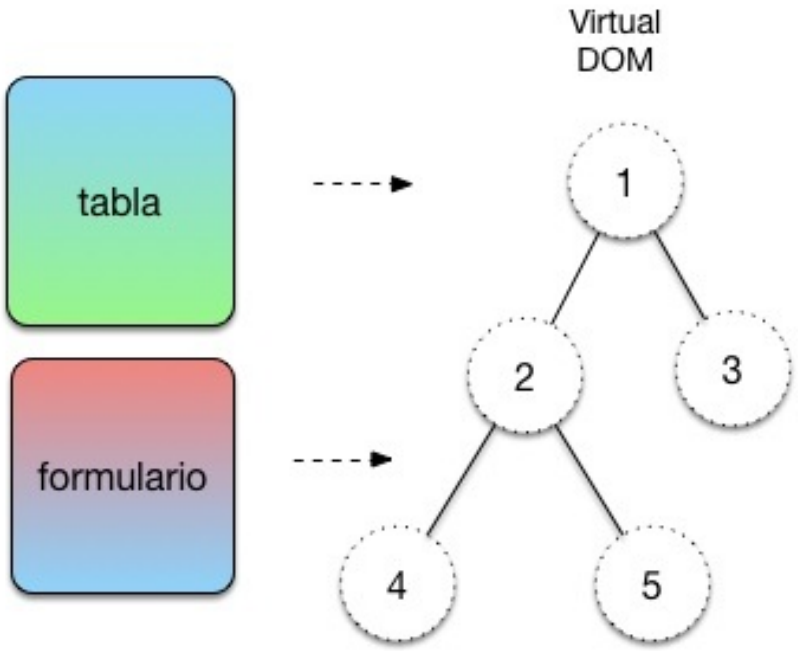
Al trabajar con el concepto de componente, veremos que cada uno de ellos está compuesto por un conjunto de etiquetas html o nodos DOM que se relacionan formando la página HTML



El concepto de **Virtual DOM hace referencia a un árbol DOM paralelo que se define y contiene una estructura simplificada del árbol original.**

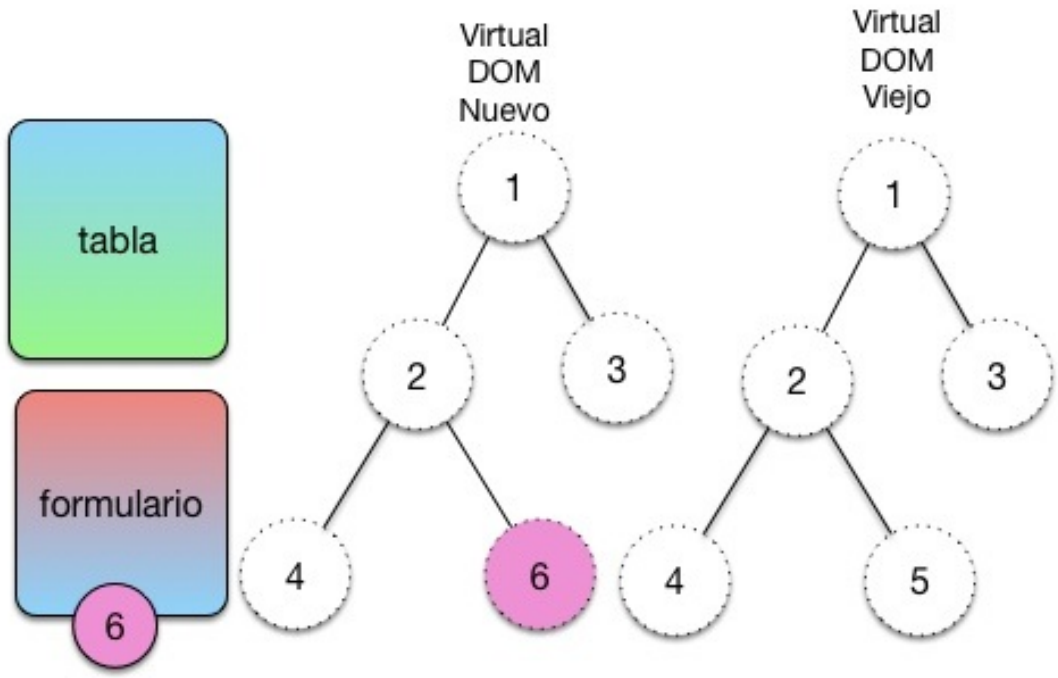


La unión de todos nuestros componentes y sus virtual DOMs da como resultado un VirtualDOM a nivel de la página HTML que representa de forma simplificada la estructura de dicha página.



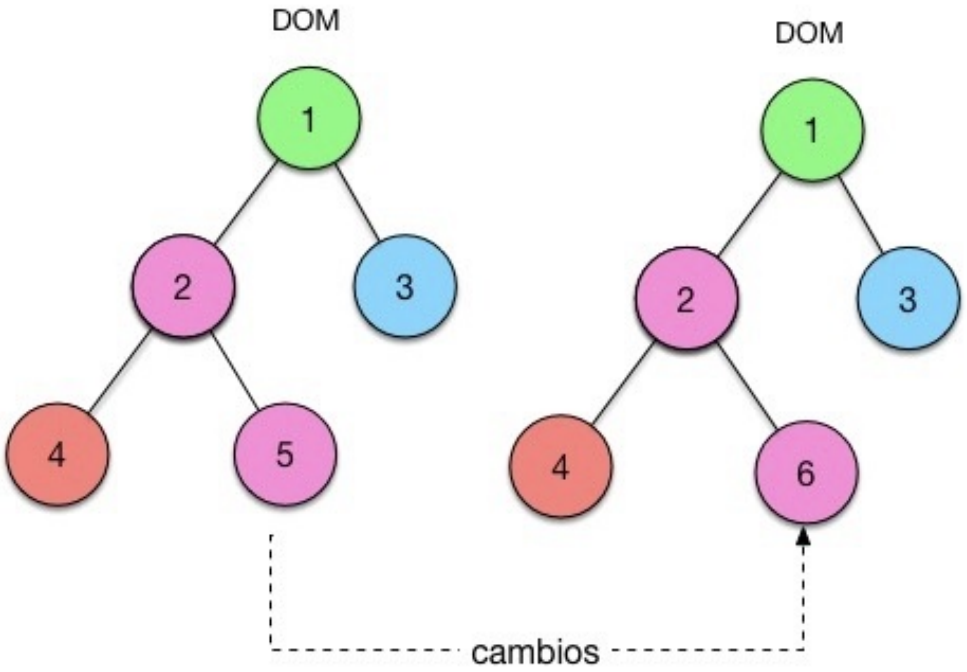
¿Para qué sirve este árbol virtual?. Cada vez que cambiemos algo en un componente ,React actualizará

el árbol virtual que tiene y generará una nueva versión.



Realizada esta operación React comparará los dos arboles **y calculará las diferencias entre ambos**. En nuestro caso únicamente se diferencia **en el nodo 6**. Así pues con ese pequeño cambio React se encargará de crear **un grafo de diferencias.**

Con el árbol de diferencias, React podrá **actualizar el árbol DOM real,** realizando los mínimos cambios imprescindibles.



Estos son los conceptos que React aporta como librería: innovadores a la hora de **incrementar el rendimiento de nuestras páginas y facilitar su reutilización** . Recordemos que cada día es mayor nuestro trabajo con arquitectura SPA **y las páginas cargadas son de gran tamaño**.
Así pues, entendidos los conceptos básicos, es momento de comenzar a crear unos ejemplos.

Hola mundo con React.js

Vamos a trabajar con React y construir el ejemplo de “Hola mundo”. Para ello, lo primero que debemos hacer es construir una página HTML **e incluir las referencias a las librerías de React.** Para añadir las referencias, simplemente accedemos a la página de React.

https://facebook.github.io/react/docs/installation.html

Asignadas las referencias, vamos a ver el primer ejemplo:

```
<!DOCTYPE html>
<html>
<head>
	<title>Hola React</title>
	<script src="https://unpkg.com/react@15/dist/react.js"></script>
	<script src="https://unpkg.com/react-dom@15/dist/react-dom.js"></script></head>
<body>
	<div id="zona">
	</div>
	<script type="text/javascript">
		ReactDOM.render(
			React.createElement("p", null, "hola react"),
			document.getElementById("zona")
		);
	</script>
</body>
</html>
```

Por ahora es muy poco código , pero **es un código bastante diferente de aquél con el que estamos habituados a trabajar.** Vamos a explicarlo detalladamente: React es una tecnología basada en la construcción de componentes. El ejemplo de hola mundo no es una excepción y la siguiente linea:

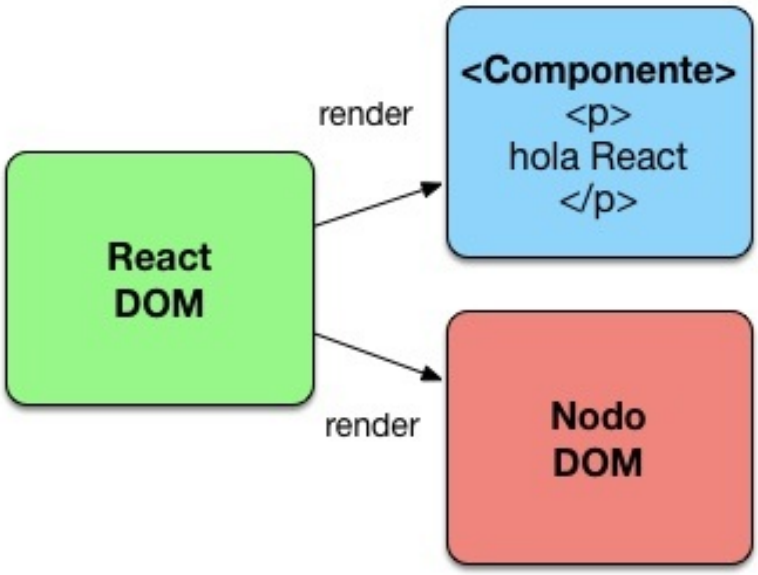
```
React.createElement("p", null, "hola react"),
```

Crea en memoria un componente de React, un párrafo con el texto de “**hola React”.**

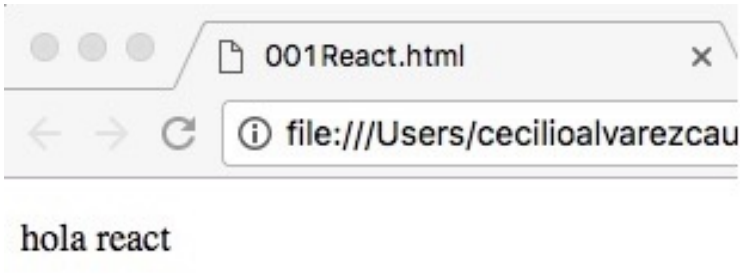
Nos queda revisar otra línea de código:

ReactDOM.render(……..)

Esta línea renderiza o dibuja en pantalla el componente. Si nos fijamos, el método render dispone de dos parámetros: **el primero es el componente que queremos dibujar en pantalla** , el segundo e**s el lugar o nodo DOM donde el componente se dibuja.**



Hemos **usado document.getElementById(“zona”) ,** haciendo referencia a un div que está dentro del body. Si cargamos la página en nuestro navegador, **veremos que nos imprime “hola react”**



Acabamos de construir nuestro primer componente con React .Queda explicar uno de los parámetros, que pasa un valor de null:

React.createElement("p", null, "hola react”),

Este valor existe para **que podamos asignar un estilo al componente** .Modificamos el código y aplicamos un estilo.

```
<script type="text/javascript">
      var Parrafo1 = React.createElement("p", {
         style: {
            color: "blue"
         }
      }, "hola react");

      ReactDOM.render(
         Parrafo1,
         document.getElementById("zona")
      );
   </script>
```

Si recargamos la página, el texto se muestra en color azul.

React y clases

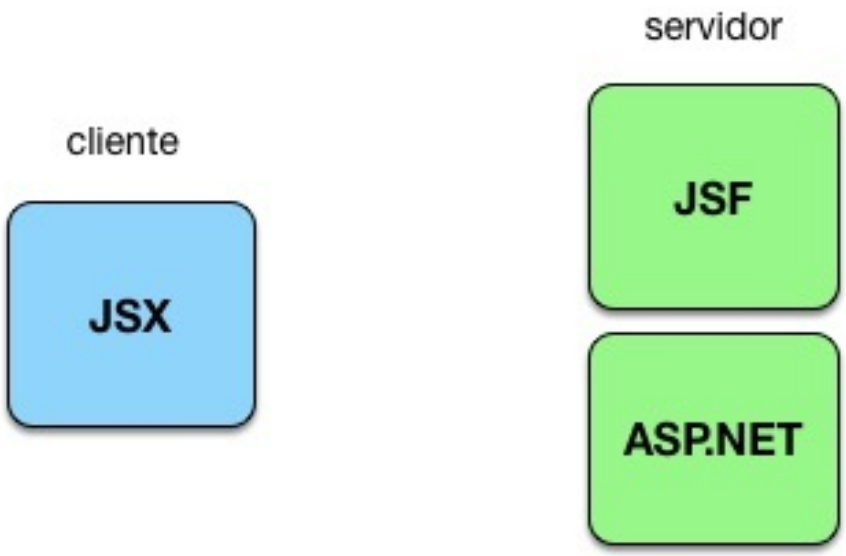
Si revisamos el código,nos daremos cuenta de que el Parrafo1 hace referencia más **a un elemento** que a un componente. En programación orientada a objeto, los componentes se definen con el **concepto de clase.** En React sucede lo mismo, así que nos queda hacer una última modificación para obtener un código adecuado**.**

```
<script type="text/javascript">
   var ParrafoAzul = React.createClass({
      render: function() {
        return  React.createElement("p", {
            style: {
               color: "blue"
            }
         }, "hola react");
      }
   });
   ReactDOM.render(
      React.createElement(ParrafoAzul),
      document.getElementById("zona")
   );
   </script>
```

El código funcionará de forma idéntica, sin embargo, ahora estamos usando clases para crear componentes: hemos terminado nuestro primer componente en React.

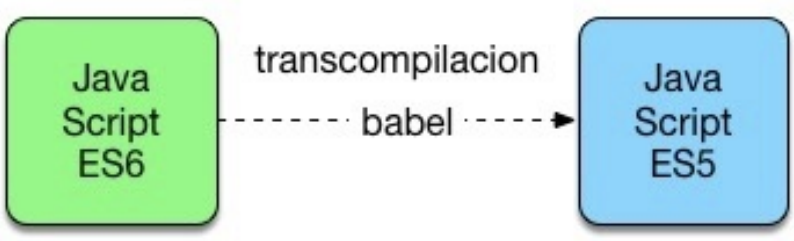
JSX y React

¿Qué es JSX? . Al querer trabajar con componentes a nivel de HTML, **es necesario incrementar el nivel de abstracción** . **¿Qué significa esto?** . En lugar de usar etiquetas como <p>,<ul>, <table> etc, usaremos etiquetas más complejas como <GridView> o <DataTable>, que aportan componentes predefinidos de mucha mayor funcionalidad. Esta idea no es nueva , de hecho **<GridView>** es una etiqueta de ASP.NET y **<DataTable>** es una etiqueta de Java Server Faces. Ambas son tecnologías de componentes. ¿Entonces qué aporta React con JSX? . La novedad consiste en **una diferencia relevante:** el sistema de controles se ejecuta en **el lado del cliente con su Virtual DOM** y no en el lado del servidor.



Configuración de JSX y Babel

Vamos a configurar JSX para nuestros ejemplos . Para ello, vamos a hacer uso de Babel. **¿Qué es Babel?**. Babel es un compilador de JavaScript que nos permite trabajar con JavaScript ES6 y compilar automáticamente a ES5 . A día de hoy, ES5 es el JavaScript soportado ampliamente en un navegador. Por otro lado, Babel soporta el uso de JSX como una extensión de JavaScript.



Vamos a realizar una configuración básica de Babel revisando GitHub :

https://github.com/babel/babel-standalone#usage

Es suficiente con añadir un script para poder trabajar con JSX:

<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-standalone/6.18.1/babel.min.js">
</script>

Vamos a crear nuestro primer ejemplo:

```
<!DOCTYPE html>
<html>

<head>
    <title></title>
    <script src="https://unpkg.com/react@15/dist/react.js"></script>
    <script src="https://unpkg.com/react-dom@15/dist/react-dom.js"></script></head>
   <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-standalone/6.18.1/babel.min.js">
</script>
</head>

<body>
    <div id="zona">
    </div>
    <script type="text/babel">

      ReactDOM.render(
        <p>hola react</p>,
        document.getElementById('zona')
      );

    </script>
</body>

</html>
```

El código se parece mucho a uno de los ejemplos anteriores, pero son destacables tres cambios importantes.

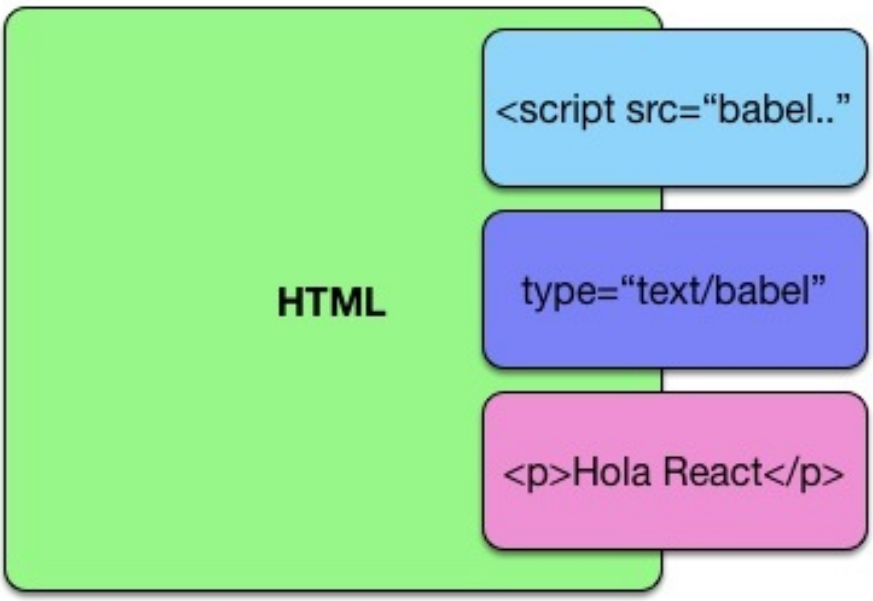
1. Tenemos q**ue añadir la dependencia de babel para poder trabajar con JSX** .

2. El script que construye los componentes cambia su propiedad type y pasa a ser **type="text/babel"**

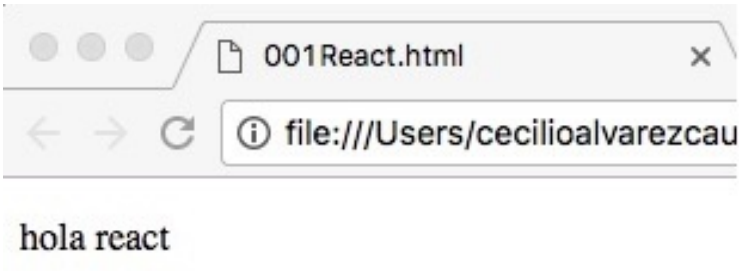
3. Ahora dentro del script podemos escribir:

hola React

Este último cambio puede parecer lógico ,pero al reflexionar vemos que n**o es un código Javascript válido.** En JavaScript **no se puede añadir etiquetas HTML**. En este caso podemos hacerlo porque el código ya no es JavaScript, **sino JSX.** Así pues, los cambios no son muchos pero sí significativos.



Si cargamos la página, el resultado será muy parecido al inicial:



JSX y Componentes

Aunque ya estamos usando JSX, no estamos todavía **bajo la filosofía de React ,pues no disponemos de ningún componente.** Es momento de cambiar nuestro código y construir un componente.

