2 EDICIÓN

REACT 0.14 + ECMASCRIPT 6

DESCUBRE REACT

JAVI JIMENEZ VILLAR

Descubre React

Javi Jimenez

Este libro está a la venta en http://leanpub.com/descubre-react

Esta versión se publicó en 2016-01-08

¡Twitea sobre el libro!

Por favor ayuda a Javi Jimenez hablando sobre el libro en Twitter!

El tweet sugerido para este libro es:

”Descubre ReactJS” es el nuevo libro en español de @soyjavi

El hashtag sugerido para este libro es #descubreReactJS.

Descubre lo que otra gente está diciendo sobre el libro haciendo click en este enlace para buscar el hashtag en Twitter:

https://twitter.com/search?q=#descubreReactJS

También por Javi Jimenez

CoffeeScript

Índice general

Agradecimientos

Escribir un libro en estos tiempos tan sumamente acelerados supone un gran esfuerzo. Principalmente necesitas tiempo y mucha gente no esta dispuesta a concedértelo; es por eso que mientras lees este libro debemos dar las gracias a todas esas personas que me han permitido *gastar* mi tiempo en una de las cosas que considero más importantes en mi vida, transmitir mi conocimiento con la misma pasión que lo obtengo.

Dar las gracias a mi familia por haberme permitido tener la libertad que he necesitado para poder equivocarme, conocerme y ser la persona que soy hoy en día.

Dar las gracias a Catalina por estar conmigo en la mayoría de mis puntos de inflexión como ser humano. Sin ella posiblemente no estaría donde hoy estoy, y este libro nunca se hubiese escrito.

Dar las gracias a todas esas personas anónimas que siguen mis locas aventuras por el mundo y que me animan desinteresadamente a seguir dando lo mejor de mí y a mejorar constantemente.

Dar las gracias también a Peter Hunt, Tom Occhino, Brenton Simpson y todo el equipo encargado de crear React y contribuir a la evolución de todo este mundo de las WWW. Dar las gracias también a los numerosos contribuyentes al proyecto React y a otros proyectos OpenSource, que por suerte son demasiados para nombrarlos uno a uno aquí.

Por último quiero darte las gracias a ti, gracias por adquirir este libro y darme una oportunidad de contarte lo que está suponiendo mi paso a React. Ahora solo te pido una última cosa, comparte este libro cómo lo he hecho yo contigo, regálalo y ayúdame a transmitir este contenido por el mundo.

Preface

React, también conocido como React.js o ReactJS, es una librería JavaScript OpenSource diseñada para crear interfaces de usuario. Ha sido concebida para facilitar el desarrollo de SPA, *Single Page Applications*, obteniendo un gran rendimiento y ofreciendo una forma de desarrollo más cercana a la creación de videojuegos que a la de aplicaciones. Esta librería está mantenida por Facebook, Instagram y una gran comunidad de desarrolladores independientes y corporaciones.

El origen de React comienza en 2011 cuando Pete Hunt y un pequeño equipo en Facebook comienzan a crear un port de XHP, una versión de PHP que Facebook creo meses antes. XHP fue diseñada para minimizar los ataques XSS (Cross Site Scripting), permitiendo una sintaxis XML con el propósito de crear elementos HTML customizables y reusables. Pero apareció un problema no contemplado con XHP: las aplicaciones con tipología SPA requieren de un mayor numero de peticiones al servidor, y XHP no conseguía resolverlo. Con esta problemática, un pequeño grupo de ingenieros de Facebook solicitó a sus jefes intentar utilizar XHP en el navegador usando JavaScript y estos les concedieron seis meses para probarlo. El resultado fue ReactJS. La tecnología desarrollada no se liberó como *OpenSource* hasta el año 2013.

Según palabras del propio Pete Hunt, "durante el último par de años, hemos visto un cambio hacia lo que nos gusta llamar la programación reactiva". Meteor y Angular son ejemplos de ello, pero te aseguro que son totalmente diferentes a React y tienes que aprender a diferenciarlos. Cuando los datos en tu aplicación son actualizados, la interfaz de usuario se actualiza de manera automática reflejando esos cambios, el framework/librería (Angular, meteor, react…) lo gestiona por ti. La diferencia con React es la forma en la que se programa, asemejándose mucho más al motor de un videojuego, puesto que el resto de soluciones ofrecen un sistema de *data-binding*. En un engine de videojuegos, con cada cambio de estado se borra todo lo que sucede en la pantalla y se vuelve a dibujar toda la escena. En cambio con un enfoque (tradicional) de *data-binding* tienes una especie de widgets los cuales el usuario manipularán y se quedarán todo el tiempo en la pantalla. Así que la terminología de esto es que React es un renderizador inmediato y el resto de los sistemas podríamos considerarlos como renderizadores persistentes.

React intenta ayudar a los desarrolladores a crear aplicaciones web complejas que utilizan un gran intercambio de datos. Su principio es sencillo **declarative and composable**. React solo se preocupará de la interfaz de usuario de tu app; si piensas en el paradigma de software Modelo-Vista-Controlador (MVC) React es únicamente la `V`. Esto hace que puedas utilizarlo combinándolo con otras librerías JavaScript o con frameworks como AngularJS, Ember o Backbone. Además React, gracias a su enorme y creciente comunidad, esta generando una gran biblioteca de extensiones, las cuales en muchas ocasiones no se ocupan solo de la interfaz de usuario sino que te ayudarán a complementar las necesidades de tu *App*.

Actualmente React está siendo utilizado por organizaciones como Khan Academy, Netflix, Yahoo, Airbnb, Doist, Facebook, Instagram, Sony, Attlasian y muchas otras. Esto demuestra el gran estado

en el que se encuentra React, con una gran acogida por muchas de las empresas más punteras del sector y asegurando por tanto un desarrollo continuado y lo que es más importante un gran soporte por parte de la comunidad.

Introducción

Actualmente tenemos dos formas de enfrentarnos a un proyecto Web. Una de ellas, llamémosla la *forma tradicional*, soluciona cualquier proyecto utilizando renderizaciones desde el servidor utilizando frameworks como Symphony (PHP), Rails (Ruby) o Django (Python) y jugueteando con jQuery (JavaScript) en el cliente. La otra solución, llamémosla SPA, *Single Page Application*, utiliza al servidor para *renderizar* objetos JSON los cuales serán tratados por un framework o librería JavaScript en el cliente como pueden ser Backbone, Ember, Angular o Atoms. Ahora mismo seguramente te podrás identificar en uno de estos dos grupos, o tal vez te haya tocado estar en ambos.