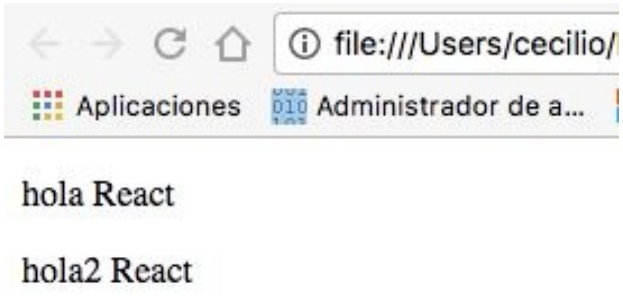
```
<script type="text/babel">
    var Parrafo = React.createClass({
```

```
        render: function() {

           return (<p>hola React</p>);
         }

    });

    ReactDOM.render(
    <Parrafo/> , document.getElementById('zona') );
</script>
```

El método de ReactDOM renderiza el componente “Parrafo”. Estamos usando JSX de la una forma adecuada. Creamos un segundo componente con **el mensaje “hola React2”.**

```
<script type="text/babel">
     var Parrafo = React.createClass({
     render: function() {
        return (<p>
            hola React
            </p>);
    }
    });

    var Parrafo2 = React.createClass({
      render: function() {
        return (<p>hola2 React</p>);
       }
     });
    ReactDOM.render(
    <div>
    <Parrafo/><Parrafo2/>
    </div> , document.getElementById('zona') );
</script>
```

Hemos añadido un que contiene ambos componentes a la hora de renderizar y así, React los imprime.



El objetivo de React **es construir componentes reutilizables** . En este ejemplo ambos componentes son reutilizables en poca medida. Será necesario revisar esto a fin de ganar en flexibilidad. Ahora bien, antes tenemos que instalar un servidor de node.

Node y React

A partir de este momento vamos a trabajar con todos los ficheros de JavaScript separados de la página html y descargados de un servidor.

Para ello necesitamos instalar Node.js en nuestra máquina descargándolo de :

http://nodejs.org/es

Una vez instalado Node, ejecutaremos:

npm install express
npm install body-parser

Nos instalará Express.js , un framework MVC de Node.js y BodyParser, una librería para gestionar JSON que necesitaremos posteriormente. El siguiente paso es crear un servidor web con node, del cuál nos podamos descargar ficheros. **Para ello creamos el fichero servidorBasico.js con este código:**

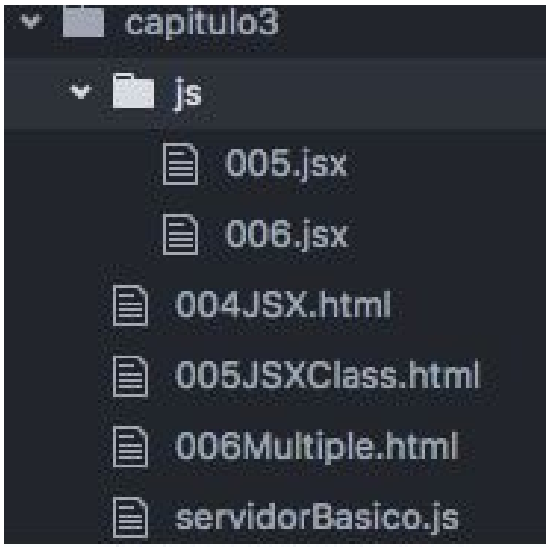
```
var express = require('express');
var app = express();
app.use(express.static('./'));
app.listen(3000, function() {
    console.log('servidor arrancado 3000');
});
```

Arrancamos el servidor desde linea de comandos con :

node servidorBasico.js .

Este servidor nos devolverá todas las paginas html que tengamos a su mismo nivel o sus subcarpetas. Así, ya podemos cargar las páginas usando http://localhost:3000/004JSX.html. y podemos diseñar una

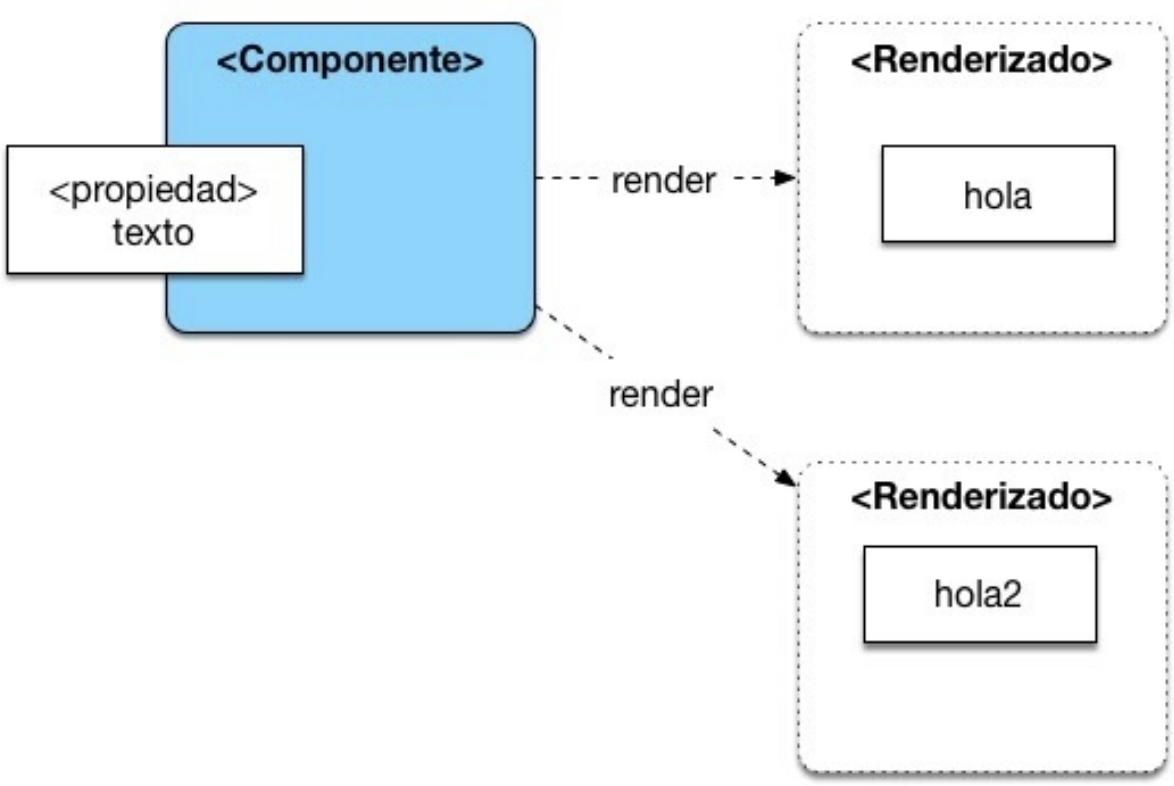
estructura similar a lo siguiente:



Ahora es momento de trabajar en profundidad con React.

React Properties y State

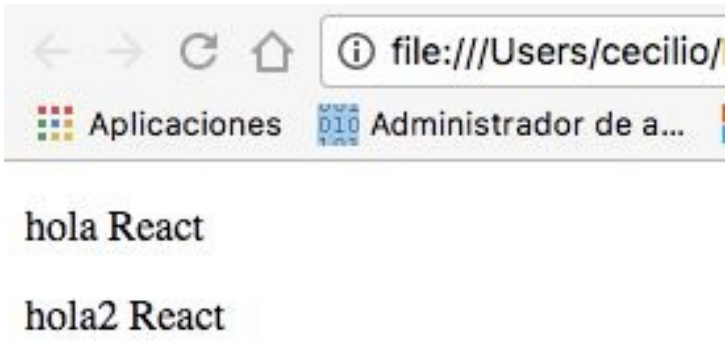
Todos los componentes de React pueden incluir propiedades como parámetros para añadir flexibilidad.



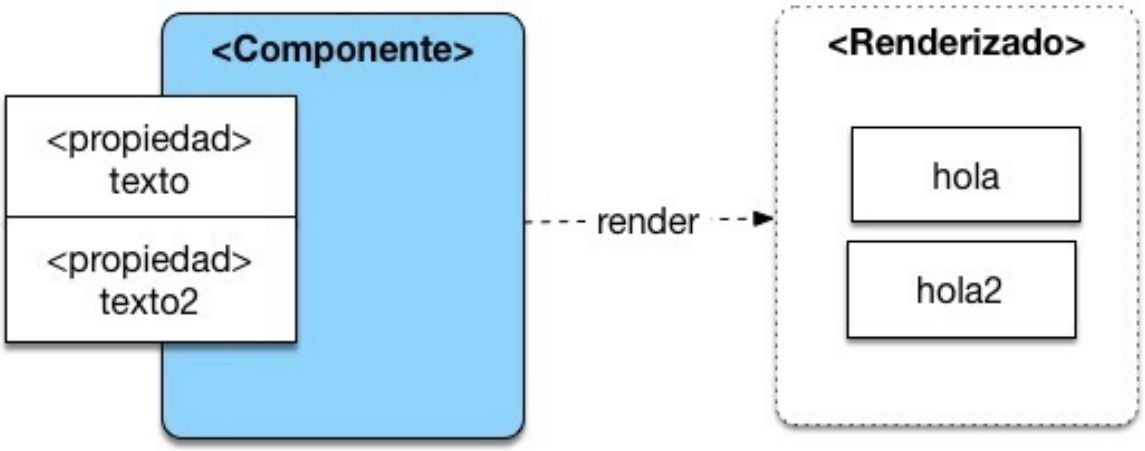
Es momento de cambiar nuestro componente para que incluya **una propiedad texto** y así, con un solo componente podamos renderizar tanto “hola” como “hola2”

```
var Parrafo = React.createClass({
   render: function() {
      return (
         <p>
            {this.props.texto}
         </p>
      );
   }
});

ReactDOM.render(
   <div>
   <Parrafo texto=“hola React"/>
   <Parrafo texto=“hola React 2"/>
</div>, document.getElementById(‘zona'));
```

Hemos creado nuestra primera propiedad y ahora con un único componente podemos mostrar ambos mensajes.



Tendríamos la posibilidad de modificar el componente para que reciba ambos mensajes a través de dos propiedades y los muestre.



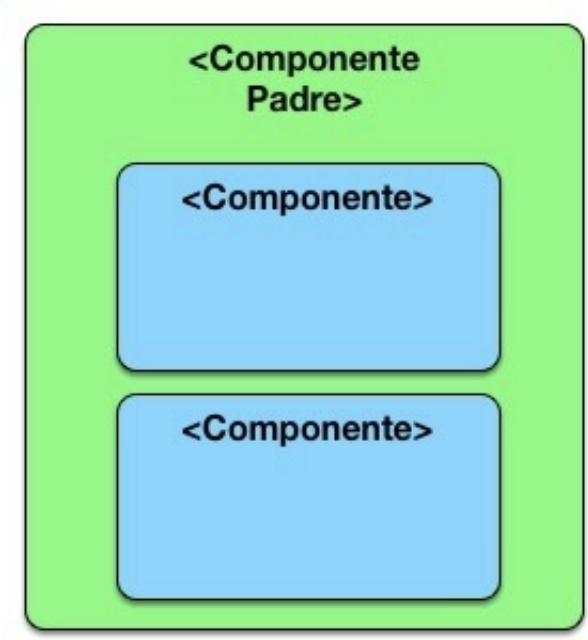
Vamos a ver el nuevo código:

```
var Parrafos = React.createClass({
   render: function() {
      return (
         <div>
            <p>
               {this.props.texto1}
            </p>

            <p>
               {this.props.texto2}
            </p>
         </div>
```

```
    );
  }

});

ReactDOM.render(
  <div>
  <Parrafos texto1="hola" texto2="react"/>
</div>, document.getElementById('zona'));
```

Componentes y anidamiento

Las modificaciones funcionan ,sin **embargo esta no es la forma más natural de trabajar** .En React lo que se intenta es reutilizar los componentes. Por lo tanto una solución más elegante sería construir **un componente "Parrafos" que incluya dos elementos de tipo "Parrafo".**



Modificamos el código :

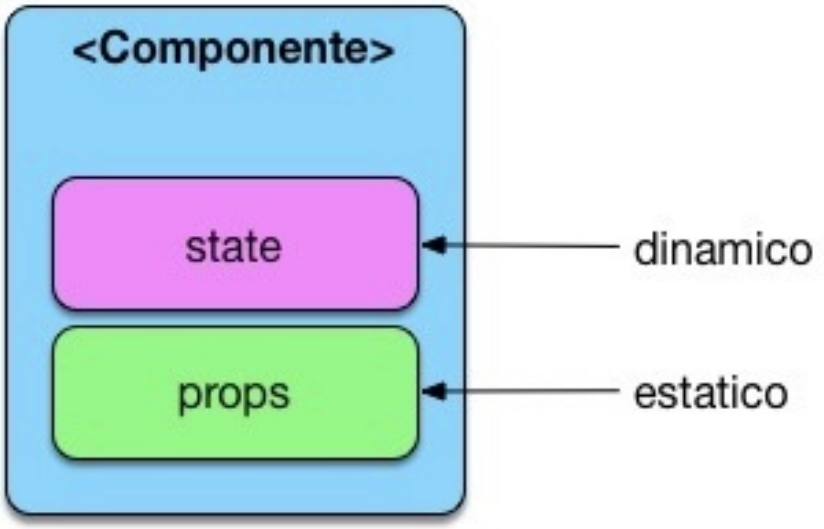
```
var Parrafo = React.createClass({
  render: function() {
    return (
      <p>
        {this.props.texto}
      </p>
    );
  }
```

```
});

var Parrafos = React.createClass({

   render: function() {

      return (
         <div>
            <Parrafo texto=“hola"/>
            <Parrafo texto="react"/>
         </div>);

   }
});
 ReactDOM.render(
   <div>
   <Parrafos/>
</div>, document.getElementById(‘zona'));
```

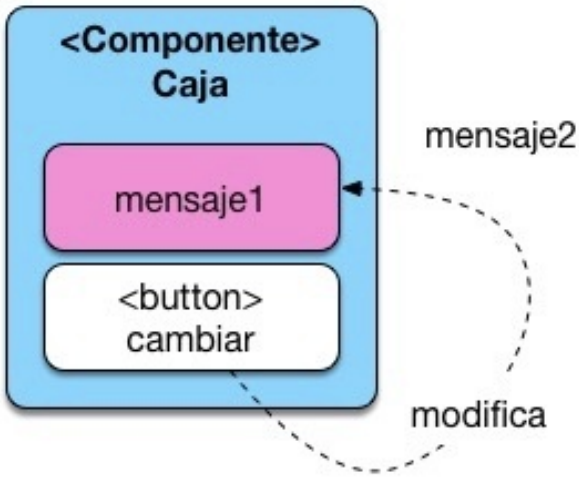
El resultado es similar **(si hubiéramos usado sería idéntico):**

React State

Hemos utilizado el concepto de propiedad para añadir flexibilidad a los componentes. Sin embargo, las propiedades **son conceptos estáticos y no pueden modificarse**. Esto es un problema ya que normalmente necesitamos cambiar **parte de la información que un componente almacena**. Para solventar este problema, React aporta **el concepto de state o estado.** El estado en React es la parte del componente con capacidad de almacenar información **que puede ser modificada.**



Vamos a construir un ejemplo en React que maneje el estado. Para ello, creamos un componente que contenga un párrafo con un mensaje y cuyo valor se pueda cambiar usando un botón.

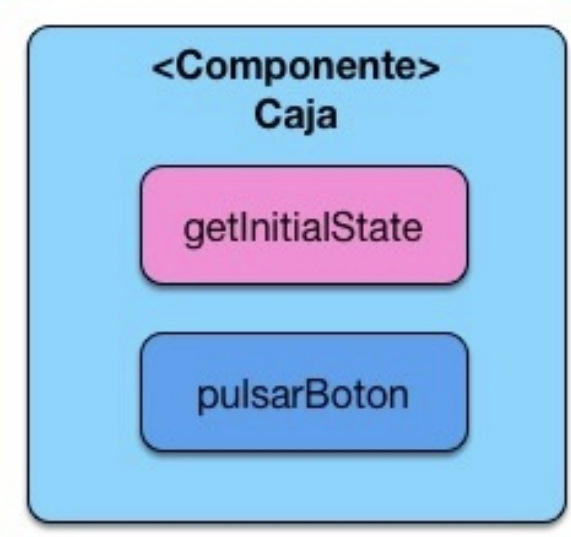


Veamos el código:

```
var CajaTexto = React.createClass({

   getInitialState:function() {

      return {texto:"mensaje1"};

   },
   pulsarBoton: function() {
```

```
      this.setState({texto:"mensaje2"});

    },
    render: function() {
        return (
          <div>
            <p>{this.state.texto}   </p>
             <input type="button" value="cambiar" onClick={this.pulsarBoton}/>

          </div>

        );
    }


});

ReactDOM.render(
    <div>
      <CajaTexto texto="mensaje1" />

      </div>,
    document.getElementById('zona'));
```

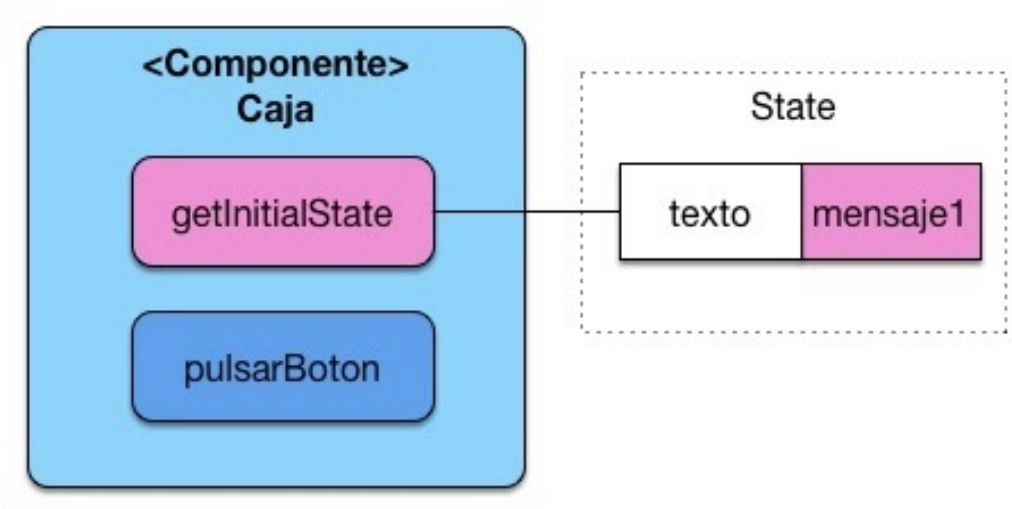
Podemos ver que es un componente diferente a lo que hemos creado anteriormente .Hasta estos momentos, nuestros componentes **únicamente tenían la función render** que servía para renderizarlos. Ahora sus capacidades se amplían, con dos funciones adicionales :**getInitialState() y pulsarBoton().**



El método **getInitialState** se encarga de inicializar el estado del componente.

```
getInitialState:function() {
    return {texto:"mensaje1"};
},
```

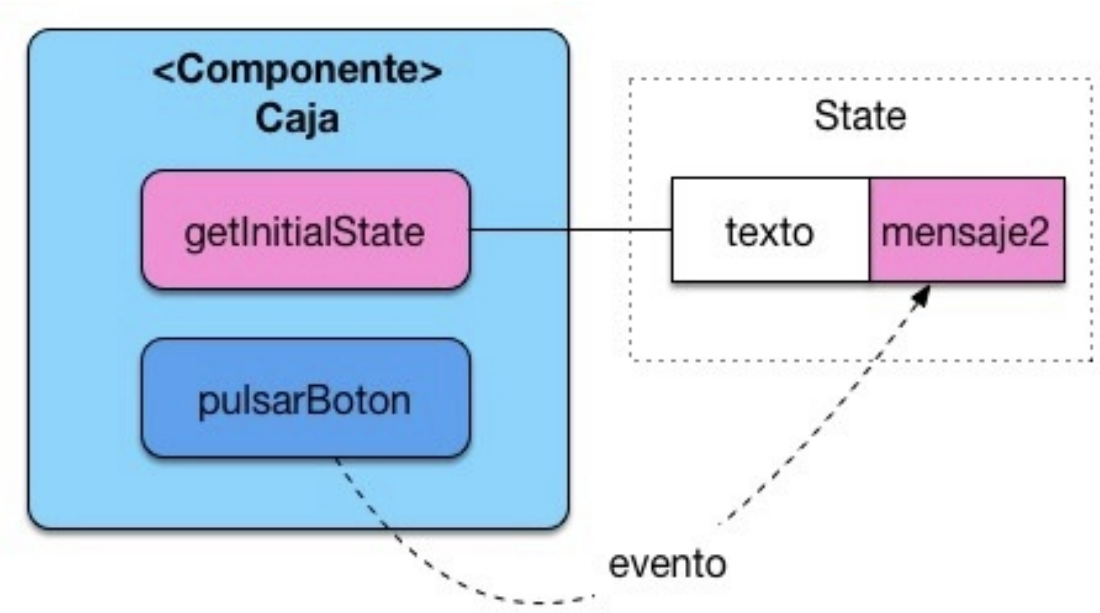
Este estado viene definido **por una estructura JSON y pertenece al componente** .



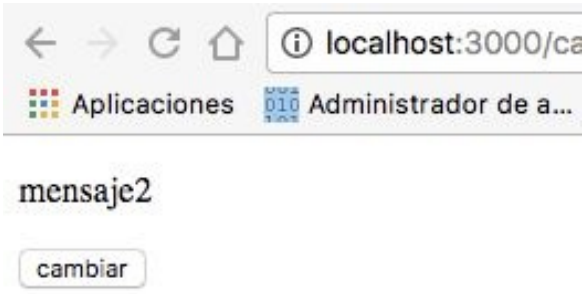
El método pulsarBoton es el encargado de modificar esta estructura y cambiar el texto de mensaje1 a mensaje2.

```
pulsarBoton: function() {
    this.setState({texto:"mensaje2"});
},
```

Al principio es difícil de entender cómo se implementa esta funcionalidad. En cuanto iniciamos el componente, una variable “texto” se instancia dentro de su estado. Al pulsar el botón de nuestro componente, éste se enlaza a **un** evento click asociado a la función “pulsarBoton” . Dicha función se encarga de cambiar la variable de texto almacenada en el estado del componente.



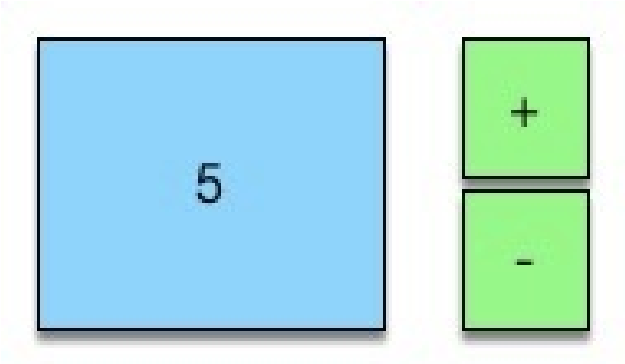
Si ejecutamos el código, veremos cómo al pulsar el botón cambia el contenido del párrafo de “mensaje1” a “mensaje2”.



Ya tenemos nuestro primer componente con estado y propiedades funcionando.

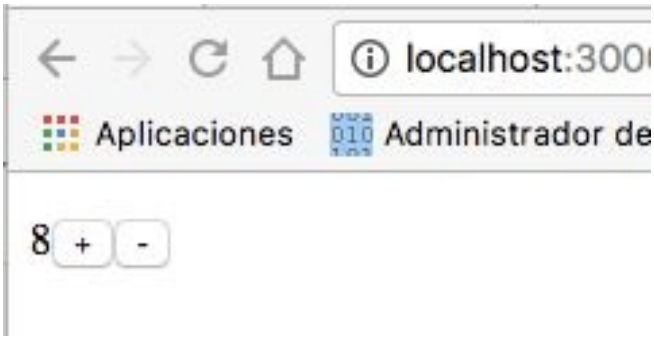
React y Componentes

Hemos visto una introducción a React , al concepto de Virtual DOM y al manejo de propiedades y estado . Es momento de avanzar y ver cómo construir **un componente más complejo**. Para ello ,construiremos un componente que nos muestre en pantalla la nota obtenida en un examen y que nos la pueda modificar mediante dos botones de + y -.



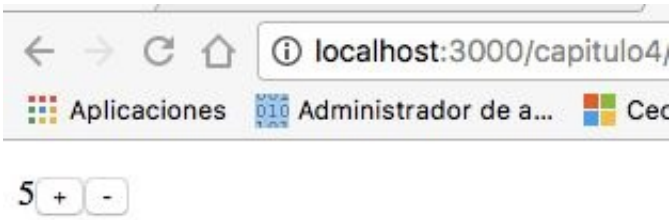
Vamos a construir una primera versión del código con los conceptos que hemos aprendido:

```
var Caja = React.createClass({

    getInitialState:function() {
      return {valor:5};
    },

    incrementa: function() {

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

    },
    decrementa: function() {

      this.setState({valor:this.state.valor-1});
    },
    render: function() {
        return (
            <p>
            {this.state.valor}
             <input type="button" value="+" onClick={this.incrementa}/>
             <input type="button" value="-" onClick={this.decrementa}/>
```

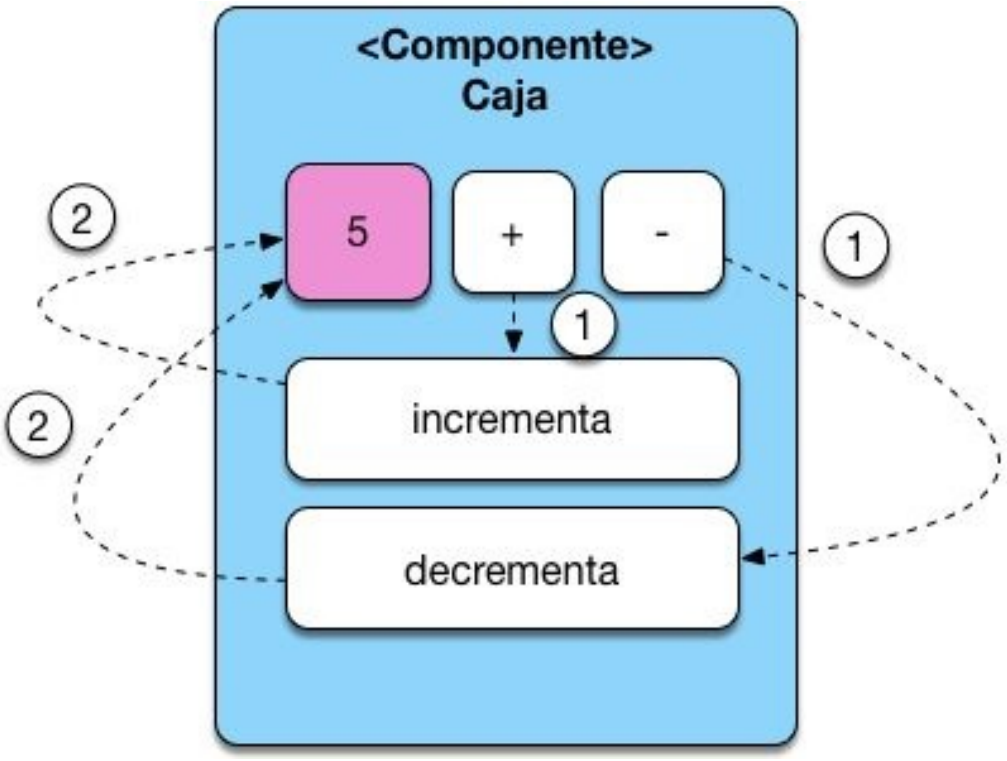



```
        </p>
      );
    },

});

ReactDOM.render(
    <div>
      <Caja/>
      </div>,
    document.getElementById('zona'));
```

Hemos definido el componente Caja y su funcionalidad . Ahora cargamos la página en el navegador.

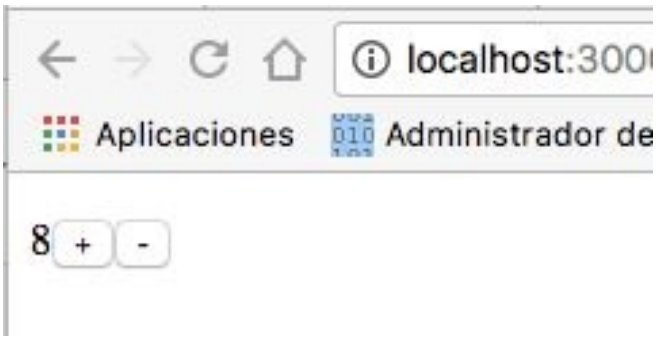


El componente almacena la nota como estado y el botón de + y - se encarga de cambiar el estado usando las funciones de incremente y decremento.

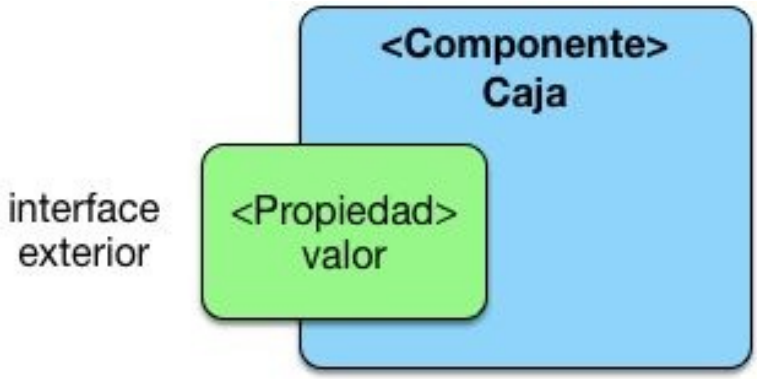


Así, podemos incrementar o hacer disminuir el valor de cada una según vamos pulsando uno u otro de los botones.

Componentes y reutilización



Ya tenemos un componente funcional, pero cuando uno construye un componente lo hace con la intención de que sea reutilizable y tenga entidad propia. En nuestro caso, el componente siempre comienza con un valor de 5 . Sería mucho mejor que este valor **pudiera ser parametrizado**. Vamos a modificar el componente para obligar al estado a tener un valor predefinido, asignado a través de una propiedad.



De alguna manera estamos **definiendo un interface hacia el exterior del componente**. Vamos a añadir una función de propTypes donde podemos definir las propiedades de un componente y definir si éstas son obligatorias :

```
propTypes: {

    valor: React.PropTypes.number.isRequired
  },
```

El siguiente paso es usar el valor de la propiedad para iniciar el estado del componente .Esto lo podemos hacer en el método getInitalState del componente:

```
  getInitialState: function() {

    return {valor: this.props.valor};

  },
```

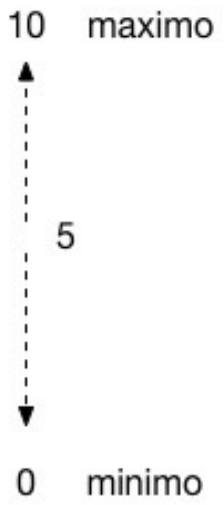
Realizadas estas modificaciones, ya podemos asignar al componente el valor inicial que deseemos.

```
ReactDOM.render(
   <div>
     <Caja valor={8}/>
     </div>,
   document.getElementById('zona'));
```

El navegador mostrará el componente con un valor inicial de 8:

Componente y Limites

Hemos dado otro paso adelante y ahora el componente **es más reutilizable**. El siguiente paso consiste en añadir más funcionalidad al componente. **Una opción sencilla es añadir límites, de tal forma que el valor no pueda superar unos rangos determinados.**



Así, modificamos el componente para añadir estos límites .Para ello, creamos 2 propiedades en su interface pública denominadas “valorMaximo” y valorMinimo” y modificamos las funciones de incremento y decremento.

Vamos a ver el nuevo código:

```
var Caja = React.createClass({

   propTypes: {

      valor: React.PropTypes.number.isRequired,
      valorMinimo: React.PropTypes.number,
      valorMaximo: React.PropTypes.number
   },

   getInitialState: function() {

      return {valor: this.props.valor};
```

```
  },

  render: function() {
    return (
      <p>
        {this.state.valor}
        <input type="button" value="+" onClick={this.incrementa}/>
        <input type="button" value="-" onClick={this.decrementa}/>
      </p>
    );
  },
  incrementa: function() {

    if (this.state.valor < this.props.valorMaximo) {

      this.setState({
        valor: this.state.valor + 1
      });
    }
  },
  decrementa: function() {
    if (this.state.valor > this.props.valorMinimo) {
      this.setState({
        valor: this.state.valor - 1
      });
    }
  }
});

ReactDOM.render(
  <div>
  <Caja valor={3} valorMaximo={10} valorMinimo={0}/>
  <Caja valor={5} valorMaximo={10} valorMinimo={0}/>

</div>, document.getElementById('zona'));
```

El nuevo código incluye muchos cambios:

1 Añadir propiedades: Lo primero y más obvio es que hemos añadido las dos

propiedades “valorMaximo” y “valorMinimo”.

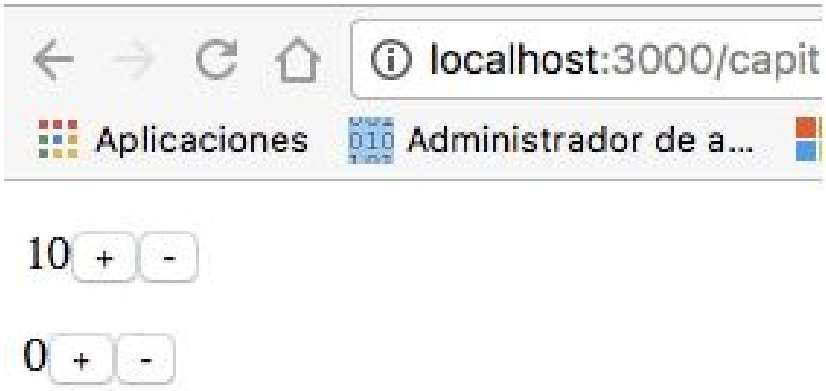
2 Incremento / decremento: Hemos modificado las funciones de incremento y decremento de tal forma que ahora dependen de los límites que se le asignen.

```
 incrementa: function() {

      if (this.state.valor < this.props.valorMaximo) {

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

3 JSX y propiedades: Hemos usado la sintaxis de JSX para asignar un valor máximo y mínimo.

```
ReactDOM.render(
   <div>
   <Caja valor={3} valorMaximo={10} valorMinimo={0}/>
   <Caja valor={5} valorMaximo={10} valorMinimo={0}/>

</div>, document.getElementById(‘zona'));
```

Ahora, cuando carguemos la página, el componente no podrá superar los límites definidos:



Hemos avanzado en el diseño del componente y añadido funcionalidad adicional .Es momento de abordar el tema de los estilos.

Componentes y estilos

Vamos a generar una hoja de estilo que nos muestre dos colores diferentes, dependiendo de si el valor que tenemos en el componente es válido o no.

```
<style type="text/css">
.rojo {
color:red;
}
.verde {
color:green;
}
</style>
```

Implementaremos los cambios necesarios en el componente para que esta nueva funcionalidad encaje. Como siempre, mostramos el nuevo código y explicamos sus modificaciones.

```
var Caja = React.createClass({

   propTypes: {

      valor: React.PropTypes.number.isRequired,
      valorMinimo: React.PropTypes.number,
      valorMaximo: React.PropTypes.number,
      estiloOK:React.PropTypes.string,
      estiloNoOK:React.PropTypes.string,
      valorOK:React.PropTypes.number,
   },

   getInitialState: function() {

      return {valor: this.props.valor};

   },
   estilo:function() {
```

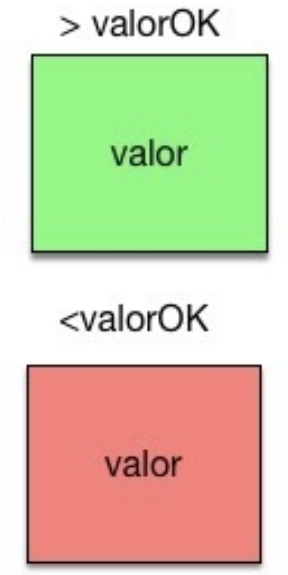
```
    if (this.state.valor >= this.props.valorOk) {

        return this.props.estiloOK;

    } else {

        return this.props.estiloNoOK;
    }

  },
  render: function() {
    return (
      <p>
        <span className={this.estilo()}>{this.state.valor}</span>
        <input type="button" value="+" onClick={this.incrementa}/>
        <input type="button" value="-" onClick={this.decrementa}/>
      </p>
    );
  },
  incrementa: function() {

    if (this.state.valor < this.props.valorMaximo) {

      this.setState({
        valor: this.state.valor + 1
      });
    }
  },
  decrementa: function() {
    if (this.state.valor > this.props.valorMinimo) {
      this.setState({
        valor: this.state.valor - 1
      });
    }
  }
});

ReactDOM.render(
  <div>
  <Caja valor={3}
   valorMaximo={10}
   valorMedio={5}
   valorMinimo={0}
   estiloOK="verde"
   estiloNoOK="rojo"
```

```
    />

   <Caja valor={7}
      valorMaximo={10}
      valorMinimo={0}
      valorOK={5}
      estiloOK="verde"
      estiloNoOK="rojo"
      />
</div>, document.getElementById('zona'));
```

La primero que hemos hecho es añadir nuevas propiedades al componente. Dos pertenecen al estilo que queremos aplicar para asignar **estiloOK o estiloNoOK**. La otra propiedad hace referencia **al valorOK, que es el número a partir del cuál damos por bueno (o no) un valor determinado.**

```
      valor: React.PropTypes.number.isRequired,
       valorMinimo: React.PropTypes.number,
       valorMaximo: React.PropTypes.number,
       estiloOK:React.PropTypes.string,
       estiloNoOK:React.PropTypes.string,
       valorOK:React.PropTypes.number,
```

Es decir, si el estado esta por encima de **valorOK** mostraremos el dato **en color verd**e , si no, lo mostraremos e**n valor rojo.**



Para conseguir que esta funcionalidad se implemente de forma correcta, hemos definido una función “estilo” encargada de asignar uno estilo u otro dependiendo del valorOK.