Con React todo cambia ya que su principal premisa es "solo me preocupo de la Interfaz de Usuario". Lo que quiere decir es que no vamos a tener *routers*, *models*, *bindings*, *observers* o *templates* y en el caso de que los necesites podrás utilizar cualquier otro framework o librería. Este es un punto clave, después de llevar más de 4 años creando librerías como QuoJS[1], TukTuk[2] o frameworks como LungoJS[3] y Atoms[4] los cuales son utilizados por miles de desarrolladores y empresas por todo el mundo, puedo asegurar que tal vez estaba equivocado con los frameworks. A pesar de crear frameworks sumamente sencillos de utilizar, o eso creía, puede que tu naturaleza como desarrollador sea totalmente diferente a la mía. Todos los frameworks hasta la fecha han intentado empaquetar el mayor número de funcionalidades posibles, imponiendo de una manera la forma de desarrollar. Soy un devoto del `<code/>` y en mi pasado me leí los *internals* de jQuery, Angular o Ember (todos tenemos un pasado oscuro) y es por eso que decidía crear mis propias soluciones. Estaba tan convencido de que podía hacerlo mejor creando mi propio acercamiento, pero en realidad los desarrollaba porque no estaba cómodo con el *workflow* de trabajo que ofrecían el resto de soluciones. Si lo piensas bien la mayoría de los frameworks han sido creado por un equipo (o un loco) que lo necesitaba para un determinado proyecto, pero deberías saber que no existe el framework perfecto que pueda solucionar cualquier proyecto, solo existe el programador pragmático que sabe decir "esta vez no vamos a utilizar Angular/Ember/jQuery".

Por eso me reafirmo en decir que el equipo React ha sido muy inteligente a la hora de centrarse en una única area, una de las más importantes, la interfaz de usuario. Tu tendrás la capacidad y libertad de decidir que complementos quieres utilizar en tu solución web, ¿necesitas gestos táctiles? podrás utilizar QuoJS, FastClick o HammerJS, ¿necesitas un enrutador? podrás utilizar director, react-router, SPArouter o crear el tuyo propio. Después de dedicar mucho tiempo a crear aplicaciones con React no he echado en falta absolutamente nada y me ha ofrecido la suficiente autonomía de decisión para utilizar lo que realmente necesito en cada una de mis aplicaciones.

Ahora te tienes que quedar con 3 claves fundamentales que React hace diferente al resto de sistemas:

[1] https://github.com/soyjavi/quojs
[2] https://github.com/soyjavi/tuktuk
[3] https://github.com/soyjavi/lungojs
[4] https://github.com/tapquo/atoms

- Renderiza todo con cada cambio
- Virtual DOM
- Eventos sintéticos

Renderiza todo con cada cambio

React, como he dicho, no necesita de Observers, Bindings y mucho menos usar Dirty-Checking Models (un saludo Angular :)). React es funcional y todo componente creado con esta librería debe ser capaz de autogestionarse, debe saber cuando volver a renderizarse ya sea por cambios de estado o por paso de propiedades heredadas.

Virtual DOM

El Virtual DOM a mi parecer es una de las mejores características que trae consigo React. Aunque parezca increíble en mi framework Atoms, diseñé algo parecido que llamé *pseuDOM* que no deja de ser una referencia al DOM en memoria. Pero React lo hace mucho mejor, ya que contiene en memoria toda la estructura de datos, todo el sistema de eventos sintéticos y un gestor de memoria que ya me hubiese gustado implementarlo en Atoms. Para que te hagas una idea con cada cambio en un componente React sucede lo siguiente:

- Se genera un nuevo árbol Virtual DOM
- Se compara con el árbol previo
- Se deciden cuales son los cambios mínimos a realizar
- Se mandan esos cambios a la cola
- Se procesan los cambios en el navegador

Resumiéndolo de una manera sencilla, trata el DOM como si de una GPU se tratará. Pero además el Virtual DOM trae unos cuantos ases en la manga, es muy fácil crear componentes testables, puedes renderizar estos desde tu servidor y tienes soporte a elementos HTML5 como `svg` y `canvas`.

Los eventos sintéticos

React implementa su propio sistema de eventos, por lo que no tendrás que preocuparte por tener jQuery, QuoJS o demás librerías de manejo de DOM. Como he comentado al comienzo de este capítulo, solo tendrás que utilizarlo si realmente lo necesitas. React crea un único manejador de evento nativo en el nivel superior de la estructura de cada componente. Este sistema de eventos está normalizado para que sea funcional en todos los navegadores e incluso es capaz de diferenciar entre eventos desktop (*click*, *double click*, *drag*…) y eventos móviles (*touchstart*, *touchend*…). Ni que decir que todos esos eventos están desacoplados del DOM ya que realmente tienen una referencia directa con el Virtual DOM, no volverás a ver un atributo `onClick` en ninguno de tus elementos HTML.

Configurando tu entorno React

Para comenzar con React voy a intentar facilitarte la vida, vas a aprender a desarrollar tu primer proyecto React de varias maneras, las cuales podríamos definirlas como:

- La manera *rápida*: Utilizando jsfiddle.net
- La manera *sencilla*: En el propio documento HTML.
- La *mejor* manera: Utilizando un sistema de gestión de procesos como pueden ser WebPack[5], Gulp[6] o GruntJS[7]

La manera rápida: JSFiddle

He creado un pequeño proyecto en JSFiddle que te ayudará de una manera sencilla a dar tus primeros pasos. Para ello únicamente tienes que acceder a la url http://jsfiddle.net/soyjavi/tLpuv6p7/[8]. El proyecto se divide en 2 únicos ficheros:

Un fichero `.html`:

```
<script src="https://facebook.github.io/react/js/jsfiddle-integration.js"></scri\
pt>

<div id=“container”></div>
```

Un fichero `js`:

```
var Hello = React.createClass({
  render: function() {
    return <div>Hello {this.props.name}</div>;
  }
});

ReactDOM.render(<Hello name='React' />, document.getElementById('container'));
```

[5] http://webpack.github.io/
[6] http://gulpjs.com/
[7] http://gruntjs.com/
[8] http://jsfiddle.net/soyjavi/tLpuv6p7/

Te recomiendo que hagas un *fork* del proyecto para que puedas hacer todas las variaciones que te interesen. De todas formas adelantarte que en este libro utilizaremos EcmaScript 6 por lo que el método `createClass` no lo utilizaremos en los siguientes capítulos, en *pro* de usar las clases nativas que ofrece la *nueva* version de JavaScript. Desgraciadamente *jsfiddle* todavía no soporta EcmaScript 6 sino pasamos antes por algún precompilador como puede ser BabelJS.

La manera sencilla: En el documento HTML

Si el JSFiddle no es suficiente, obvio por otra parte, podrías comenzar a jugar con React creando un documento HTML, y aunque te parezca increíble, va a ser el único código HTML que escribirás y leerás en todo el libro:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Descubre React</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.0/react.js"><\
/script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.0/react-dom.j\
s"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browse\
r.min.js"></script>
  </head>
  <body></body>
</html>
```

Como puedes comprobar solo estamos haciendo referencia a dos ficheros ficheros *JavaScript*:

- **`react.js`**: La librería principal de ReactJS.
- **`react-dom.js`**: Desde React 0.14 el manejador de DOM de tus componentes React se realiza con esta libreria JavaScript.
- **`browser-min.js`**: Una version minima de BabelJS para el navegador.

Vamos a utilizar React 0.14 y en el caso de que quieras descargar nuevas versiones puedes dirigirte a la documentación[9] oficial.

Ahora vamos a escribir nuestra *super-fantástica* primera aplicación React para ver que todas las referencias funcionan correctamente. Para ello y desbordando una gran originalidad vamos a escribir por pantalla el ya casi centenario `Hello World`. Escribiremos el siguiente código dentro de la etiqueta `<body>`:

[9] https://facebook.github.io/react/downloads.html

```
...
<body>
  <div id="container"></div>
    <script type="text/babel">
      const Hello = class Button extends React.Component {
        render () {
          return <div>Hello {this.props.name}!</div>;
        }
      }

      ReactDOM.render(
        <Hello name='React' />,
          document.getElementById('container')
      );
    </script>
</body>
...
```

Ahora nos vamos a centrarnos en la propia sintaxis de React, sino en que nuestro proyecto muestre 'Hello World'. ¿Lo hace? ¿Sí? Bien ahora vamos a *limpiar* un poco el código, porque siempre me ha parecido muy maligno eso de escribir código JavaScript en un documento HTML. Vamos a crear un nuevo fichero:

Como ves la etiqueta `script` tiene definida el atributo `type` con el valor `text/babel` y no el tan conocido `text/javascript`, esto es debido a que a partir de ahora vamos a escribir código EcmaScript6 con Babel. Lo empezamos a hacer definiendo nuestra primera clase `Hello` extendiendo de la clase primitiva `React.Component`. Como habrás podido adivinar BabelJS ejecuta toda la *transpilación* entre ES6 y JavaScript desde tu navegador y es por eso que en la siguiente sección comprenderás porque es mejor utilizar un gestor de procesos que genere ficheros JavaScript precompilados.

La mejor manera: WebPack

Cuando te enfrentas a un proyecto JavaScript, sea del tamaño que sea, la anterior solución nunca te serviría para tener un flujo de trabajo óptimo y mantenible. A lo largo de mi vida como *developer* me encuentro habitualmente con que existen proyectos donde los desarrolladores no utilizan ningún tipo de proceso de empaquetado. Van haciendo referencias simbólicas a cada fichero JavaScript, o lo que tampoco es mucho mejor escriben un fichero enorme con miles y miles de lineas JavaScript, resumiendo son unos locos *Cowboys* del `code`.

Desde hace ya varios años tenemos herramientas que nos ayudan a gestionar nuestros diferentes procesos en el ciclo de vida de un proyecto:

- *Lintar* nuestro código JavaScript
- Minificar y ofuscar el código
- Transpilar código como CoffeeScript, scss, Stylus, Less...
- Concatenar archivos
- Testear
- ...

Para ello puedes utilizar diferentes gestores de procesos, cada uno con sus diferencias y propia naturaleza. Los gestores más conocidos actualmente son GruntJS[10], GulpJS[11] y el último en llegar WebPack[12] (de la propia Facebook). Con todos ellos podrás realizar las tareas que he citado antes, pero mi recomendación es que utilices WebPack, ya que con diferencia es el más rápido y sencillo de utilizar y evidentemente el que mejor se integra con React.

Antes de comenzar tengo que añadir que la tendencia hoy en día es que tus ficheros JavaScript estén modularizados. Esta es una práctica que se lleva utilizando mucho en el backend con servidores NodeJS y que desde hace un tiempo se está llevando al cliente pudiendo reutilizar código en ambos contextos. Deberás profundizar en este area y para ello te recomiendo que comiences con CommonJS[13] y Browserify[14]. Por ponerte en contexto, un ejemplo realmente sencillo sería este:

someModule.js

```
1  module.exports.doSomething = function() {
2    return 'foo';
3  };
```

otherModule.js

```
1  var someModule = require('someModule');
2
3  module.exports.doSomething = function() {
4    return someModule.doSomething() + 'bar';
5  };
```

Ahora que te has convertido en un *Jedi* no utilizando prácticas *Cowboy*, conozcamos como puedes comenzar a trabajar con WebPack. El proceso es muy sencillo, se basa en crear un fichero en la raiz de tu proyecto llamado `webpack.config.js` en el que definiremos nuestros procesos:

webpack.config.js

[10] http://gruntjs.com/
[11] http://gulpjs.com/
[12] http://webpack.github.io/
[13] https://www.wikiwand.com/en/CommonJS
[14] http://browserify.org/