```
  estilo:function() {

    if (this.state.valor >= this.props.valorOK) {

       return this.props.estiloOK;

    } else {

       return this.props.estiloNoOK;
    }

   },
```

Estos estilos se pasarán a nivel de componente, basándose en las CSS definidas:

```
 <Caja valor={3}
    valorMaximo={10}
    valorOK={5}
    valorMinimo={0}
    estiloOK="verde"
    estiloNoOK="rojo"
    />
```

Por último, hemos modificado el componente para que su propiedad className invoque a la función de estilo cada vez que le mostremos:

```
    <span className={this.estilo()}>{this.state.valor}</span>
```

Hechas todas estas operaciones, vamos a construir varios componentes:

```
ReactDOM.render(
   <div>
   <Caja valor={3}
    valorMaximo={10}
    valorOK={5}
    valorMinimo={0}
    estiloOK="verde"
    estiloNoOK="rojo"
    />
```

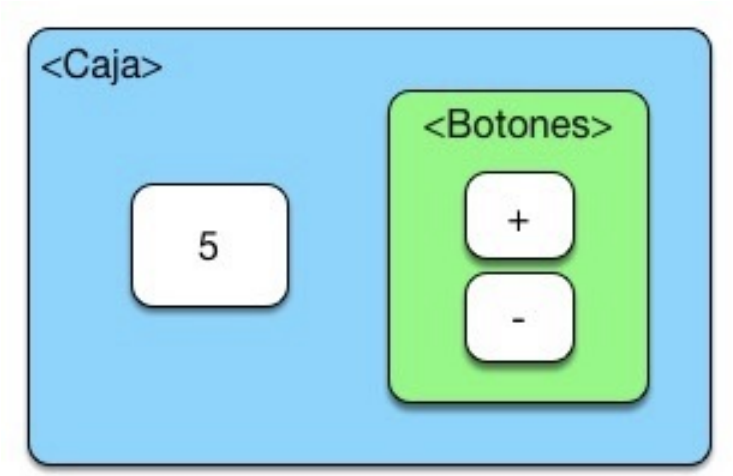
```
  <Caja valor={7}
    valorMaximo={10}
    valorMinimo={0}
    valorOK={5}
    estiloOK="verde"
    estiloNoOK="rojo"
    />
</div>, document.getElementById('zona'));
```

En este caso hemos decidido que, si el valor es menor de 5, se muestre en rojo y si no, en verde:

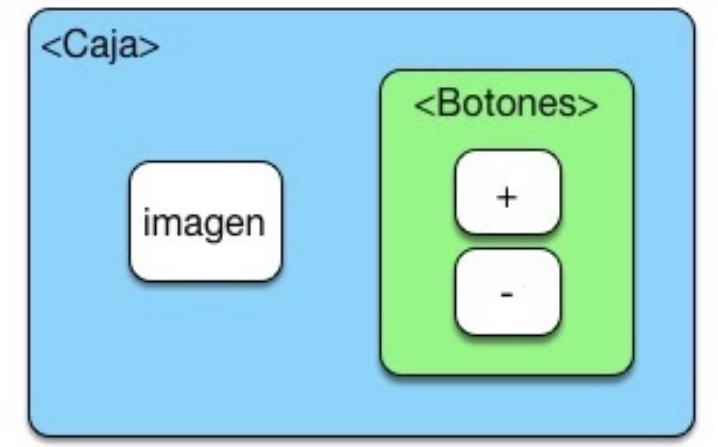
Hemos añadido funcionalidad al componente a través del uso de nuevas propiedades y estilos CSS. Ahora avanzaremos al abordar cómo trabajar con varios componentes.

Componentes y comunicación

Ya hemos construido nuestro primer componente con React **y manejamos los conceptos de propiedad y estado**. Sin embargo, cuando uno desarrolla una aplicación, necesita trabajar con varios componentes. Ahora debemos **dividir nuestro componente en dos:** por un lado los botones y por otro lado, la caja que asigna un valor.



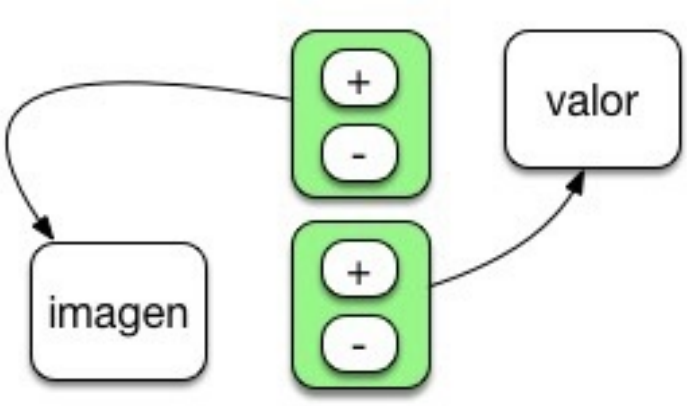
Es común que surja una duda inicial sobre por qué dividir el componente en dos partes si funcionaba perfectamente. La clave está muchas veces en aumentar **el nivel de reutilización**. Al diseñar la estructura de esta forma resulta relativamente sencillo construir otro componente que, por ejemplo, aumente de tamaño una imagen:



División de componentes

Cuando comencemos a dividir los componentes nos aparecerán las dudas: **¿Dónde ubicamos el estado?** ,

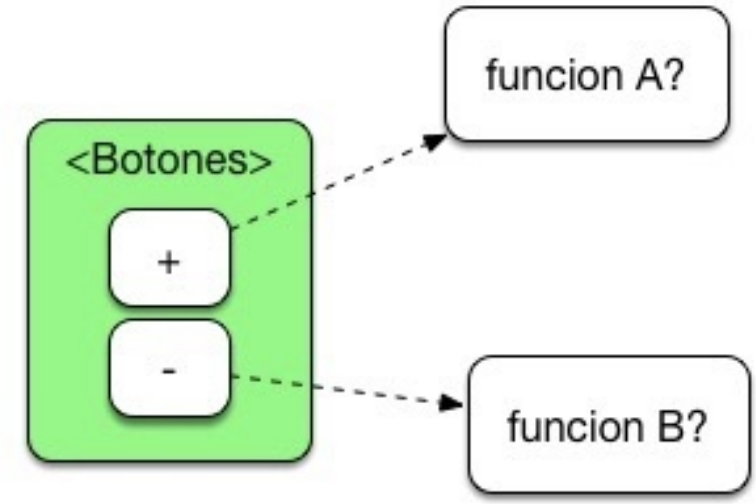
¿Dónde ubicamos las diferentes funciones?. Estas preguntas no tienen una respuesta sencilla. Siempre es útil pensar en cómo podríamos reutilizar los componentes en el futuro para tomar la decisión de asignar las responsabilidades. Por ejemplo, en nuestro caso podríamos usar los botones en otros componentes para realizar otras operaciones.



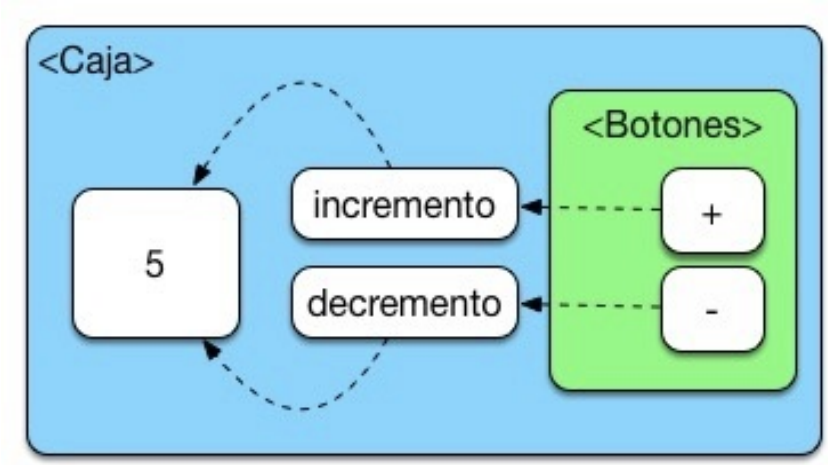
Por lo tanto, n**o tiene mucha lógica** que de entrada los botones tengan integradas las funcionalidades de **incrementar o hacer disminuir el valor, ya que no les podríamos reutilizar**. Lo que deberíamos hacer es dejar abierta la posibilidad de asignar l**a funcionalidad en un futuro**. **¿Cómo podemos hacer eso?**. Vamos a ver el nuevo código de el componente Botones:

```
var Botones = React.createClass({

   propTypes: {

      onMas: React.PropTypes.func.isRequired,
      onMenos: React.PropTypes.func.isRequired
   },

   render: function() {

      return (
         <span>
            <input type="button" value="+" onClick={this.props.onMas}/>
            <input type="button" value="-" onClick={this.props.onMenos}/>
         </span>
      );
   }
})
```

Podemos ver como los eventos onClick de cada botón se asocian a una propiedad **que es de tipo función**. Estas propiedades se definen como obligatorias en la zona de propTypes y tendrán un tipo especial **denominado "func" usado en el caso de que pasemos referencias a funciones que implementan otros componentes.**

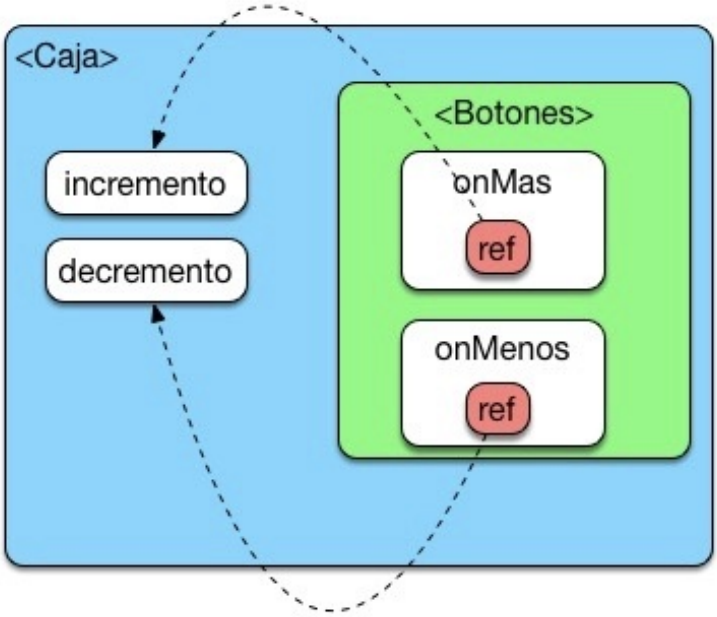


Nuestros botones todavía no tienen asignadas las funciones . **Así no podremos usarlos.** Para poder usarlos, necesitamos conectarlos a otro componente que es el que aporte la implementación de estas funciones.



Vamos a crear un nuevo componente Caja que contiene a los Botones e implementa la nueva funcionalidad.

```
var Caja = React.createClass({
```

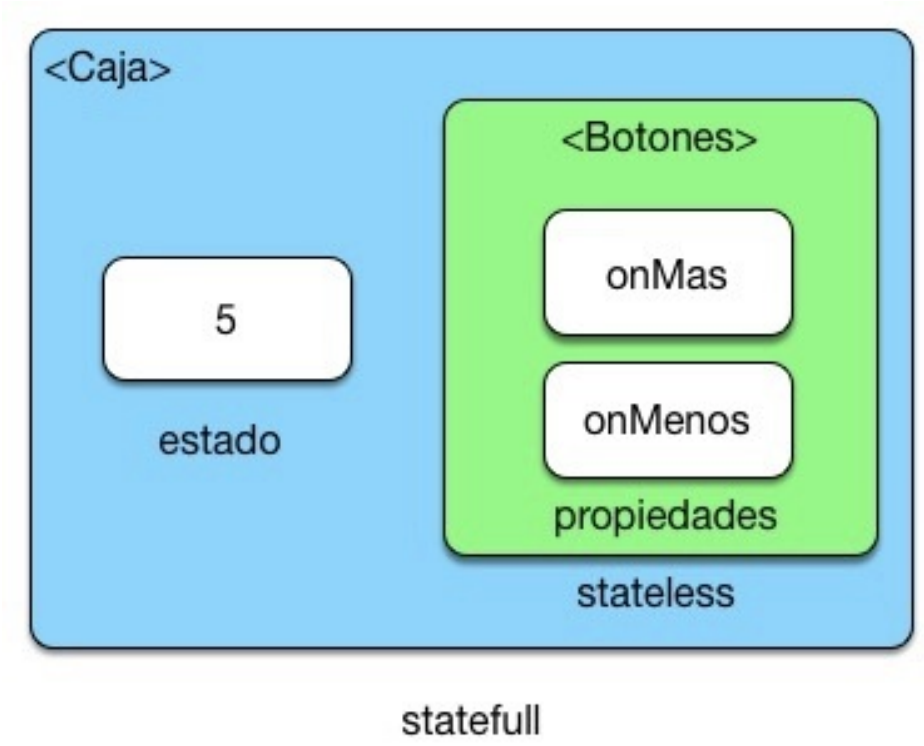



```
propTypes: {

   valor: React.PropTypes.number.isRequired,
   valorMinimo: React.PropTypes.number,
   valorMaximo: React.PropTypes.number,
   valorOK:React.PropTypes.number,
   estiloOK: React.PropTypes.string,
   estiloNoOK: React.PropTypes.string
},
getInitialState: function() {

   return {valor: this.props.valor};
},
estilo: function() {

   if (this.state.valor >= this.props.valorOK) {
      return this.props.estiloOK;
   } else {
      return this.props.estiloNoOK;
   }
},
render: function() {
   return (
      <p>
         <span className={this.estilo()}>{this.state.valor}</span>
         <Botones
          onMas={this.incrementa}
          onMenos={this.decrementa} />
```

```
        </p>
      );
    },
    incrementa: function() {
      if (this.state.valor < this.props.valorMaximo) {

        this.setState({
          valor: this.state.valor + 1
        });
      }
    },
    decrementa: function() {
      if (this.state.valor > this.props.valorMinimo) {
        this.setState({
          valor: this.state.valor - 1
        });
      }
    }
});
```

Ya tenemos el componente construido y podemos ver cómo el componente de Botones delega la responsabilidad de los botones + y - en las funciones incremento y decremento de la Caja.

Esta es una de las formas más sencillas, pero también más útiles de comunicación entre componentes: es como pasar un puntero. Ahora es el componente Caja el encargado de almacenar el estado mientras que el componente de Botones es un componente sin estado que delega en su componente padre. A estos componentes se les denomina en el argot d**e React Stateless y Statefull**.



Hasta ahora hemos trabajado con un estado muy sencillo ,es momento de avanzar y ver cómo manejar el estado con una listas de objetos.

React y Arrays

Vamos a crear un ejemplo que aborde el trabajo con **un array**. Para ello construiremos un **componente desde cero**. El componente **genera una lista de alumnos con sus notas.**

nombre	nota
angel	5
gema	7

Vamos a ver una primera versión del código de este nuevo componente:

```
var ListaAlumnos = React.createClass({
   render: function() {

     return (

        <table>
         <theader>

           <tr>
              <td>Nombre</td>
               <td>Nota</td>
           </tr>
         </theader>
         <tbody>

           {this.props.lista.map(function(alumno) {

                return <tr  key={alumno.nombre} ><td>{alumno.nombre}</td><td>{alumno.nota}</td></tr>
           })}
         </tbody>

        </table>

    );
 }
```

})

Hay muchas cosas que por ahora no entendemos del componente y que explicaremos más adelante. Por ahora, vamos a pasarle una lista de alumnos y ver lo que muestra en pantalla:

```
var alumnos =[{"nombre":"angel",nota:5},{"nombre":"gema",nota:7}];


ReactDOM.render(
   <div>
      <ListaAlumnos  lista={alumnos}/>
   </div>
   , document.getElementById('zona'));
```

El navegador nos mostrará la tabla:

Tabla y Arrays

Procedamos a revisar el código: lo primero que podemos observar es que hemos pasado el array de alumnos al componente como propiedad

```
<ListaAlumnos  lista={alumnos}/>
```

Hecho esto, el siguiente paso será **recorrer la lista usando la función map de JavaScript y generar cada una de las filas con la etiqueta <tr>**:

```
return (

   <table>
    <theader>
```

```
        <tr>
            <td>Nombre</td>
             <td>Nota</td>
        </tr>
      </theader>
      <tbody>

        {this.props.lista.map(function(alumno) {

             return <tr  key={alumno.nombre} ><td>{alumno.nombre}</td><td>{alumno.nota}</td></tr>
        })}
      </tbody>

    </table>

  );
```

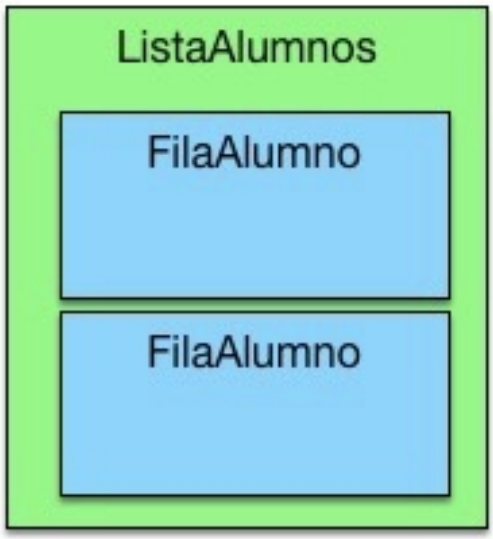
Aquí hay que hacer una aclaración: cuando generamos cada una de las filas, asignamos los datos de los alumnos y una propiedad un poco peculiar **que se denomina key.**

<tr **key={alumno.nombre}** ><td>{alumno.nombre}</td><td>{alumno.nota}</td></tr>

Esta propiedad es usada por React para saber el número de fila en la que se está operando y poder procesar los cambios adecuadamente.

JSX y modularización

El código que tenemos **construido es correcto y funcional.** Ahora bien, en React es común tener **una fuerte modularización en los componentes**. Así pues, podríamos redefinir la estructura que tenemos actualmente al añadir **un nuevo componente FilaAlumno.**



Vamos a ver el código del nuevo componente:

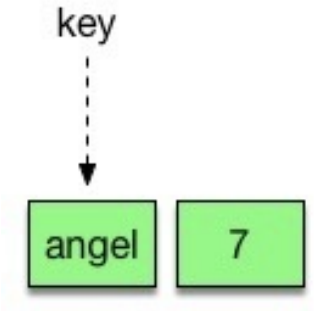
```
var FilaAlumno=React.createClass({
  render: function() {

     return (
          <tr>
           <td>{this.props.alumno.nombre}</td>
           <td>{this.props.alumno.nota}</td>
           </tr>
     );
  }
});
```

El código de la ListaAlumnos se simplifica **y dividimos las responsabilidades.**

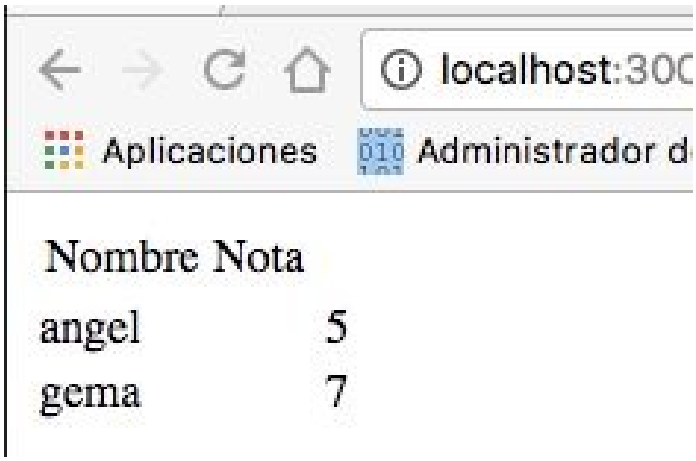
```
var ListaAlumnos=React.createClass({
  render: function() {

     return (

        <table>
          <theader>

            <tr>
               <td>Nombre</td>
                <td>Nota</td>
            </tr>
```

```
        </theader>
        <tbody>

          {this.props.lista.map(function(alumno) {

              return <FilaAlumno key={alumno.nombre} alumno={alumno}/>
          })}
        </tbody>