```
var pkg = require('./package.json');

module.exports = {
  resolve: {
    extensions: ['', '.jsx', '.js']
  },

  entry:  './app.jsx',

  output: {
    path: './build',
    filename: pkg.name + '.js'
  },

  module: {
    loaders: [
      {
        test: /(\.js|\.jsx)$/,
        loader: 'babel',
        query: { presets: ['es2015', 'stage-0', react'] }
      }
    ]
  }
};
```

En posteriores capítulos veremos con más detalle como utilizar Webpack por lo que ahora solo te voy a explicar de manera introductoria que es lo que está haciendo nuestro nuevo querido gestor de procesos:

- Solo va a tener en cuenta los ficheros con extensions `.js` o `.jsx`
- El fichero de inicio de nuestra aplicación estará en la raíz y se llamará `app.jsx`
- Una vez compile todo el proyecto dejará un fichero en la carpeta *build* con la concatenación del atributo name que tenemos en nuestro fichero `package.json` y la extensión `js`.
- Además como vamos a escribir tanto EcmaScript6 (es2015) como sintaxis JSX hemos definido un loader con Babel que se encargará de hacer la transpilación a código React JavaScript.

Increíble, pero cierto, me ha costado menos definir el fichero `webpack.config.js` que explicar todo lo que hace, que es mucho. Como has podido descubrir es un fichero `package.json` y eso solo puede indicar una cosa, vamos a utilizar NodeJS en el cliente para obtener módulos y ejecutar todas estas tareas. Este sería nuestro fichero:

package.json

```
{
  "name": "descubre-react",
  "version": "0.2.0",
  "main": "./build/descubre-react.js",
  "license": "MIT",
  "devDependencies": {
    "babel-core": "^6.1.4",
    "babel-loader": "^6.0.1",
    "babel-plugin-react-transform": "^1.1.1",
    "babel-preset-es2015": "^6.1.4",
    "babel-preset-react": "^6.1.4",
    "babel-preset-stage-0": "^6.1.4",
    "react": "^0.14",
    "react-dom": "^0.14.0",
    "webpack": "^1.9.10"
  }
}
```

Como ves las únicas dependencias que hemos añadido han sido `babel` (y sus modules), `react`, `react-dom` y `webpack`, ¿sencillo verdad?. Ahora vamos a escribir nuestro ejemplo `Hello World` utilizando un sistema de módulos. Lo primero que vamos a hacer es crear el archivo `hello.jsx` que ahora si que contendrá nuestro primer componente React escrito en EcmaScript 6:

hello.jsx

```
import React from 'react';

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

export default Hello;
```

El segundo, y último paso, será hacer referencia al componente contenido en `hello.jsx` en un nuevo fichero llamado `app.jsx`:

app.jsx

```
import React from 'react';
import ReactDOM from 'react-dom';
import Hello from './chapter-0/hello';

ReactDOM.render(<Hello name='React' />, document.getElementById('container'));
```

Con esta definición una vez que ejecutemos webpack nos generará un único fichero *JavaScript* con el nombre `descubre-react.js` en el directorio `build`. Para facilitar las cosas podrás ejecutar en tu consola el comando `webpack -watch` que estará alerta a cualquier cambio que realices en tu código para autocompilar los cambios. Nuestro fichero `index.html` quedara reducido de la siguiente manera:

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Descubre React</title>
  </head>
  <body>
    <div id="container"></div>
    <script src="./build/descubre-react.js"></script>
  </body>
</html>
```

Por otra parte, no te voy a explicar que es eso de `React.createClass` o `React.Component` ni porque utilizo la etiqueta `<Hello/>` eso lo vamos a dejar para el siguiente capítulo donde comenzarás a descubrir la verdadera magia de React.

Tu primer React Component

Un componente React encapsula absolutamente todo, no encontrarás una división en varios fichero entre la vista y la lógica de la misma. En mi honesta opinión nunca tuvo mucho sentido separar esos contextos cuando estás construyendo interfaz de usuario (de ahí que cada acercamiento MVC en el *frontend* sea tan diferente). La vista y su lógica inevitablemente tienen que ir acopladas y no tendremos que ir conmutando entre el fichero de *template* y una especie de vista-controlador que actúa de pegamento. Los componentes React normalmente serán lo suficientemente pequeños para que no tengas la necesidad de dividir nada, y en el caso de que sea así posiblemente sea mejor dividir tu componente en componentes más pequeños (y reutilizables).

Llevas leyendo varios capítulos y todavía no has podido interiorizar la creación de componentes. Veamos como crear nuestro primer componente:

hello-example-1.js

```
1  import React from 'react';
2
3  class Hello extends React.Component {
4    render() {
5      return React.DOM.h1(
6        {className: 'mystyle'}, `Hello ${this.props.name}!`
7      );
8    }
9  };
10
11 export default Hello;
```

Como ves hemos creado una clase `Hello` que extiende de la clase base `React.Component`. En nuestra clase definimos el método `render()`, este método es obligatorio y nativo de React en el cual devolvemos un elemento `<h1>` con el atributo `class` rellenado con el valor `mystyle`. Además el contenido HTML que contiene nuestro `<div>` es el resultado de una interpolación EcmaScript 6 de `Hello` con el valor de la propiedad `name`.

app.jsx

```
1  import Hello from './chapter-1/hello-example-1';
2
3  ReactDOM.render(<Hello name='React' />, document.getElementById('container'));
```

Una vez tenemos la definición de nuestro componente `Hello` solo nos queda incluirlo en el DOM y para ello utilizamos el método `ReactDOM.render`. A este método debemos pasar como parámetros el componente que queremos utilizar y las propiedades con las que queremos instanciarlo, y como segundo parámetro enviaremos el nodo donde queremos renderizar el componente, en nuestro caso un elemento con el id `container`. Se que todavía no entenderás mucho y sobre todo intentarás compararlo con lo que conoces hasta ahora, pero ya te avisé; React es diferente y me gustaría acercarte su sintaxis lo más rápidamente posible.

Aunque puedes comprender fácilmente lo que hace la función `React.DOM.div` (crea un elemento `<div>` en el DOM) el equipo de React ha pensado en crear una sintaxis mucho más sencilla y cercana a nuestro querido lenguaje HTML, JSX[15]. A lo largo de este libro podrás conocer con más detalle JSX ahora solo quiero que veas nuestro componente `Hello` con esta sintaxis:

hello-example-2.jsx

```
1  import React from 'react';
2
3  class Hello extends React.Component {
4    render() {
5      return <h1 className='myStyle'>Hello {this.props.name}!</h1>
6    }
7  };
8
9  export default Hello;
```

Cómo puedes comprobar es una sintaxis mucho más legible y cercana a lo que estamos acostumbrados. Aunque se perfectamente que te parece muy extraño escribir esta especie de HTML sobre alimentado que rompe con todo lo establecido hasta ahora, te puedo asegurar que te acostumbrarás y que con el tiempo lo amarás. Algo que es muy agradable de utilizar son las inserciones de código JavaScript dentro del propio JSX utilizando los caracteres `{` y `}`.

Mientras que el resto de framework o librerías continúan introduciendo JavaScript en el HTML, React hace todo lo contrario incluye HTML en su JavaScript.

Con estos ejemplos, tienes que haberte dado cuenta de un punto que diferencia a React del resto de librerías y frameworks; normalmente **no escribirás código HTML en un fichero html** y tampoco definirás ficheros con un sistema de *templates*.

Como era de esperar, y siendo un total evangelizador de CoffeeScript[16], veremos el mismo ejemplo con sintaxis JSX y mi lenguaje preferido:

hello.cjsx

[15] https://facebook.github.io/react/docs/jsx-in-depth.html
[16] http://coffeescript.org

```
Hello = React.createClass
  render: -> <h1 className='myStyle'>{"Hello #{@props.name}"}</h1>
```

En este libro no vas a ver más CoffeeScript, muy a mi pesar, pero te puedo asegurar que es un lenguaje que te puede facilitar mucho las cosas cuando te enfrentas a proyectos con miles y miles de lineas de código. Si vienes de lenguajes como Ruby o Python te costará una mañana aprender a utilizarlo y te puedo asegurar que el código JavaScript resultante es simplemente perfecto. Si quieres profundizar más sobre este lenguaje tienes disponible mi libro totalmente gratuito en la plataforma Leanpub[17].

Etiquetas HTML y componentes

Como has podido comprobar todos los componentes creados en React generan un símbolo a la instancia por medio de una pseudo-etiqueta HTML. En nuestro caso, el componente `<Hello>` a la hora de renderizarse en el DOM se convierte en un elemento HTML con la etiqueta `<div>`. La convención en React recomienda utilizar el formato *BumpyCase* cuando hagamos referencia a componentes React y *LowerCase* para los elementos HTML nativos. Ahora veamos un ejemplo en el que nuestro componente `<Hello>` estará contenido en otro componente React y lo combinaremos con etiquetas nativas HTML:

messages.jsx

```
import React from 'react';
import Hello from './hello-example-2';

class Messages extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello React</h1>
        <Hello name='Javi' />
      </div>
    )
  }
};

export default Messages;
```

Lo que tienes que tener en cuenta en todo momento es que todos los componentes de React que escribas con sintaxis JSX tienen que estar contenidos bajo un nodo. En el caso del componente

[17] http://leanpub.com/coffeescript

`<Messages>` utilizamos la etiqueta `<div>` como contenedor. Con esto quiero decir que si hubiésemos escrito la siguiente sintaxis:

messages_invalid.jsx

```
...
class Messages extends React.Component {
  render() {
    return (
      <h1>Hello React</h1>
      <Hello name='Javi' />
    )
  }
};
...
```

En este caso React solo renderizaría el elemento `<h1>` ya que es el primer nodo que tiene inicio `<>` y fin `</>`. Esto es muy importante que lo interiorices ya que normalmente es un foco muy generalizado de errores. Ahora volvamos al ejemplo, como puedes ver el componente `<Messages/>` combina una etiqueta nativa como es `<h1>` junto con nuestro componente `<Hello>` siendo su renderización en el DOM de la siguiente manera:

```
<div>
  <h1>Hello React</h1>
  <h1>Hello Javi</h1>
</div>
```

Ahora seguramente te estarás preguntando, ¿por qué debería utilizar un componente React para elementos sencillos?, la respuesta es muy sencilla. Utilizando componentes React puedes además de encapsular la funcionalidad necesaria, puedes también crear una definición de interfaz más sencilla y transparente a las etiquetas HTML resultantes. Para ello vamos a crear un nuevo component `<Link>` que va a actuar de *wrapper* del elemento nativo `<a>`:

link.jsx

```
import React from 'react';

class Link extends React.Component {
  render() {
    return <a href={this.props.url}>{this.props.caption}</a>
  }
};

export default Link;
```

Como vemos nuestro componente tiene una propiedad `url`, que pasa a ser utilizado internamente como el atributo nativo `href` , y la propiedad `caption`, que pasa a ser el valor literal del elemento `<a>`. Ahora vamos a incluirlo en nuestra `app`:

app.jsx

```
import React from 'react';
import ReactDOM from 'react-dom';
import Messages from './chapter-1/messages';
import Link from './chapter-1/link';

class App extends React.Component {
  render() {
    const url = 'http://soyjavi.com';

    return (
      <div>
        <Messages />
        <nav>
          <Link url={url} caption='My site' />
          <Link url='https://github.com/soyjavi' caption='My github' />
        </nav>
      </div>
    )
  }
};

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

Como ves nuestra clase `App` ahora contiene tanto el componente `Messages` como varias instancias de `Link` donde el resultado de la renderización de nuestro componente será:

```
<div>
  <h1>Hello React</h1>
  <h1>Hello Javi!</h1>
  <nav>
    <a href="http://soyjavi.com">My site</a>
    <a href="https://github.com/soyjavi">My github</a>
  </nav>
</div>
```

Ahora vamos a imaginar que queremos cambiar la definición JSX de nuestro componente `link` a la siguiente estructura:

link.jsx

```
var Link = React.createClass({
  render: function() {
    return (
      <a href={this.props.url}>
        <label>{this.props.caption}</label>
      </a>
    );
  }
});
```

La magia de utilizar los componentes react es que nuestras propiedades seguirán siendo las mismas pero el resultado HTML en nuestro DOM será algo diferente. En el siguiente capítulo aprenderás a trabajar con las propiedades de tus componentes React, ahora solo me queda recordarte que las etiquetas HTML que podrás utilizar en tus componentes son:

```
a abbr address area article aside audio b base bdi bdo big blockquote body br
button canvas caption cite code col colgroup data datalist dd del details dfn
dialog div dl dt em embed fieldset figcaption figure footer form h1 h2 h3 h4 h5
h6 head header hr html i iframe img input ins kbd keygen label legend li link
main map mark menu menuitem meta meter nav noscript object ol optgroup option
output p param picture pre progress q rp rt ruby s samp script section select
small source span strong style sub summary sup table tbody td textarea tfoot th
thead time title tr track u ul var video wbr
```

Atributos soportados

React soporta todos los atributos `data-*` y `aria-*` así como todos los atributos de la siguiente lista:

```
accept acceptCharset accessKey action allowFullScreen allowTransparency alt
async autoComplete autoFocus autoPlay cellPadding cellSpacing charSet checked
classID className colSpan cols content contentEditable contextMenu controls
coords crossOrigin data dateTime defer dir disabled download draggable encType
form formAction formEncType formMethod formNoValidate formTarget frameBorder
headers height hidden high href hrefLang htmlFor httpEquiv icon id label lang
list loop low manifest marginHeight marginWidth max maxLength media mediaGroup
method min multiple muted name noValidate open optimum pattern placeholder
poster preload radioGroup readOnly rel required role rowSpan rows sandbox scope
scoped scrolling seamless selected shape size sizes span spellCheck src srcDoc
srcSet start step style tabIndex target title type useMap value width wmode
```

Hay que tener únicamente cuidado con el atributo `class` (utilizado para establecer clases CSS), ya que es una palabra reservada dentro de JavaScript, por tanto tienes que utilizar la propiedad `className`. Veamos un ejemplo extendido con nuestro componente `<Link>`:

link.jsx

```
var Link = React.createClass({
  render: function() {
    return (
      <a href={this.props.url} className={this.props.color}>
        <label>{this.props.caption}</label>
      </a>
    );
  }
});
```

Lo que hemos hecho es crear una propiedad `color` que establece el atributo `className` del elemento HTML `<a>`. Con esto si esta propiedad la *seteamos* con el valor `'red'`, su renderización HTML sería:

```
<a href="http://soyjavi.com" class="red">My Site</a>
```

Lo que me gustaría que entendieses tras leer este capitulo es que tienes que comenzar a diferenciar entre la definición de propiedades de tu componente y el resultado de su renderización en atributos HTML. Una vez tengas claro que tus componentes pueden tener todo tipo de propiedades con el nombre que decidas, excepto `class`, y que estas nunca se renderizarán a menos que establezcas sus valores en atributos HTML... podemos pasar al siguiente capítulo.

Propiedades

En el anterior capítulo comenzaste a descubrir como se crean componentes básicos con React, fue un pequeño avance y estoy seguro que mucha de la sintaxis te resultaba extraña. Por ejemplo en el componente `<Link>` asignábamos propiedades que actuaban de `wrapper` de atributos HTML, pero el uso de estas propiedades va mucho más allá.

Lo primero que tienes que entender es que las propiedades son parámetros inmutables que se reciben siempre desde un componente *Padre*. El propio componente nunca podrá modificar sus propiedades, y la única manera de modificarlas será desde el *padre* cuando este ejecute una nueva renderización del componente *Hijo* el cual recibirá las nuevas propiedades. Podemos decir entonces que las propiedades siempre se establecen desde componentes superiores, *Padres*, a componentes inferiores, *Hijos*.

Ahora vamos a ver un ejemplo en el que un componente `<ToDo>` contiene 3 instancias del componente `<Task>` cada uno de ellas con diferentes valores en sus propiedades `name` y `done`:

todo.jsx

```
1  import React from 'react';
2  import Task from './task'
3
4  class ToDo extends React.Component {
5    render() {
6      return (
7        <ul>
8          <Task name='Introduction' done />
9          <Task name='Chapter 1 - First component' done />
10         <Task name='Chapter 2 - Properties' done={false} />
11       </ul>
12     )
13   }
14 };
15
16 export default ToDo;
```

task.jsx

```
import React from 'react';

class Task extends React.Component {
  render() {
    return (
      <li>
        <input type="checkbox" checked={this.props.done ? 'checked' : null} />
        {this.props.name}
      </li>
    )
  }
};

export default Task;
```

Como puedes ver en nuestro componente *Padre* `<ToDo>` hemos establecido una lista de tareas, algunas de ellas ya finalizadas usando la propiedad `done`. Si ahora te fijas en el componente *hijo* `<Task>` ves que el uso de las propiedades `name` y `done` es realmente sencillo, solo tienes que hacer referencia al objeto `this.props`. Este objeto contiene todas las propiedades establecidas en el componente y en nuestro caso usaremos una ternaria dentro de nuestra sintaxis JSX, entre `{` y `}`. Es en este punto donde comenzarás a amar JSX, si nos fijamos en el uso de la propiedad `done` dentro de `<Task>`:

```
...
<input type="checkbox" checked={this.props.done ? 'checked' : null} />
...
```

Lo que estamos haciendo es testar si la propiedad `done` es un valor `true` y en el caso de que así sea establecer el atributo HTML `checked` para el elemento `input`. ¿No te parece realmente sencillo? JSX es realmente cómodo y expresivo y cuando asimiles que no estas escribiendo HTML te sentirás como un verdadero *Jedi*.

Propiedades por defecto

React te ofrece una interfaz para poder establecer tanto tipos usando `propTypes` como valores por defecto utilizando `defaultProps`. Éstas propiedades, evidentemente, serán sobrescritas cuando vengan establecidas desde el componente *Padre*. Esto es realmente útil cuando en nuestro componente tenemos propiedades opcionales las cuales queremos que tengan un valor por defecto. Por ejemplo en nuestro componente `<Task>` vamos a establecer la propiedad `name` con ´'Unknown task'´ y `done` con `false`:

task.jsx

```
class Task extends React.Component {

  static propTypes = {
    name: React.PropTypes.string,
    done: React.PropTypes.bool
  };

  static defaultProps = {
    name: 'Unknown task',
    done: false
  };

  ...
};
```

De esta manera si ahora nuestro componente *Padre* `<ToDo>` tuviese las siguientes instancias de `<Task>`:

todo.jsx

```
class ToDo extends React.Component {
  render() {
    return (
      <ul>
        <Task name='Introduction' done />
        <Task name='Chapter 1 - First component' done />
        <Task name='Chapter 2 - Properties' done={false} />
        <Task />
      </ul>
    )
  }
};
```

El resultado final en HTML sería tal que así:

```
<ul>
  <li><input type="checkbox" checked/>Introduction</li>
  <li><input type="checkbox" checked/>Chapter 1 - First component</li>
  <li><input type="checkbox"/>Chapter 2 - Properties</li>
  <li><input type="checkbox"/>Unknown task</li>
</ul>
```

Combinar el uso de `propTypes` y `defaultProps` te permite de forma segura utilizar tus propiedades sin tener que escribir por tu cuenta código repetitivo y normalmente frágil ante el error.

propTypes

El atributo `propTypes` dentro de tus componentes establece un validador de tipo que generará *warnings* en tu consola de desarrollo. Es muy recomendable usar siempre la definición de `propTypes` en todos tus componentes, a pesar de que solo obtendrás un beneficio cuando estés desarrollando tu aplicación es una buena manera de generar consistencia en tus componentes. También te servirá como una vía de documentación implícita, puesto que quien colabore contigo sabrá muy bien qué propiedades espera tu componente y de qué tipo.

A medida que tu aplicación vaya creciendo es útil definir `propTypes` para asegurarse de que tus componentes se utilizan correctamente. `React.PropTypes` exporta una serie de validadores que se pueden utilizar para asegurarse de que los datos que recibes son válidos. Cuando se proporciona un valor no válido, una advertencia se mostrará en la consola JavaScript. Ten en cuenta que por razones de rendimiento propTypes solamente se comprueba en el modo de desarrollo.

Veamos ahora como definir las propiedades de `<Task/>`:

task.jsx

```
class Task extends React.Component {

  static propTypes = {
    name: React.PropTypes.string,
    done: React.PropTypes.bool
  };
  ...
```

En el caso de que necesitemos que alguna de nuestras propiedades sea obligatoria podemos hacer uso del validador `isRequired`:

task.jsx

```
class Task extends React.Component {

  static propTypes = {
    name: React.PropTypes.string.isRequired,
    done: React.PropTypes.bool
  },
  ...
```

En este caso estamos estableciendo como requerida la propiedad `name` por lo tanto podemos eliminarla de la definición `defaultProps` ya que ya no tiene ningún sentido generar un valor por defecto:

task.jsx

```
class Task extends React.Component {

  static propTypes = {
    name: React.PropTypes.string.isRequired,
    done: React.PropTypes.bool
  };

  static defaultProps = {
    done: false
  };
  ...
```

Transferencia *shortcut* de propiedades

Muchas veces te encontrarás con componentes React que extienden un elemento HTML básico y que las propiedades establecidas son un simple *wrapper* de los mismos atributos HTML. En el caso de que nuestro componente tenga muchas propiedades puede ser realmente tedioso el tener que escribir cada una de estas propiedades, para ello te recomiendo que uses la siguiente sintaxis JSX:

image.jsx

```
import React from 'react';

class Image extends React.Component {

  render() {
    return <img {...this.props} />
  }
};

export default Image;
```

En este caso hemos creado un nuevo componente `<Image>` el cual renderizará un simple elemento html `<img>`. Fíjate que hemos utilizado la expresión `...this.props` la cual genera todos los atributos HTML automáticamente. Veamos un sencillo ejemplo usando `<Image>`:

```
<Image src="http://your_image_url.jpg" alt="Amazing Image" />
```

Aunque el ejemplo sea realmente sencillo resume la utilidad de esta técnica. En el próximo capítulo conoceremos como combinar `{...this.props}` con los estados de un componente.

Refs

React te permite establecer una propiedad especial que puedes añadir a cualquiera de tus componentes. Esta propiedad especial te permite crear una referencia, a modo de enlace simbólico, a la instancia generada una vez el componente ha sido renderizado en el DOM. Siempre garantizando que la referencia contenga la instancia apropiada, en cualquier punto del ciclo de vida del componente. Vamos a ver un ejemplo con `<ToDo>` y `<Task>`:

todo.jsx

```
1  import React from 'react';
2  import ReactDOM from 'react-dom';
3  import Task from './task'
4
5  class ToDo extends React.Component {
6
7    componentDidMount() {
8      const el = ReactDOM.findDOMNode(this.refs.first);
9      console.log(el);
10   }
11
12   render() {
13     return (
14       <ul>
15         <Task ref='first' name='Introduction' done />
16         <Task name='Chapter 1 - First component' done />
17         <Task name='Chapter 2 - Properties' />
18         <Task />
19       </ul>
20     )
21   }
22 };
23
24 export default ToDo;
```

En nuestra primera instancia de `<Task>` hemos añadido la propiedad especial `ref`, la cual crea una referencia a esa instancia. Como ves, declaramos el método de ciclo de vida `componentDidMount()`, el cual se ejecuta una vez se ha renderizado nuestro componente, y en el podemos acceder a todas las referencias declaradas utilizando el objeto `this.refs`. En nuestro caso accedemos a `this.refs.first` y mostramos por consola el elemento DOM utilizando el método `findDOMNode` del modulo `react-dom` que dará como resultado:

```
<li data-reactid=".0.0">
  <input type="checkbox" checked/>
  Introduction
</li>
```

Children

Te encontrarás con situaciones donde necesitarás envolver tus componentes en vez de generarlos via propiedades. Por ejemplo ahora nuestro componente `<ToDo>` no va a contener las instancias `<Task>` sino que estas estarán contenidas en un nuevo componente `<Tasks>`:

todo.jsx

```
...
class ToDo extends React.Component {
  ...
  render() {
    return (
      <div>
        <Tasks>
          <Task ref='first' name='Introduction' done />
          <Task name='Chapter 1 - First component' done />
          <Task name='Chapter 2 - Properties' />
          <Task />
        </Tasks>
      </div>
    )
  }
};
...
```

tasks.jsx

```
import React from 'react';

class Tasks extends React.Component {

  render() {
    return (
      <ul>
        { this.props.children }
      </ul>
    )
  }
};

export default Tasks;
```

Como vemos en la nueva definición de nuestro nuevo componente `<Tasks>` utilizamos la propiedad especial `this.props.children` la cual contiene la referencia a los componentes creados dentro de si mismo. Esto puede ser realmente útil cuando queremos crear estructuras algo más complejas como podría ser:

tasks.jsx