      </table>

    );
  }
})
```

A la hora de generar una lista de FilaAlumno, React obliga a asignar una clave o key para poder controlar el estado de cada fila. En este caso hemos elegido el nombre como clave.

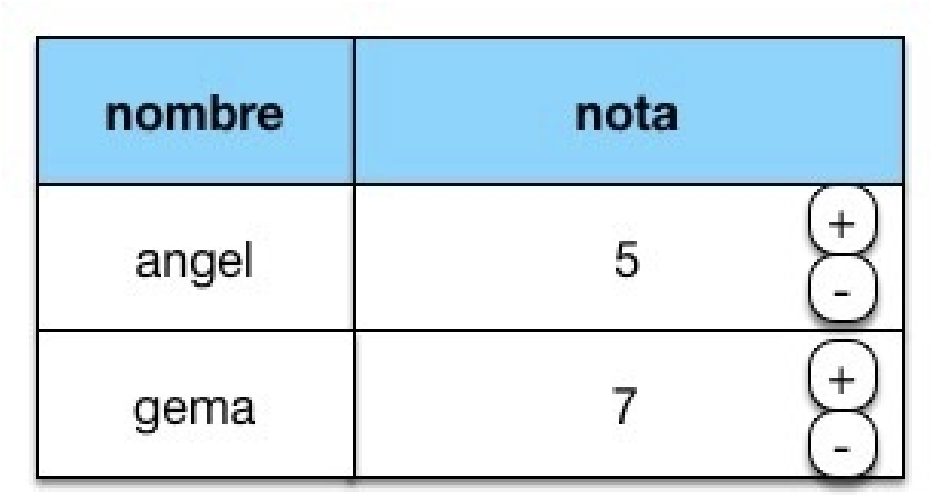


Resueltas estas modificaciones, el resultado por pantalla será el mismo :



Integración de componentes

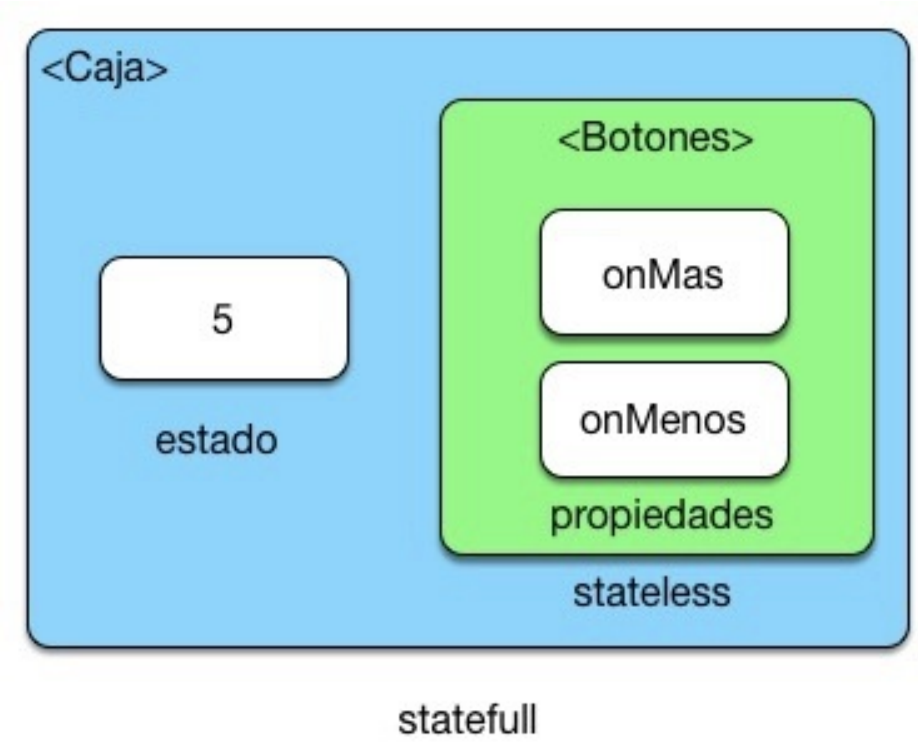
Es evidente que nos puede venir muy bien integrar la funcionalidad de la <Caja> y <Botones> en nuestro <ListaAlumnos/>



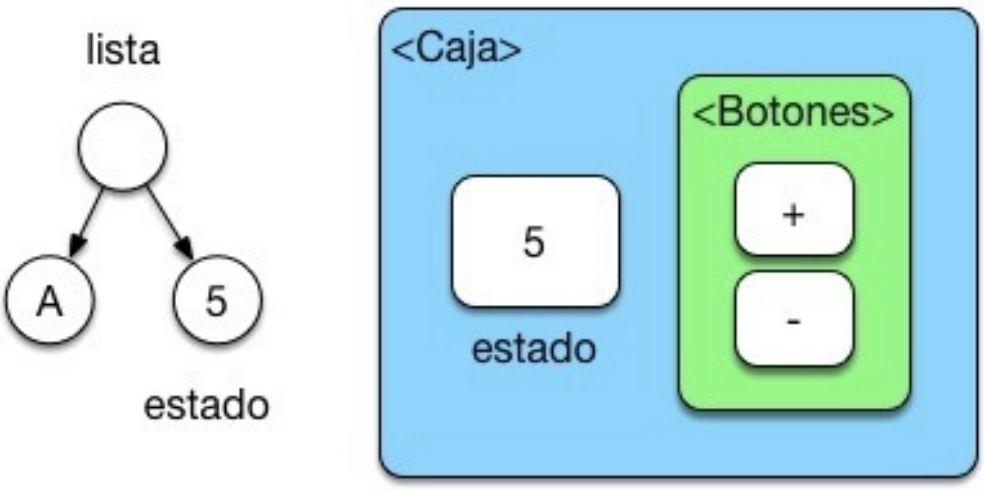
Pero antes de abordar este paso tenemos que revisar el componente Caja creado anteriormente.

JSX y Containers

Tenemos creado el componente Caja: recordemos que los <Botones> son stateless y la <Caja> es statefull y mantiene estado.



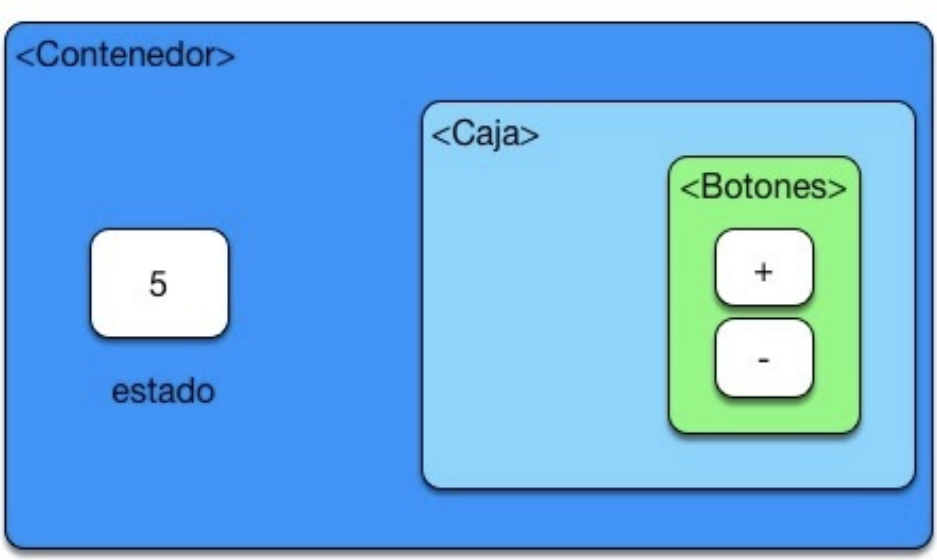
La pregunta clave en este caso es : **¿Hemos hecho lo correcto al almacenar el estado en la Caja?**. **En principio todo parece indicar que sí** . Ahora bien: **cuando integremos varios componentes,** comenzarán a aparecer preguntas incomodas. ¿Dónde almacenamos el estado: en la lista de alumnos o en la propia <Caja>?



Esta es una pregunta totalmente justificada. Si nuestra aplicación contiene un array de alumnos donde almacenar el estado, **¿Es necesario que nuestro componente lo almacene a la vez?**

React y Contenedores

Con frecuencia, React hace uso del concepto de contenedor para gestionar el estado compartido entre varios componentes.



Ahora debemos analizar nuestro componente y ver si podemos enfocar de otra forma **y no almacenar estado en él** .Como ya vimos anteriormente, el componente <Botones> es stateless y no almacena ningún estado. Es el componente <Caja> el que mayor responsabilidad almacena. Concretamente gestiona **los estilos ,los limites de valores que se pueden asignar y el estado.**

¿Cuáles de estas responsabilidades pertenecen a la Caja y cuales no?. Vamos a analizar cada situación:

Estilos: La caja es la que se encarga de dibujarse **a sí misma en el método render().** Por lo tanto, es lógico pensar que la gestión de los estilos se encuentre dentro de ella.

Limites Valores: Esto ya genera más dudas: las propiedades de valorMáximo y valorMínimo las recibe la caja y opera con ellas.

```
incrementa: function() {
      if (this.state.valor < this.props.valorMaximo) {
         this.setState({
            valor: this.state.valor + 1
         });
      }
   },
   decrementa: function() {
      if (this.state.valor > this.props.valorMinimo) {
         this.setState({
            valor: this.state.valor - 1
         });
      }
   }
```

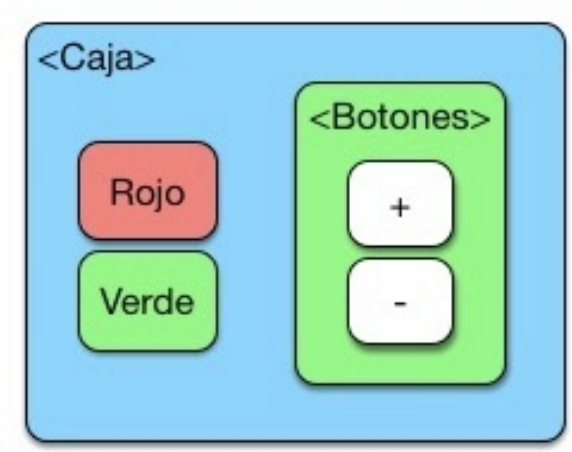
¿Es una responsabilidad de la Caja o de un elemento externo?. La realidad es que los valores tope se usan para limitar hasta que valor la Caja puede llegar. Sin embargo, es aquí donde debemos diferenciar dos responsabilidades.

1. **Tope de valores**
2. **Asignación de estado**

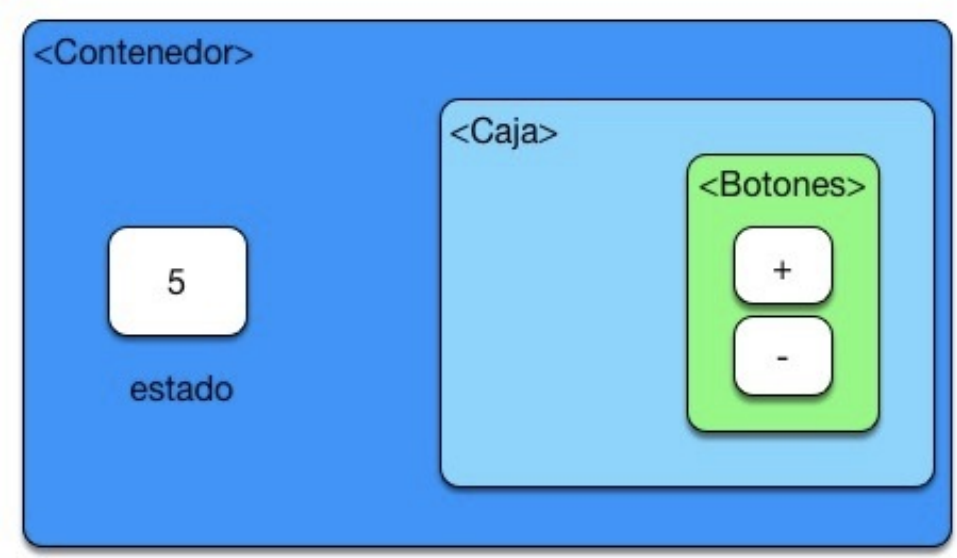
La primera parte la definen los if y la segunda parte la define el contenido de las sentencias if

```
if (this.state.valor > this.props.valorMinimo) {    // limites tope
        this.setState({
           valor: this.state.valor - 1 //asignación estado
        });
    }
```

Tras reflexionar nos daremos cuenta de que la funcionalidad de los topes pertenece a la <Caja>, ya que si la extraemos **otros componentes, se verían obligados a implementarla por su cuenta**. Es mejor tenerla ubicada dentro del propio componente y reutilizar código.



La pregunta que nos queda es **¿y el estado?. Ahora sabemos que en el futuro vamos a tener un array de objetos que lo almacenará.** Así pues debemos extraer el estado de la Caja. No se trata de una operación sencilla de entender , por lo tanto lo vamos a hacer poco a poco**.** En este capítulo lo vamos a sacar a un componente <Contenedor> que lo almacene como un primer paso.



Vamos a ver el código fuente para este enfoque :

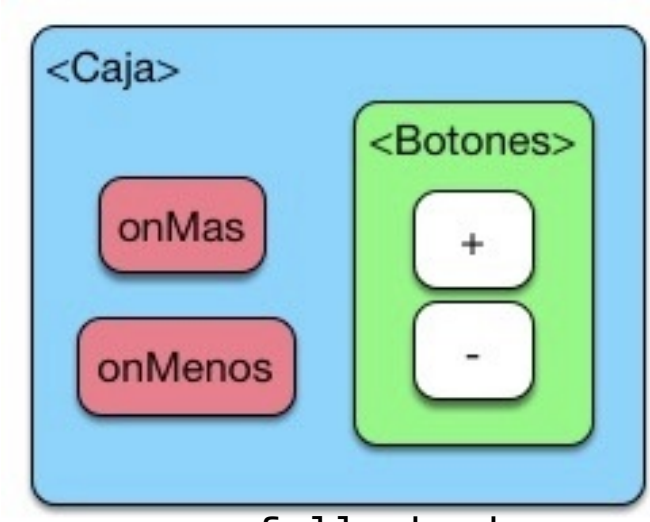
```
var Botones = React.createClass({
   propTypes: {

      onMas: React.PropTypes.func.isRequired,
      onMenos: React.PropTypes.func.isRequired
   },
   render: function() {
      return (
         <span>
            <input type="button" value="+" onClick={this.props.onMas}/>
            <input type="button" value="-" onClick={this.props.onMenos}/>
         </span>
      );
   }
})

var Caja = React.createClass({

   propTypes: {

      valor: React.PropTypes.number.isRequired,
      valorMinimo: React.PropTypes.number,
      valorMaximo: React.PropTypes.number,
      valorOK:React.PropTypes.number,
      estiloOK: React.PropTypes.string,
      estiloNoOK: React.PropTypes.string,
      onMas:React.PropTypes.func.isRequired,
      onMenos:React.PropTypes.func.isRequired
   },

   onMas: function() {

      if (this.props.valor < this.props.valorMaximo) {

        this.props.onMas();

      }
   },
   onMenos: function() {
      if (this.props.valor > this.props.valorMinimo) {
```

```
      this.props.onMenos();
    }
  },

  estilo: function() {

    if (this.props.valor >= this.props.valorOK) {

      return this.props.estiloOK;

    } else {

      return this.props.estiloNoOK;
    }

  },
  render: function() {
    return (
      <p>
        <span className={this.estilo()}>{this.props.valor}</span>
        <Botones
          onMas={this.onMas}
          onMenos={this.onMenos} />
      </p>
    );
  },

});


var nota=5;

var Contenedor = React.createClass({

 valor: React.PropTypes.number.isRequired,

 getInitialState: function() {
   return {valor: this.props.valor};
 },

 incrementa: function() {
      this.setState({
        valor: this.state.valor + 1
      });
 },
```

```
  decrementa: function() {

        this.setState({
          valor: this.state.valor - 1
        });
  },
   render: function() {
      return (
         <div>
          <Caja valor={this.state.valor}
             valorOK={5}
             valorMaximo={10}
             valorMinimo={0}
             estiloOK="verde"
             onMas={this.incrementa}
             onMenos={this.decrementa}
             estiloNoOK="rojo"/>
         </div>
      );
   }
})
ReactDOM.render(
   <div>
     <Contenedor valor={5} />
   </div>
   , document.getElementById('zona'));
```

Contenedor y Caja

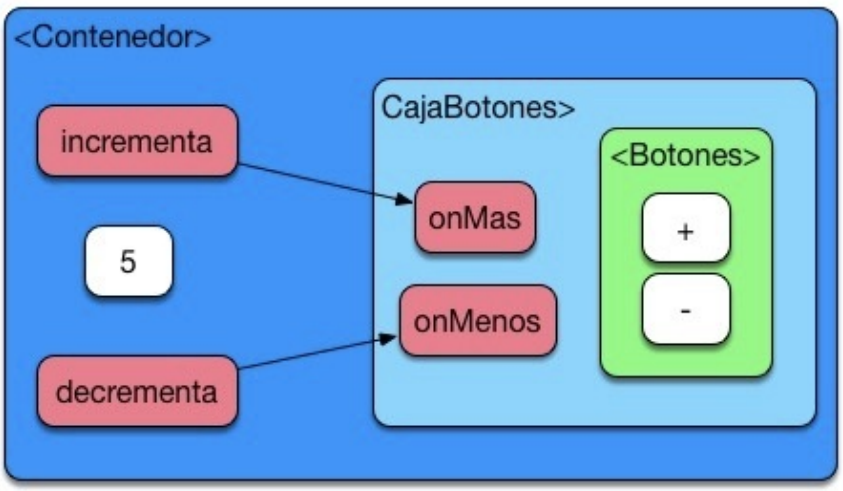
Tenemos 3 componentes y mucho código que revisar .Vamos a destacar varios bloques de código. En primer lugar, las funciones onMas y onMenos del componente Caja:



Estos métodos se van a encargar de controlar los topes a los que el valor puede llegar .

```
onMas: function() {

    if (this.props.valor < this.props.valorMaximo) {

      this.props.onMas();

    }
  },
  onMenos: function() {
    if (this.props.valor > this.props.valorMinimo) {

      this.props.onMenos();
    }
  },
```

Ahora bien, ya no se encarga de cambiar su propio **estado, sino que delega el contenedor (componente padre) para hacerlo, ya que éste almacena el estado**.Para implementar la funcionalidad de forma correcta, necesitamos que nuestra <Caja> reciba como parámetros las funciones que debe invocar del contenedor.



Por eso, el componente Caja ahora define dos propiedade**s nuevas obligatorias que hacen referencia a las funciones.**

```
propTypes {

……..
 onMas:React.PropTypes.func.isRequired,
```

```
 onMenos:React.PropTypes.func.isRequired
}
……..
```

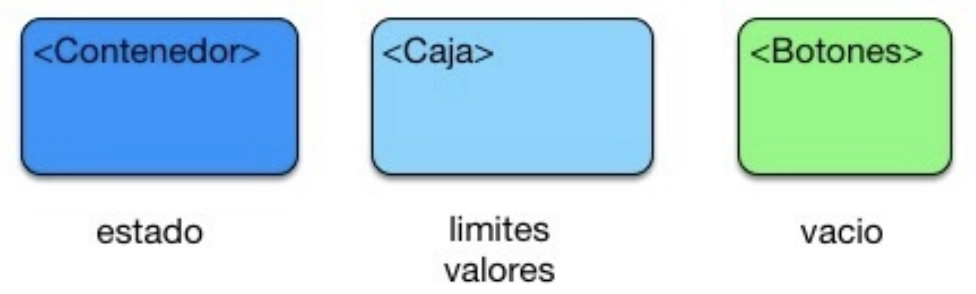
Estas propiedades se pasan a través del Componente:

```
 <Caja valor={this.state.valor}
              valorOK={5}
              valorMaximo={10}
              valorMinimo={0}
              estiloOK="verde"
              onMas={this.incrementa}
              onMenos={this.decrementa}
              estiloNoOK="rojo"/>
```

Nos queda revisar **el Contenedor :** es este el que almacena el estado y la responsabilidad **de cambiarlo:**

```
incrementa: function() {
       this.setState({
          valor: this.state.valor + 1
       });

 },
 decrementa: function() {

       this.setState({
          valor: this.state.valor - 1
       });
 },
```

Como vemos, las responsabilidades han quedado muy repartidas entre todos los componentes.

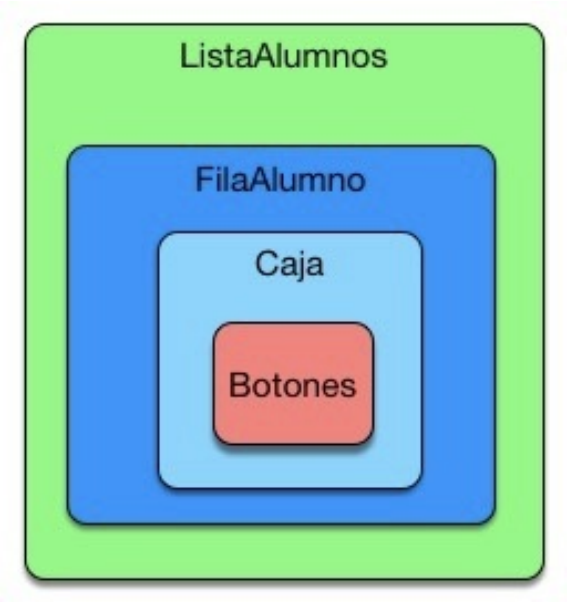


El resultado cuanto a la funcionalidad **es idéntico.**

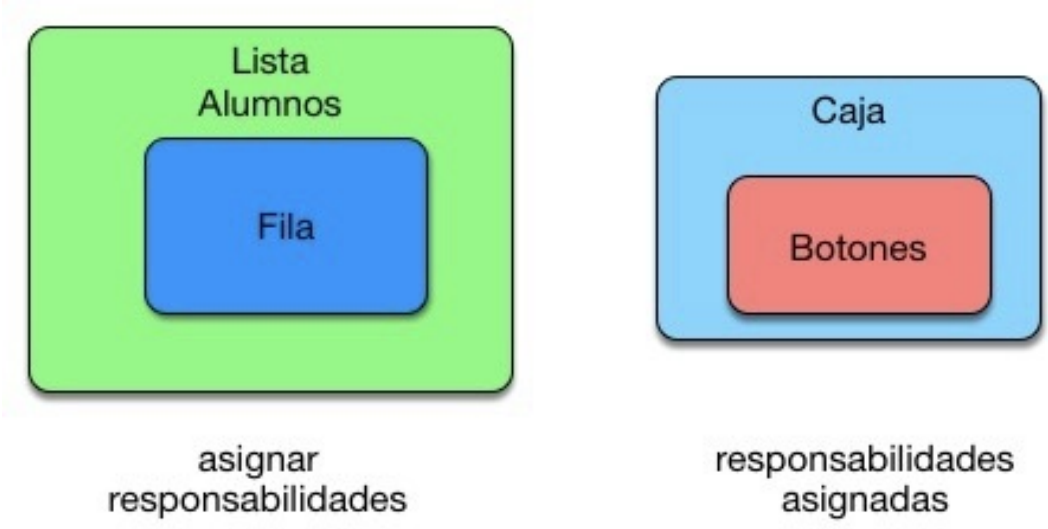
Acabamos de ver cómo dividir responsabilidades. Es momento de ver como podemos integrar <ListaAlumnos> con el componente de <Caja> sin usar el contenedor, ya que será <ListaAlumno> el que almacene el estado.

JSX e integración

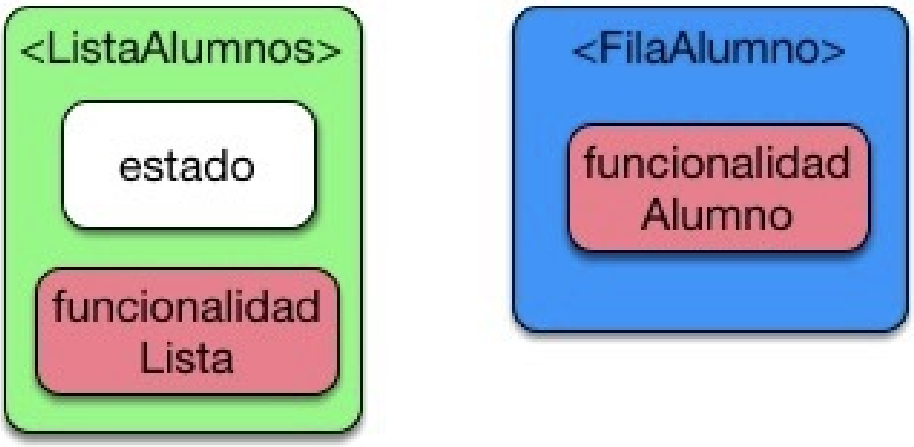
Ahora mismo disponemos de cuatro componentes .Las responsabilidades de <Caja> y <Botones> están definidas. Es momento de abordar su integración con <ListaAlumnos> y <FilaAlumno>.



En estos momentos lo que no parece tan claro es cómo dividir las responsabilidades del componente de FilaAlumnos y TablaAlumnos.



Deberemos definir en qué componente se va a quedar el estado y qué responsabilidades va a tener tanto <ListaAlumnos> como <FilaAlumno>. FilaAlumno se va a encargar de renderizar el objeto Alumno y es probablemente el mejor lugar donde ubicar la funcionalidad **que afecte a cada Alumno de forma independiente**. Por otro lado, la<ListaAlumnos> contendrá la funcionalidad que gestione la lista de Alumnos y será el componente raíz que almacene el estado.



Vamos a ver el código del nuevo componente FilaAlumno y qué funcionalidad concreta almacena:

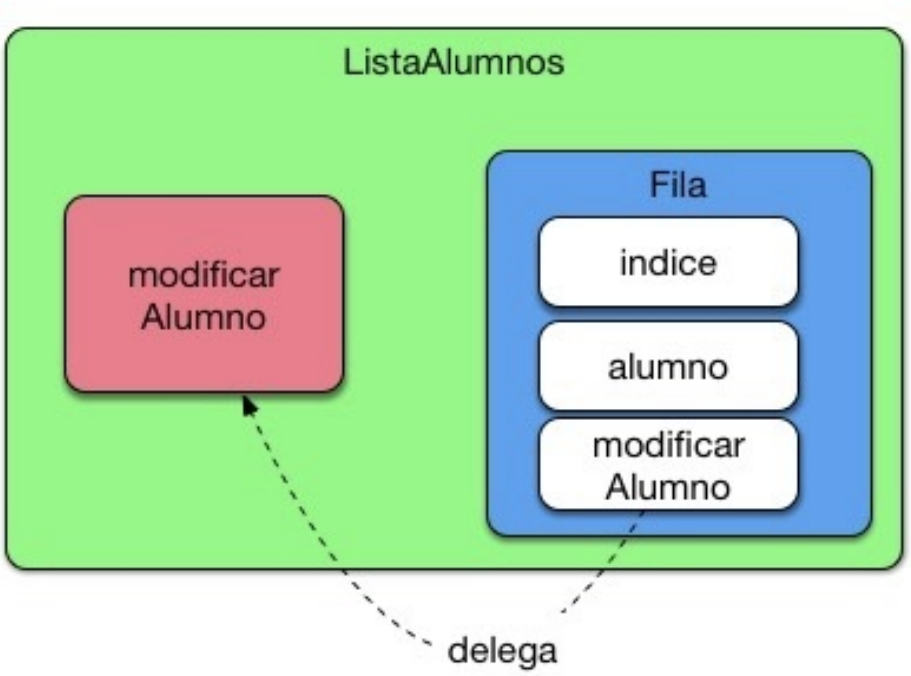
```
var FilaAlumno=React.createClass({

 propTypes: {

    alumno: React.PropTypes.object.isRequired,
    indice: React.PropTypes.number.isRequired,
    modificarAlumno:React.PropTypes.func.isRequired,
 },

 incrementarNota:function() {

    var nuevoAlumno={};
    nuevoAlumno.nombre=this.props.alumno.nombre;
    nuevoAlumno.nota=this.props.alumno.nota+1;
    this.props.modificarAlumno(nuevoAlumno,this.props.indice);

 },
 decrementarNota:function() {

   var nuevoAlumno={};
   nuevoAlumno.nombre=this.props.alumno.nombre;
   nuevoAlumno.nota=this.props.alumno.nota-1;
   this.props.modificarAlumno(nuevoAlumno,this.props.indice);

 },
 render: function() {
```

```
    return (
        <tr>
         <td>{this.props.alumno.nombre}</td>
         <td> <Caja valor={this.props.alumno.nota}
                onMas={this.incrementarNota}
                onMenos={this.decrementarNota}
                valorOK={5}
                valorMaximo={10}
                valorMinimo={0}
                estiloOK="verde"
                estiloNoOK="rojo"/></td>
        </tr>
    );
  }
});
```

Hemos hecho algunos cambios al componente y como siempre los analizaremos. En primer lugar vamos a echar un vistazo **a los parámetros recibidos como propiedades.**

```
    alumno: React.PropTypes.object.isRequired,
    indice: React.PropTypes.number.isRequired,
    modificarAlumno:React.PropTypes.func.isRequired,
```

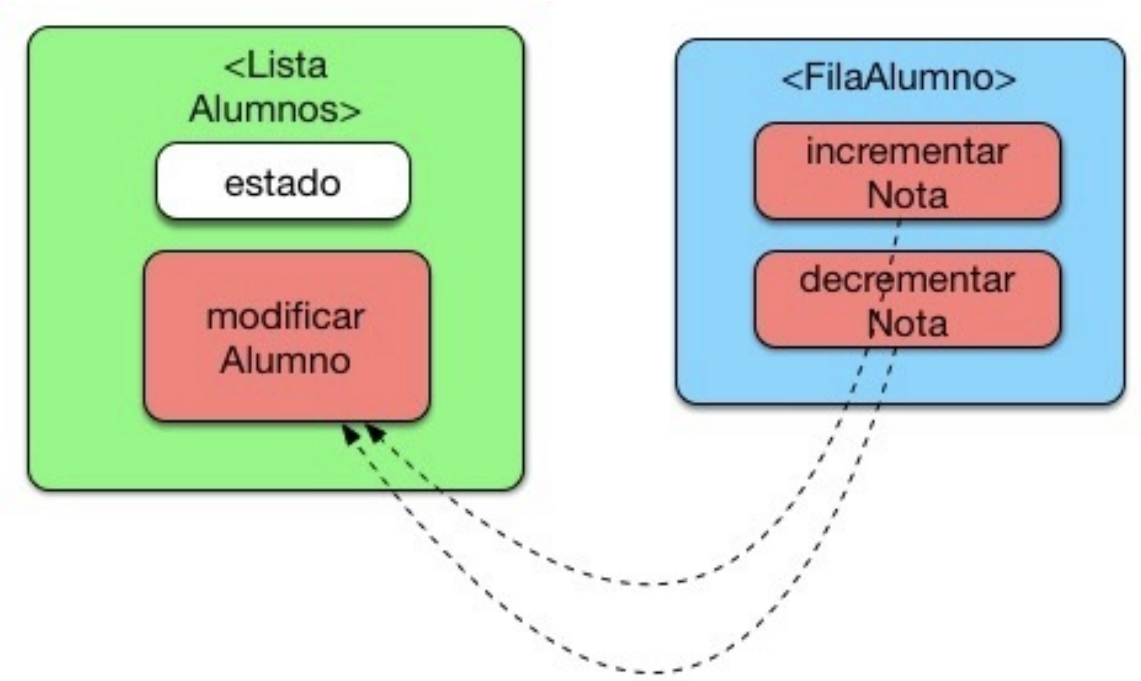
La primera es la más sencilla de entender: se trata del alumno que tiene que renderizar . La segunda propiedad es un índice que nos permite saber **en qué número de fila estamos** . Por último recibe como parámetro una función para delegar en el componente padre (ListaAlumnos) que será el encargado de d**e almacenar y modificar el estado de cada Alumno.**



Vamos a ver detalladamente las funciones implementadas para trabajar con un objeto alumno:

```
incrementarNota:function() {

    var nuevoAlumno={};
    nuevoAlumno.nombre=this.props.alumno.nombre;
    nuevoAlumno.nota=this.props.alumno.nota+1;
    this.props.modificarAlumno(nuevoAlumno,this.props.indice);

},
decrementarNota:function() {

  var nuevoAlumno={};
  nuevoAlumno.nombre=this.props.alumno.nombre;
  nuevoAlumno.nota=this.props.alumno.nota-1;
  this.props.modificarAlumno(nuevoAlumno,this.props.indice);

},
```

En este caso, la FilaAlumno se encarga de leer los valores del objeto alumno **y crear un nuevo alumno que contenga el valor de la nota actualizado**. Este nuevo alumno se pasará al componente padre TablaAlumno para que él se encargue de actualizar el estado a través del método modificarAlumno.



Componente ListaAlumnos

El componente ListaAlumnos es el encargado de de modificar el estado de los diferentes Alumnos:

```
var ListaAlumnos=React.createClass({

 getInitialState: function() {

    return {lista: this.props.lista};
 },

 modificarAlumno: function(alumno,indice) {

  var listaNueva= this.state.lista.slice();
  listaNueva.splice(indice,1,alumno);
  this.setState({lista:listaNueva});
 },

 render: function() {

    return (

       <table>
        <tbody>
```

```
        {this.state.lista.map(function(alumno, indice ) {
            return <FilaAlumno
              key={alumno.nombre}
              alumno={alumno}
              modificarAlumno={this.modificarAlumno}
              indice={indice} />
          }.bind(this))}
        </tbody>

      </table>

    );
  }
});
```

A pesar de no ser mucho código, entraña cierta dificultad:

1. En primer lugar está el método bind (this) ,que se encarga de que la variable this almacene un puntero al objeto ListaAlumnos y podamos en FilaAlumno invocarlo.

2. El método modificar alumno se encarga de crear una nueva lista con el alumno modificado ,para ello, primero clona la lista y luego cambia el alumno en la posición definida por el indice Por último, actualizamos el estado del componente asignándole una nueva lista. Lo único que queda es renderizar el componente y ya tendremos integrados nuestros botones dentro de la Tabla.

```
ReactDOM.render(
  <div>
    <ListaAlumnos  lista={alumnos} />
  </div>
  , document.getElementById('zona'));
```

El resultado es completamente operativo :

Acabamos de integrar los cuatro componentes que habíamos construido con React delegando de forma

adecuada.

Optimización y ShouldComponentUpdate

Si queremos optimizar el código y que React sólo renderice aquellos componentes que cambian de estado, podemos usar el método ShouldComponentUpdate que permite definir cuándo el componente ha de volver a ser renderizado. En nuestro **caso, sólo es obligatorio al cambiar la nota del Alumno. Añadiremos el método shouldComponentUpdate en FilaAlumno**

```
shouldComponentUpdate:function(nextProps, nextState){

    if (this.props.alumno.nota!==nextProps.alumno.nota) {
       return true;
    }else {
       return false;
    }
 },
```

De esta manera, únicamente cuando un componente **cambie el valor de la nota, la fila este se volverá a dibujar**.El siguiente paso será obtener la lista de alumnos del servidor usando AJAX.

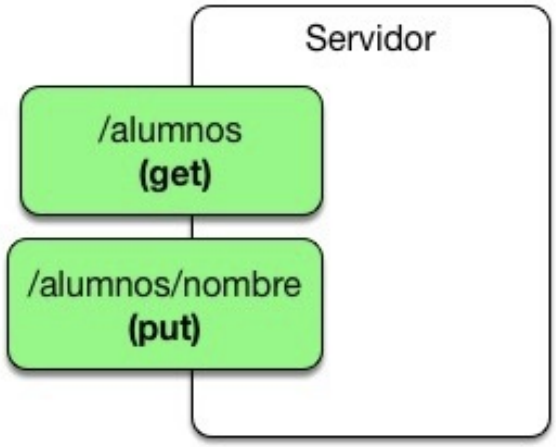
React y Ajax

Hasta ahora hemos construido una jerarquía de componentes que opera sobre u**na lista de objetos JavaScript estáticos cargados en la propia página.** Es momento de configurar nuestros componentes para que se apoyen en el servidor y sean capaces de obtener los datos vía Ajax. Para ello, vamos a modificar el código del servidor a fin de que sea capaz de gestionar la lista de alumnos.

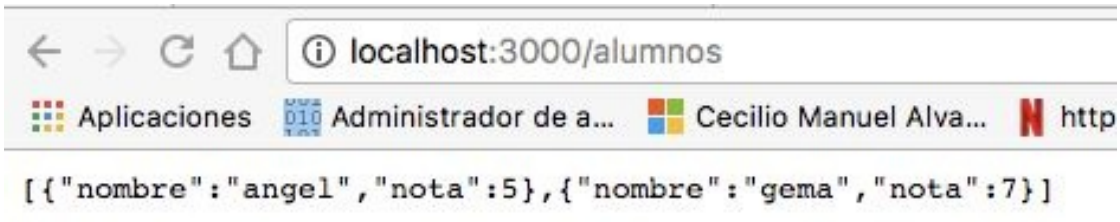
```
var express = require('express');
var bodyParser = require('body-parser');
var app = express();
app.use(bodyParser.json());

var alumnos = [{
   "nombre": "angel",
   nota: 5
}, {
   "nombre": "gema",
   nota: 7
}];

app.get("/alumnos", function(request, response) {

   return response.send(alumnos);

});
app.put("/alumnos/:nombre", function(request, response) {


   var alumno = alumnos.find(function(alumno) {
      return alumno.nombre == request.params.nombre;
   });
   var indice = alumnos.indexOf(alumno);
   alumnos.splice(indice, 1, request.body);

   return response.send({
      mensaje: "ok"
   });

});

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

app.listen(3000, function() {
```

```
    console.log('servidor arrancado 3000');
});
```

Hemos dado de alta nuevos métodos que nos permiten tanto obtener la lista de alumnos así como actualizarla (métodos get/put).



Cuando invocamos la url de /alumnos estamos realizando una petición GET y el servidor nos devolverá la lista en formato JSON.



Componentes y Ajax

Ahora es necesario ver qué modificaciones debemos hacer en nuestros componentes para que se puedan apoyar en los datos disponibles en el servidor. En nuestro caso es suficiente con modificar el componente ListaAlumnos:

```
var ListaAlumnos = React.createClass({

  propTypes: {

     recurso: React.PropTypes.String.isRequired,

  },
```

```
  getInitialState: function() {

    return {lista: []};
  },

  componentDidMount() {

    var componente = this;

    fetch(this.props.recurso, {method: 'get'}).then(function(response) {
      return response.json();
    }).then(function(datos) {
      componente.setState({lista: datos});

    }).catch(function(err) {

      console.log("errores" + err);

    });
  },
  modificarAlumno: function(alumno, indice) {

    var listaNueva = this.state.lista.slice();
    listaNueva.splice(indice, 1, alumno);
    this.setState({lista: listaNueva});
  },

  render: function() {

    return (

      <table>
        <tbody>

          {this.state.lista.map(function(alumno, indice) {
            return <FilaAlumno key={alumno.nombre} alumno={alumno} modificarAlumno=
{this.modificarAlumno} indice={indice}/>
          }.bind(this))}
        </tbody>

      </table>

    );
  }
});
```

```
ReactDOM.render(
   <div>
   <ListaAlumnos recurso="/alumnos"/>
</div>, document.getElementById('zona'));
```

En este caso **encontramos pocas modificaciones** .Hemos añadido una nueva propiedad denominada "recurso" a ListaAlumnos. Recordemos que vamos a acceder a un API REST.

```
propTypes: {

     recurso: React.PropTypes.string.isRequired,

  },
```

Hecho este primer cambio, el siguiente paso es añadir **el método componentDidMount, que se ejecutará cuando el componente esté construido** y usará el API de promesas de JavaScript ES6 para hacer una petición al servidor ,obtener la lista de Alumnos y actualizar el estado del componente.

```
 componentDidMount() {

       var componente = this;

       fetch(this.props.recurso, {method: 'get'}).then(function(response) {
          return response.json();
       }).then(function(datos) {
          componente.setState({lista: datos});

       }).catch(function(err) {

          console.log("el errores" + err);

       });
    },
```

Realizadas las modificaciones, el componente funcionará apoyándose en los datos almacenados en el lado del servidor y realizando una petición REST.

Es aquí donde pueden surgir los problemas, ya que podemos incrementar o decrementar la nota de cada Alumno ,pero **estos valores no se actualizarán en el servidor.** Lo más lógico es crear una función dentro <ListaAlumnos/> que realice una petición PUT y actualice las notas de los alumnos en el lado servidor.

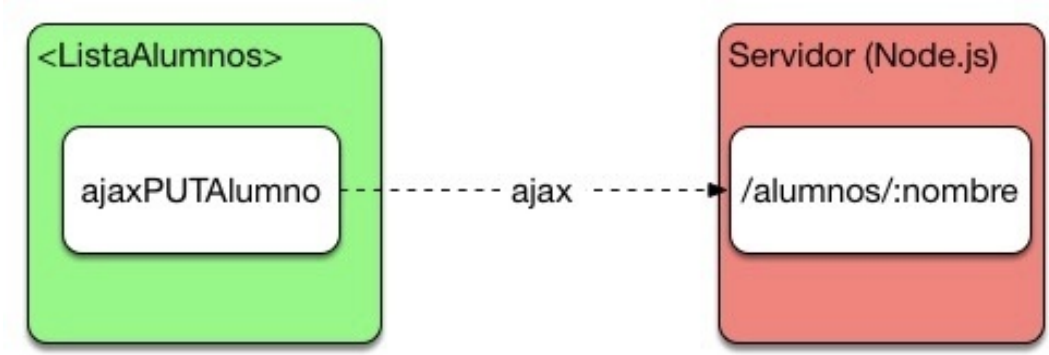
Existen muchas formas de enfocar esta situación: podemos hacer un PUT cada vez que la nota se incremente. Podemos mantener algún tipo de estado en cada una de las <FilaAlumno/> para saber cuál ha cambiado y realizar un PUT desde ellas. Hay muchas opciones ,**en este caso vamos a optar por solución relativamente sencilla:** vamos a realizar una petición por cada fila cuando se hagan cambios :**no es lo más óptimo pero es sencillo de implementar**. Para ello vamos a diseñar dos métodos nuevos en el componente ListaAlumnos.

El primer método se encarga de realizar una petición de PUT por cada alumno que tenemos:

```
ajaxPUTAlumno: function(alumno) {
```

```
    var peticion = fetch(this.props.recurso + "/"+alumno.nombre, {
        method: 'put',
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(alumno)
    });
    return peticion;

},
```

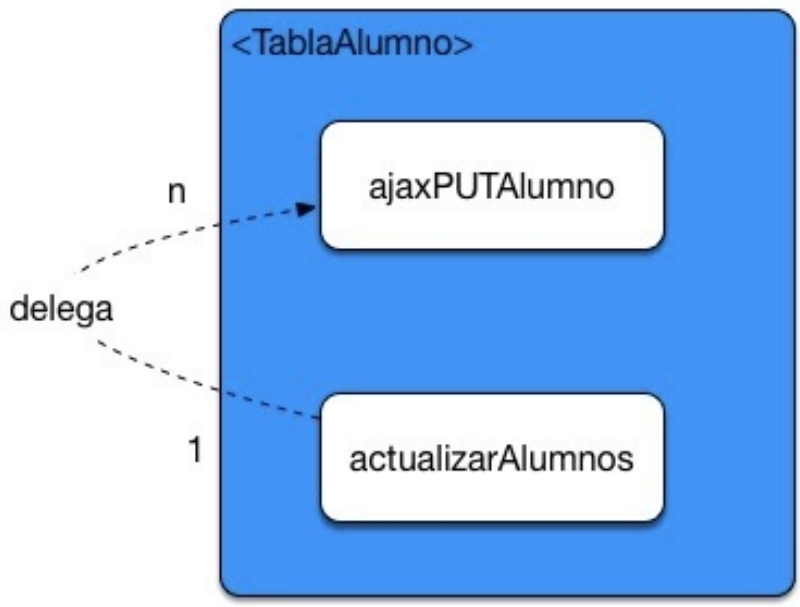
Como en otros casos ,nos hemos apoyado en fetch API para diseñar las peticiones asíncronas. Cada vez que invoquemos al método ajaxPUTAlumno ,invocaremos al método /alumnos/:nombre para un alumno determinado.



Recordemos que el código del lado servidor es:

```
app.put("/alumnos/:nombre", function(request, response) {


  var alumno = alumnos.find(function(alumno) {
    return alumno.nombre == request.params.nombre;
  });
  var indice = alumnos.indexOf(alumno);
  alumnos.splice(indice, 1, request.body);

  return response.send({
    mensaje: "ok"
  });

});
```

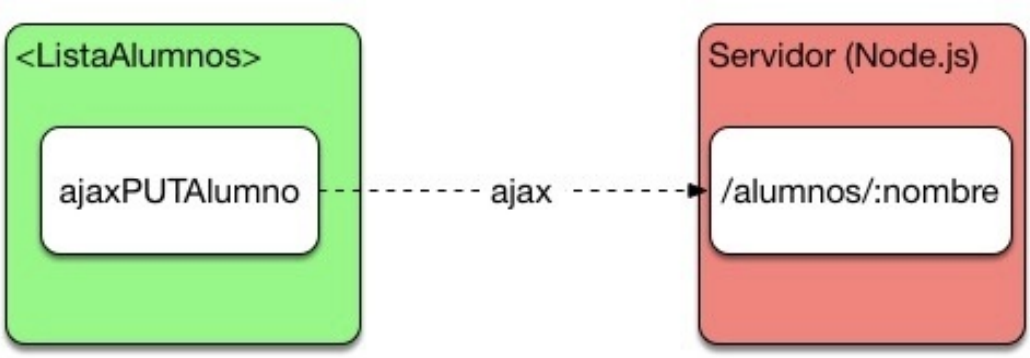
El código del servidor se parece bastante al del cliente :**estamos trabajando con JavaScript en ambos lados** . En este caso simplemente actualizamos el registro en el servidor y devolvemos un mensaje de “ok”. Queda por ver el otro método: el que actualiza la lista entera en el lado cliente y que delega en el método ajaxPUTAlumno para realizar las peticiones al servidor.



Veamos el contenido de una primera versión del método:

```
  actualizarAlumnos: function() {

      var componente = this;
      var listaPromesas = this.state.lista.map(function(alumno) {
         return componente.ajaxPUTAlumno(alumno);

      })

      Promise.all(listaPromesas);
   },
```

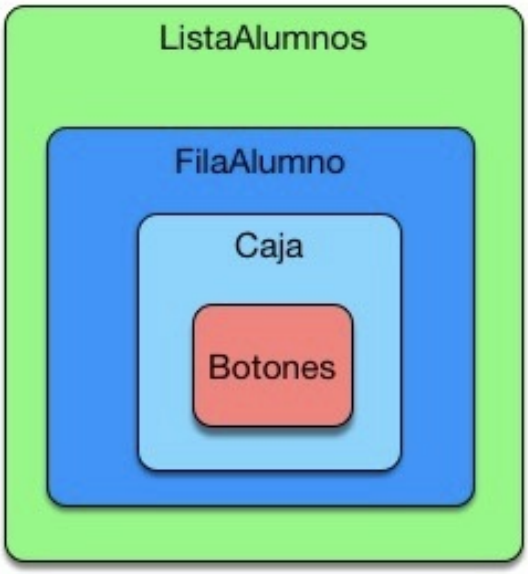
Como podemos ver, se delega en el método ajaxPUTAlumno por cada alumno y se realizan todas las invocaciones de forma simultánea. Para ello, se usa el API de promesas de JavaScript ES6 . Generando por cada item en la lista una petición put.



Este API permite saber cuándo todas las peticiones se han ejecutado a través del método:

```
Promise.all(listaPromesas);
```

Ahora surge una duda importante: **¿Desde dónde invocamos este método para que en el servidor se actualicen todos los datos? .**



JavaScript setInterval

Podemos usar el método componentDidMount y cuando el componente se construya, usar un setInterval para que actualicemos los datos del servidor cada 5 segundos. **Esto sería similar al funcionamiento de Google Drive.**

```
componentDidMount() {
    var componente = this;
```

```
setInterval(function() {
    componente.actualizarAlumnos();
}, 5000);

fetch('/alumnos', {method: 'get'}).then(function(response) {
    return response.json();
}).then(function(datos) {
    componente.setState({lista: datos});

});
```

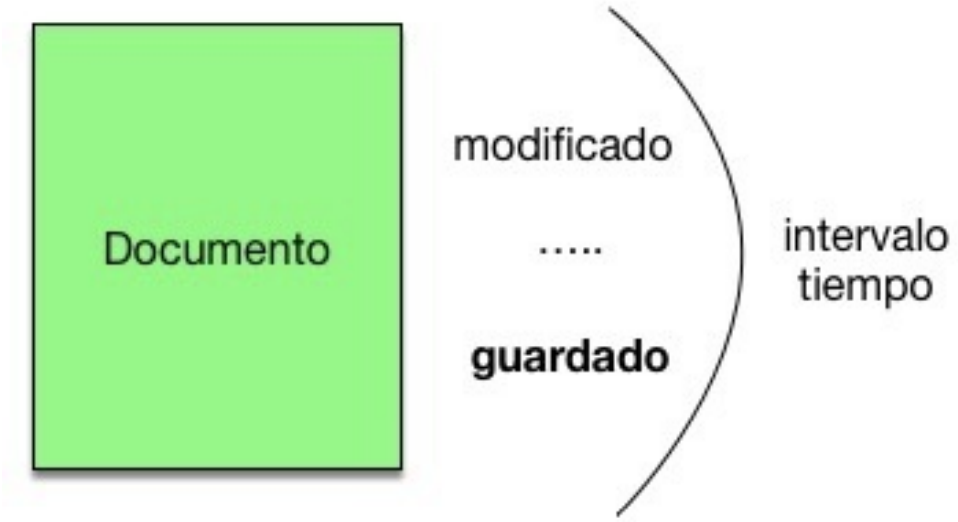
Con este código, cada 5 segundos se invoca al servidor y se actualiza la información que éste almacena.

Podemos ver en con un console.log en node como los datos llegan cada 5 segundos y actualizan la información.

Ya tenemos nuestro componente funcionando y actualizando la información del servidor. Aun así, nos quedan algunos puntos que revisar.

Componentes y Estado

En estos momentos, el componente almacena el estado de una lista de alumnos. Sin embargo, no es tan sencillo trabajar con él como parece . Esto es **debido a que el usuario no sabrá cuando los datos son actualizados en el lado del servidor**. Sería interesante que nuestro componente funcionara al estilo de Google Drive: al hacer un cambio en un documento de Google Drive, en unos segundos nos comunica que los datos se han guardado correctamente.



Vamos a mostrar una nueva versión del componente y comentar los cambios realizados para que pueda cumplir con la funcionalidad requerida.

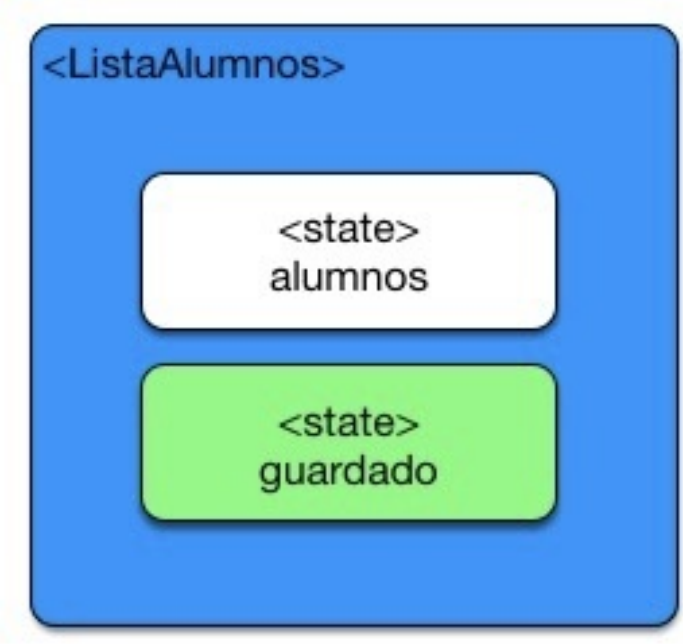
```
var ListaAlumnos = React.createClass({

   getInitialState: function() {

      return {lista: [], guardado: "si"};
   },

   componentDidMount() {
      var componente = this;
      setInterval(function() {
         componente.actualizarAlumnos();
      }, 5000);

      fetch('/alumnos', {method: 'get'}).then(function(response) {
         return response.json();
      }).then(function(datos) {
         componente.setState({lista: datos});
```

```
    });
},
actualizarAlumnos: function() {

    var componente = this;
    var listaPromesas = this.state.lista.map(function(alumno) {
        return componente.ajaxPUTAlumno(alumno);

    })

    Promise.all(listaPromesas).then(function() {
            componente.setState({guardado: "si"});
    });
},

ajaxPUTAlumno: function(alumno) {

    var peticion = fetch('/alumnos/' + alumno.nombre, {
        method: 'put',
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(alumno)
    });
    return peticion;

},

modificarAlumno: function(alumno, indice) {


    var listaNueva = this.state.lista.slice();
    listaNueva.splice(indice, 1, alumno);
    this.setState({lista: listaNueva, guardado:"no"});
},

render: function() {
  let mensajeEstado;

  if (this.state.guardado == "si") {

      mensajeEstado=(
```

```
        <span style={{
              color: 'green'
            }}>
              guardado
            </span>
       )

    } else{

      mensajeEstado=(
        <span style={{
          color: 'red'
        }}>
          modificado
        </span>)
    }
      return (

        <table>
          <tbody>

            {this.state.lista.map(function(alumno, indice) {
              return <FilaAlumno key={alumno.nombre} alumno={alumno} modificarAlumno=
{this.modificarAlumno} indice={indice}/>
            }.bind(this))}
          </tbody>

          <tfoot>
            <tr>
              <td colspan="3">

                {mensajeEstado}

              </td>
            </tr>
          </tfoot>
        </table>

      );
  }
});
```

Hay bastantes cambios ,no son conceptos nuevos pero es importante revisarlos detalladamente. En primer lugar, **hemos añadido estado adicional al componente**. Es decir, ya no se encarga únicamente de almacenar la lista de alumnos sino que ahora también **almacena un estado adicional que se denomina**

“guardado”.



Hemos modificado getInitialState:

```
 getInitialState: function() {

     return {lista: [], guardado: "si"};
  },
```

El siguiente paso es mostrar ese estado al usuario en el componente, de lo cuál se encarga la función render a la que añadimos un bloque if else ,una variable mensajeEstado y un pie de tabla donde se muestra:

```
render: function() {
    let mensajeEstado;

    if (this.state.guardado == "si") {

       mensajeEstado=(

        <span style={{
             color: 'green'
          }}>
             guardado
          </span>
```

```
      )

    } else{

      mensajeEstado=(
        <span style={{
          color: 'red'
        }}>
          modificado
        </span>)
    }
      return (

        <table>
          <tbody>

            {this.state.lista.map(function(alumno, indice) {
              return <FilaAlumno key={alumno.nombre} alumno={alumno} modificarAlumno=
{this.modificarAlumno} indice={indice}/>
            }.bind(this))}
          </tbody>

          <tfoot>
            <tr>
              <td colspan="3">

                {mensajeEstado}

              </td>
            </tr>
          </tfoot>
        </table>

      );
  }
});
```

El resultado lo podemos ver en el navegador, este código admite refactorizaciones, pero lo dejaremos así para mayor claridad:

Cambio de estado

Nos queda ver qué modificaciones hemos hecho a los métodos para que sean capaces de actualizar el mensaje de guardado:

```
modificarAlumno: function(alumno, indice) {

    var listaNueva = this.state.lista.slice();
    listaNueva.splice(indice, 1, alumno);
    this.setState({lista: listaNueva, guardado:"no"});
  },
```

Al modificar alumno, **el cambio ha sido puntual, ya que simplemente hemos añadido que nos modifique el estado de guardado**. Así pues, en cuanto pulsemos cualquiera de los botones ,el componente nos avisará de que los datos han sido modificados:

Vamos a ver los cambios en actualizarAlumno.

```
actualizarAlumnos: function() {

    var componente = this;
    var listaPromesas = this.state.lista.map(function(alumno) {
       return componente.ajaxPUTAlumno(alumno);

    })

    Promise.all(listaPromesas).then(function() {
          componente.setState({guardado: "si"});
    });
  },
```

En este caso, al terminar todas las peticiones, actualizaremos el estado a guardado. Ahora el componente es capaz **de actualizarse cada 5 segundos.**

Ajax y Rendimiento

Todavía podemos afinar un poco más la programación del componente. Ahora mismo, cada 5 segundos el componente hace una petición PUT al servidor. El método será mucho más útil si esas peticiones PUT **se realizan cuando hay un cambio de estado:**

```
actualizarAlumnos: function() {

    var componente = this;

    if (this.state.guardado=="no") {

      var listaPromesas = this.state.lista.map(function(alumno) {
         return componente.ajaxPUTAlumno(alumno);

      })

      Promise.all(listaPromesas).then(function() {
```

```
            componente.setState({guardado: "si"});

        });
      }

  },
```

Ahora las peticiones AJAX sólo se realizan cuando hay cambios. Hemos trabajado durante muchos capítulos utilizando JavaScript y ES5, es momento de avanzar.

React y JavaScript ES6

Hemos construido ya varios componentes y gestionado sus propiedades estado. Pero nuestro código **está realizado con JSX y JavaScript ES5.** JavaScript ES6 aporta muchas novedades en cuanto a modularización y clases. En este capítulo, vamos a modificar los componentes para que se apoyen en ES6. Abordaremos los cambios componente a componente, empecemos con el de Botones.

```
class Botones extends React.Component {

  constructor(props) {
     super(props);

  }
   render() {
      return (
         <span>
            <input type="button" value="+" onClick={this.props.onMas}/>
            <input type="button" value="-" onClick={this.props.onMenos}/>
         </span>
      )
   }
}
Botones.propTypes = {
   onMas: React.PropTypes.func.isRequired,
   onMenos: React.PropTypes.func.isRequired
}
```

A partir de **JavaScript ES6 existe el concepto de clase** , así pues hemos modificado el componente para **que sea directamente una clase y disponga del método render.** Aparte de esto hemos tenido que añadir los propTypes. Vamos a por el siguiente Caja:

```
class Caja extends React.Component {

   constructor(props) {
      super(props);

   }
   onMas() {

      if (this.props.valor < this.props.valorMaximo) {

         this.props.onMas();
```

```
    }
  }
  onMenos() {
    if (this.props.valor > this.props.valorMinimo) {

      this.props.onMenos();
    }
  }

  estilo() {

    if (this.props.valor >= this.props.valorOK) {

      return this.props.estiloOK;

    } else {

      return this.props.estiloNoOK;
    }

  }
  render() {
    return (
      <p>
        <span className={this.estilo()}>{this.props.valor}</span>
        <Botones onMas={this.onMas.bind(this)} onMenos={this.onMenos.bind(this)}/>
      </p>
    );
  }
}

Caja.propTypes = {

  valor: React.PropTypes.number.isRequired,
  valorMinimo: React.PropTypes.number,
  valorMaximo: React.PropTypes.number,
  valorOK: React.PropTypes.number,
  estiloOK: React.PropTypes.string,
  estiloNoOK: React.PropTypes.string,
  onMas: React.PropTypes.func.isRequired,
  onMenos: React.PropTypes.func.isRequired
}
```


Como se puede observar, es muy similar. Sigamos avanzando:

```
class FilaAlumno extends React.Component {

   constructor(props) {
      super(props);
   }

   shouldComponentUpdate(nextProps, nextState) {

      if (this.props.alumno.nota !== nextProps.alumno.nota) {
         return true;
      } else {
         return false;
      }
   }

   incrementarNota() {
      console.log("nota");
      var nuevoAlumno = {};
      nuevoAlumno.nombre = this.props.alumno.nombre;
      nuevoAlumno.nota = this.props.alumno.nota + 1;
      this.props.modificarAlumno(nuevoAlumno, this.props.indice);

   }
   decrementarNota() {

      var nuevoAlumno = {};
      nuevoAlumno.nombre = this.props.alumno.nombre;
      nuevoAlumno.nota = this.props.alumno.nota - 1;
      this.props.modificarAlumno(nuevoAlumno, this.props.indice);

   }
   render() {

      return (
         <tr>
            <td>{this.props.alumno.nombre}</td>
            <td>
               <Caja valor={this.props.alumno.nota} onMas={this.incrementarNota.bind(this)} onMenos=
{this.decrementarNota.bind(this)} valorOK={5} valorMaximo={10} valorMinimo={0}
estiloOK="verde" estiloNoOK="rojo"/></td>
         </tr>
      );
   }
};
```

```
FilaAlumno.propTypes = {

   alumno: React.PropTypes.object.isRequired,
   modificarAlumno: React.PropTypes.func.isRequired
}
```

Por último, el componente de ListaAlumnos:

```
class ListaAlumnos extends React.Component {

   constructor(props) {
      super(props);
      this.state = {

         lista: [],
         guardado: "si"

      }
   }

   componentDidMount() {
      var componente = this;
      setInterval(function() {
         componente.actualizarAlumnos();
      }, 5000);

      fetch('/alumnos', {method: 'get'}).then(function(response) {
         return response.json();
      }).then(function(datos) {
         componente.setState({lista: datos});

      });
   }
   actualizarAlumnos() {

      var componente = this;

      if (this.state.guardado == "no") {

         var listaPromesas = this.state.lista.map(function(alumno) {
            return componente.ajaxPUTAlumno(alumno);

         })
```

```
        Promise.all(listaPromesas).then(function() {

            componente.setState({guardado: "si"});

        });
    }

}

ajaxPUTAlumno(alumno) {

    var peticion = fetch('/alumnos/' + alumno.nombre, {
        method: 'put',
        headers: {
            'Accept': 'application/json',
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(alumno)
    });
    return peticion;

}

modificarAlumno(alumno, indice) {

    var listaNueva = this.state.lista.slice();
    listaNueva.splice(indice, 1, alumno);
    this.setState({lista: listaNueva, guardado: "no"});
}

render() {
    let mensajeEstado;

    if (this.state.guardado == "si") {

        mensajeEstado = (

            <span style={{
                color: 'green'
            }}>
                guardado
            </span>
        )

    } else {
```

```
        mensajeEstado = (
          <span style={{
            color: 'red'
          }}>
            modificado
          </span>
        )
      }
      return (

        <table>
          <tbody>

            {this.state.lista.map(function(alumno, indice) {
              return <FilaAlumno key={alumno.nombre} alumno={alumno} modificarAlumno=
{this.modificarAlumno.bind(this)} indice={indice}/>
            }.bind(this))}
          </tbody>

          <tfoot>
            <tr>
              <td colspan="3">

                {mensajeEstado}

              </td>
            </tr>
          </tfoot>
        </table>

      );
    }
};
```

Ya disponemos de todos los componentes convertidos a ES6 .La funcionalidad de renderizado no cambia.

```
ReactDOM.render(
  <div>
  <ListaAlumnos/>

</div>, document.getElementById('zona'));
```


El resultado no varía ,pero ya tenemos nuestros componentes sobre ES6.

Llegados a este punto, todavía tenemos un problema importante que solventar. Todos los componentes se encuentran dentro del mismo fichero de JavaScript.

Hemos de modificar nuestro código para que sea más modular. Para ello, en el siguiente capítulo usaremos los módulos de ES6 y WebPack.

ES6 y Webpack

El primer paso que tenemos que dar es dividir nuestras clases en 4 ficheros. Veamos el contenido de cada uno de los ficheros: empecemos con los Botones:

(botones.js)

```
import React from 'react';

class Botones extends React.Component {

  constructor(props) {
     super(props);

  }
    render() {
        return (
            <span>
                <input type="button" value="+" onClick={this.props.onMas}/>
                <input type="button" value="-" onClick={this.props.onMenos}/>
            </span>
        )
    }
}
Botones.propTypes = {
    onMas: React.PropTypes.func.isRequired,
    onMenos: React.PropTypes.func.isRequired
}

export {Botones}
```

ES6 y Módulos

El código es idéntico al anterior, salvo que incluye dos nuevas instrucciones: **export e import** .Ambas pertenecen **a ES6 y permiten diseñar una estructura de módulos**. Un modulo es una estructura encargada de almacenar clases y funciones **de forma independiente para su posterior reutilización**. En este caso nuestro módulo importa la librería de React para su uso y exporta la clase Botones para que la puedan usar otros. Vamos a ver el código de Caja:

(caja.js)

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

```
import {Botones} from "./botones";

class Caja extends React.Component {

  constructor(props) {
    super(props);

  }
  onMas() {

    if (this.props.valor < this.props.valorMaximo) {

      this.props.onMas();

    }
  }
  onMenos() {
    if (this.props.valor > this.props.valorMinimo) {

      this.props.onMenos();
    }
  }

  estilo() {

    if (this.props.valor >= this.props.valorOK) {

      return this.props.estiloOK;

    } else {

      return this.props.estiloNoOK;
    }

  }
  render() {
    return (
      <p>
        <span className={this.estilo()}>{this.props.valor}</span>
        <Botones
         onMas={this.onMas.bind(this)}
         onMenos={this.onMenos.bind(this)}/>
      </p>
    );
  }
```

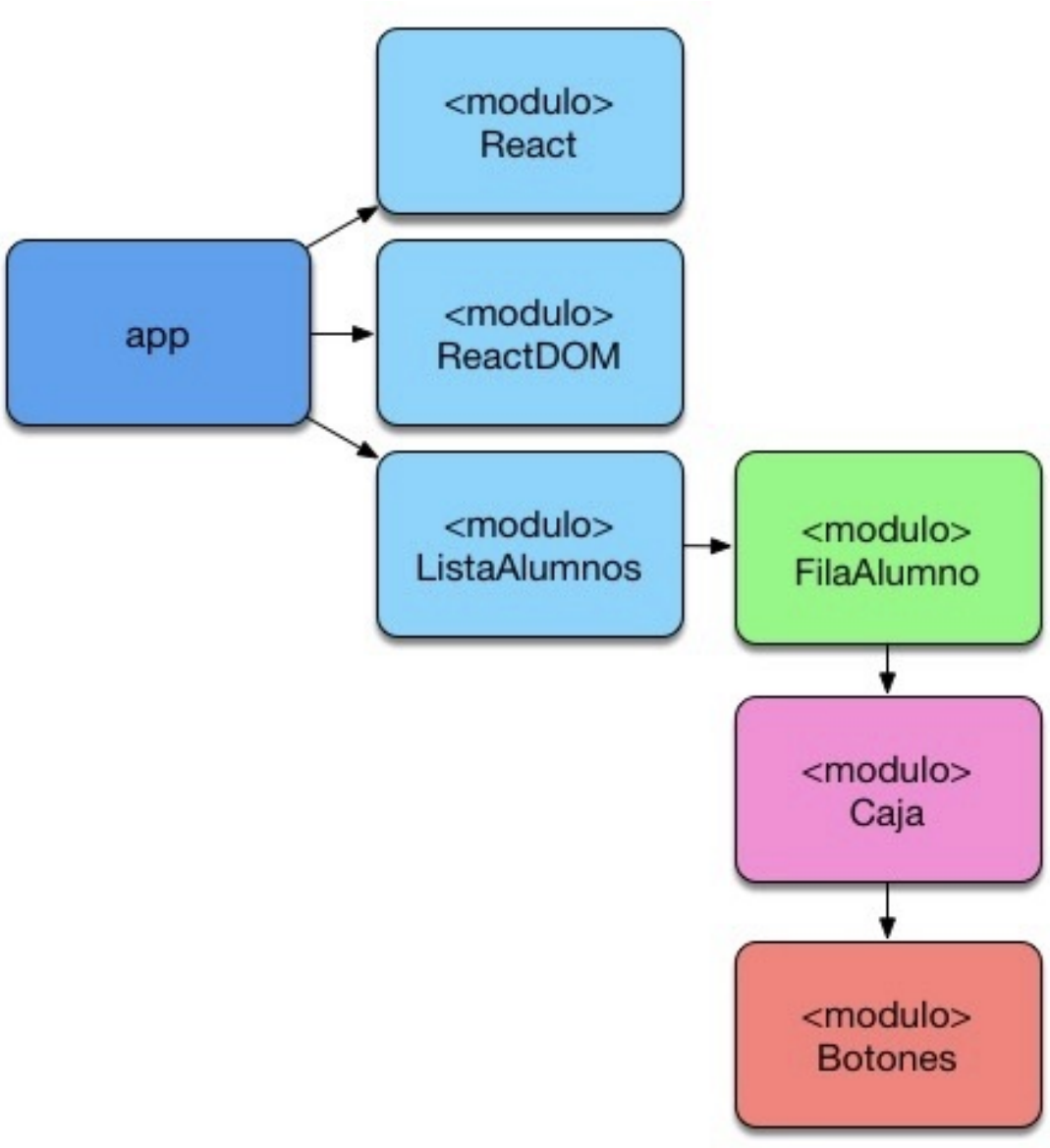
```
}

export {Caja}
```

En este caso importamos el módulo de botones con la clase Botones para usarla y exportamos la Caja. El resto de módulos funciona de forma similar: “filaalumno” importa “caja” y “listaalumnos” importa “filaalumno”. Así pues, nos queda presentar el programa principal que irá en un fichero aparte.

(app.js)

```
import React from 'react';
import ReactDOM from 'react-dom';
import {ListaAlumnos} from './listaalumnos.js';


ReactDOM.render(
   <div>

     <ListaAlumnos/>
   </div>
, document.getElementById('zona'));
```

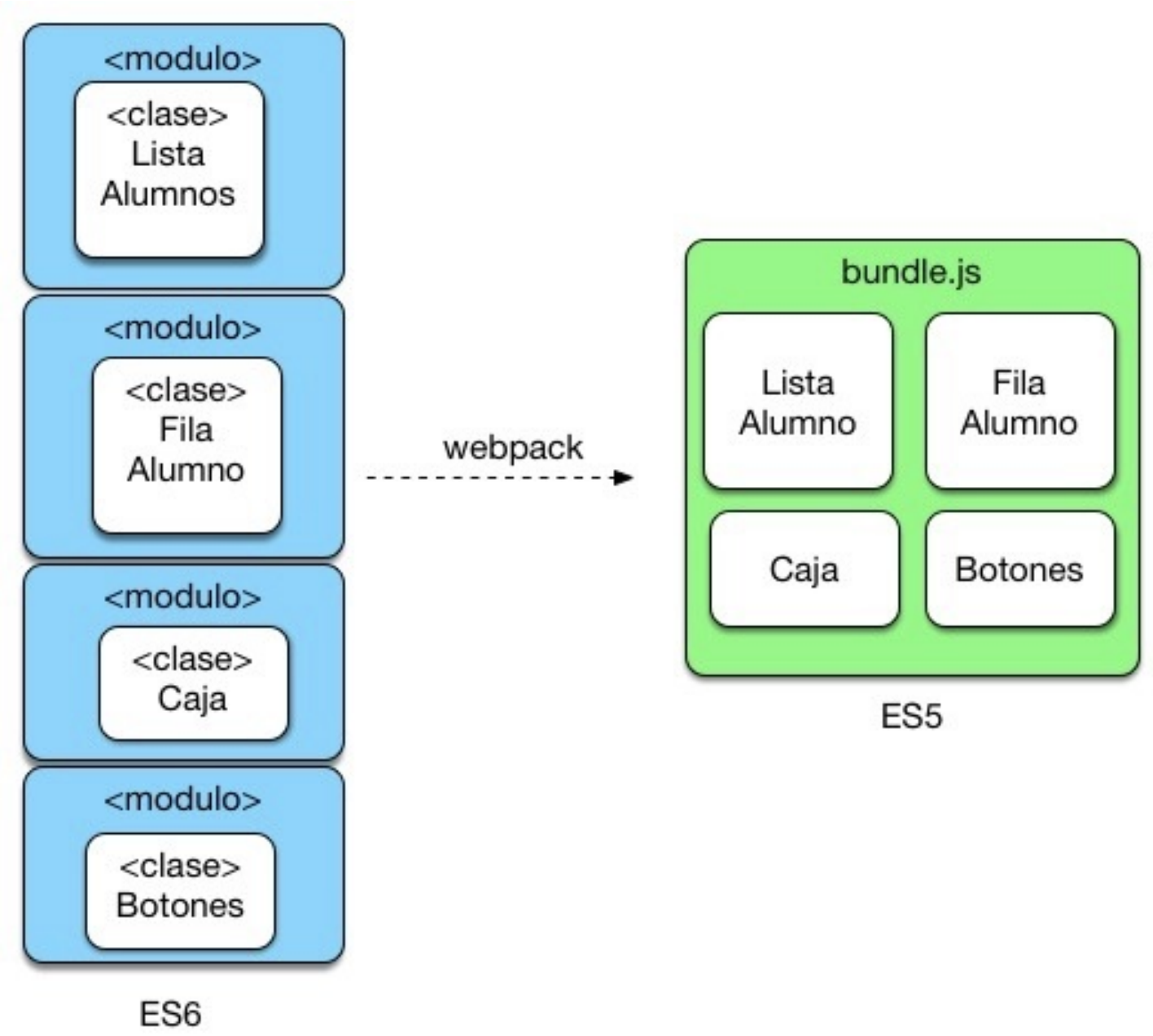
Es aquí donde vemos cómo el programa se encarga de importar los módulos React , ReactDOM y ListaAlumnos. Este último módulo



importa el resto.
Ya tenemos toda la estructura de la aplicación creada. La distribución de módulos pudiera ser distinta e incluir varias clases en un mismo modulo, pero lo hemos dejado así por mantener una disposición más sencilla. Aun así tenemos un inconveniente: toda esta estructura no funciona en un navegador actual ya que los **navegadores no soportan la gestión de ES6 y sus módulos.**

El concepto de bundle

Si queremos que todo el sistema de componentes y módulos funcione, **debemos utilizar una herramienta para empaquetar todos los módulos y clases en un solo fichero que sea entendible por el navegador**. A este tipo de fichero s**e le denomina bundle.**

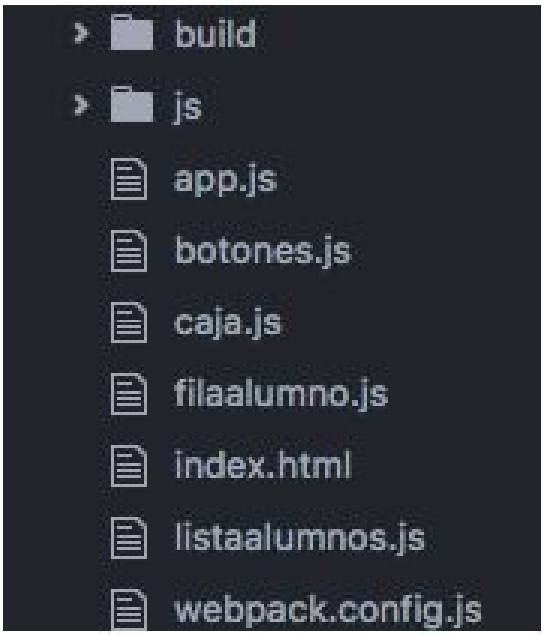


Introducción a Webpack

WebPack **es un empaquetador de módulos:** es lo que en estos momentos necesitamos para que nuestro código se pueda ejecutar en un navegador en un único fichero o bundle. Vamos a instalarla con npm:

npm install webpack -g

Es momento de usar la herramienta para generar un bundle.js a partir de los ficheros originales .



Para usar webpack, debemos crear un fichero de configuración webpack.config.js. Este es su contenido:

```
var config = {
  entry: ['./app.js'],
  resolve: { alias: {} },
  output: {
    path: './build',
    filename: 'bundle.js'
  },
  module: {
    noParse: [],
    loaders: [
      {
        test: /\.js$/
        loader: 'babel-loader',
        query: {
            presets: ['react', 'es2015']
          }
      }
    ]
  }
}

module.exports = config
```

Vamos a explicar cada uno de los parámetros:

entry: Define el fichero de JavaScript principal a partir del cuál se genera el empaquetado, en este caso app.js. Recordemos que el contenido de este fichero es :

```
import React from 'react';
import ReactDOM from 'react-dom';
import {ListaAlumnos} from './listaalumnos.js';
ReactDOM.render(
   <div>

     <ListaAlumnos/>
   </div>
, document.getElementById('zona'));
```

resolve: Se encarga de asignar alias de resolución a nuestros ficheros ,en este caso se usa la configuración por defecto.

output: Este parámetro se encarga de definir la carpeta de destino y nombre del fichero de empaquetado .El fichero **se generará como /build/bundle.js**

test: Las condiciones que deben cumplir los ficheros para ser procesados. En este caso es suficiente con ser ficheros js.

loader: El cargador que se encarga de procesar cada uno de los ficheros ,en este caso usamos **babel ya que trabajamos con ES6.**

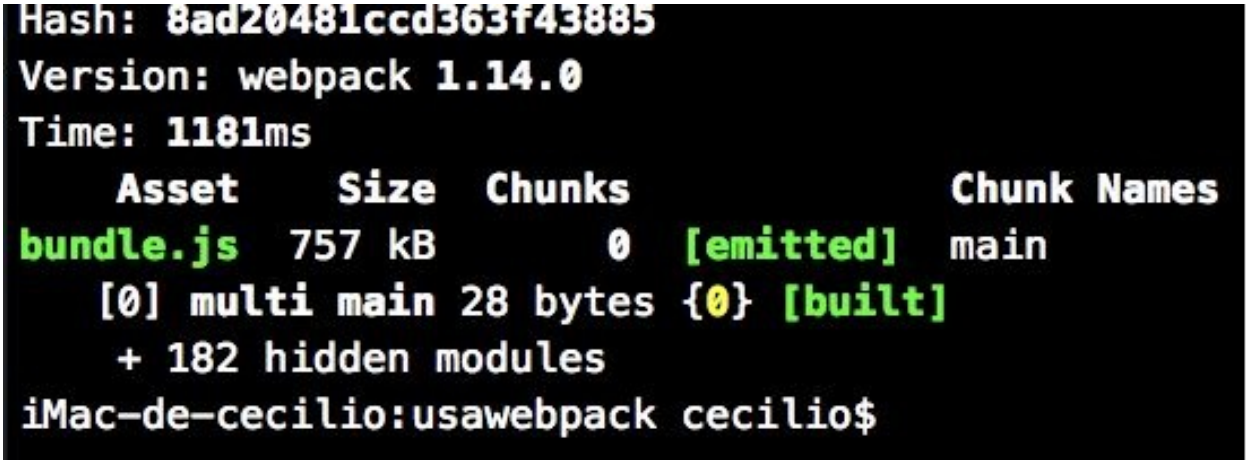
query: Parámetros soportados por el loader, hemos añadido React y es2015 (ES6) para que procese los diferentes ficheros.

Una vez configurado el fichero es tan sencillo como ejecutar webpack en la linea de comandos. Ahora bien, antes de realizar la operación deberemos tener bastantes cosas instaladas con npm , ya que webpack se apoya en ellas:

```
// instalación de  de React y JSX
npm install react
npm install react-dom


// instalación Transpilador Babel
npm install babel-loader
npm install babel-core

//configuracion de babel
npm install babel-preset-es2015
npm install babel-preset-react
```

Una vez instalado todo ejecutamos:
webpack

Esto nos generará un bundle a través de la consola:



Es momento de ver el fichero index.html , que ahora se apoya en el empaquetado:

```
<!DOCTYPE html>
<html>
<body>
   <div id="zona">
   </div>
</body>
   <script type="text/javascript" src="build/bundle.js"></script>
</html>
```

El resultado en pantalla no varía y la aplicación sigue funcionando:

Acabamos de utilizar Webpack para conseguir dos cosas: que nuestra aplicación pueda estar modularizada y que el navegador la pueda ejecutar sin ningún problema.

Resumen

Hemos visto cómo construir componentes en React , cómo asignar propiedades y estado ,cómo integrar unos con otros y cómo abordar una modularización.

Han quedado una serie de cuestiones pendientes: desde temas como Flux ,pasando por React Router o Inmutable.js , pero espero haber podido transmitir la utilidad de los conceptos fundamentales de esta librería y el porqué de su éxito y prometedor futuro.