```
...
class Tasks extends React.Component {

  render() {
    return (
      <section>
        <header/>
        <ul>
          { this.props.children }
        </ul>
        <footer/>
      </section>
    )
  }
};
...
```

className

Como ya indiqué en el primer capítulo `class` es un *keyword* reservado en JavaScript, y por lo tanto si queremos establecer el atributo HTML `class` necesitarás utilizar la propiedad `className`:

task.jsx

```
...
render() {
  return (
    <li className={this.props.done ? 'done' : null}>{this.props.name}</li>
  );
}
...
```

En este ejemplo hemos cambiado la estructura JSX de nuestro componente `<Task>` en vez de utilizar el elemento HTML `<input/>` vamos a utilizar una clase CSS `.done` la cual estableceremos cuando la propiedad `done` sea `true`. Al comienzo de este capítulo te dije que era muy importante el uso de propiedades y aquí tenemos un ejemplo muy claro; nuestra instancia de `<Task>` se sigue declarando con las propiedades `name` y `done` pero ahora el HTML resultante es diferente.

Por esta simple razón es muy recomendable intentar crear componentes que actúen de `wrappers` de elementos HTML, porque según nuestra aplicación vaya creciendo podremos mejorar estos componentes con nueva sintaxis JSX sin tener que modificar nada más en nuestro código.

Propiedades HTML booleanas

Cuando utilizas propiedades booleanas en tu JSX y necesites establecer un valor `true` no hace falta más que definir la propiedad, como si de HTML se tratará:

todo.jsx

```
...
class ToDo extends React.Component {

  render() {
    return (
      <div>
        <Tasks>
          <Task ref='first' name='Introduction' done />
          <Task name='Chapter 1 - First component' done />
          <Task name='Chapter 2 - Properties' />
          <Task />
        </Tasks>
      </div>
    )
  }
};
...
```

Si recuerdas, nuestro componente `<Task>` tenia la propiedad `done` por defecto a `false`. En la primera instancia de `<Task>` vemos que solo establecemos la propiedad `done` sin hacer uso del valor `true`, esto no hace otra cosa más que facilitar la lectura de nuestros componentes, acostumbra a utilizarlo.

Estados

En el capítulo anterior descubriste como definir propiedades en tus componentes así como establecer los valores desde el componente *Padre*. Como ya sabes las propiedades son inmutables y en el caso de que necesites que tus componentes sean interactivos las propiedades no podrán ayudarte con ello. Como dice Pete Hunt *"compartir estados mutables es el comienzo del mal"*. Por esta razón las propiedades que llegan desde un componente *Padre* son totalmente inmutables, ya que el componente que las recibe no tiene la responsabilidad ni la autoridad para modificarlas.

Con esto, debes aprender que todos los componentes de React contienen un estado mutable y nunca es un estado heredado. Esto significa que solo el componente que contiene el estado tiene la capacidad de modificarlo, ni el *Padre* ni los *Hijos* podrán modificar este estado. Al igual que ocurre con las propiedades un cambio en el estado de un componente, disparará automáticamente una renderización del árbol de componentes que contenga. Por esta razón el estado es muy util para contener datos en el componente, o para utilizarlo como intermediario en el caso de que ese dato tenga que enviarse a los componentes *Hijo* en forma de propiedad. Veamos un ejemplo de definición de estado en nuestro componente `<Task>`:

task.jsx

```
1  import React from 'react';
2
3  class Task extends React.Component {
4
5    static propTypes = {
6      name: React.PropTypes.string.isRequired,
7      done: React.PropTypes.bool
8    };
9
10   static defaultProps = {
11     done: false
12   };
13
14   state = {
15     updated: false
16   };
17
18   render() {
19     return (
20       <li className={this.props.done ? 'done' : null}>
```

```
21          { this.props.name }
22          { this.state.updated ? <small>Updated…</small> : null }
23        </li>
24      )
25    }
26  };
27
28  export default Task;
```

En nuestro componente `<Task>` hemos definido el atributo `state` el cual contiene un objeto con todos los atributos que tiene nuestro estado. En este caso hemos definido el atributo de estado `updated` a `false`. Como vemos en el método `render` accedemos a los atributos del estado mediante el objeto `this.state` seguido del atributo que queremos acceder, en nuestro caso `updated`. Al igual que las propiedades podemos hacer que la estructura JSX sea condicional. Ahora vamos a ver como modificar el estado `updated` a `true`:

task.jsx

```
1   ...
2   class Task extends React.Component {
3     ...
4     handleClick = (event) => {
5       this.setState({ updated: true });
6     };
7
8     render() {
9       return (
10        <li className={this.props.done ? 'done' : null} onClick={this.handleClick}>
11          { this.props.name }
12          { this.state.updated ? <small>Updated…</small> : null }
13        </li>
14      )
15    }
16  };
17  ...
```

Hemos declarado una función `handleClick` que modifica nuestro atributo de estado `updated` a `true` gracias al método `this.setState`. Este método recibe un objeto con todos los atributos de estado que queremos modificar y una vez haya sido ejecutado React volverá a disparar una renderización en la que a nuestro elemento html `<li>` se le habrá añadido un subelemento:

```
<small>Updated…</small>
```

Ahora vamos a tener un pequeño avance de lo que son los eventos *sintéticos*, los cuales profundizaremos en el siguiente capítulo. Como ves el elemento `<li>` tiene un atributo `onClick`, no te asustes no hemos vuelto a los noventa donde se declaraban eventos de esta manera. Recuerda que estamos escribiendo JSX y por lo tanto solo estamos definiendo una estructura para el VirtualDOM, el cual declarará internamente el handler de ese evento para cuando se haga *click* se dispare el *callback* `this.handleClick`.

Utilizando propiedades en el *state*

El propio equipo de React declara esta práctica como peligrosa, rozando el *Anti-Pattern*. Seguro que se te había ocurrido usar una propiedad como *constructor* de algún atributo de estado en tu componente. Por ejemplo algo parecido a esto:

```
...
class Task extends React.Component {
  ...
  state = {
    updated: false,
    name: `Task name is ${this.props.name}`
  };

  render() {
    return (
      <li className={this.props.done ? 'done' : null} onClick={this.handleClick}>
        { this.state.name }
        { this.state.updated ? <small>Updated…</small> : null }
      </li>
    )
  }
};
...
```

Dado que para cuando se ejecute la definición `state` ya tenemos las propiedades de nuestro componente inicializadas podríamos *setear* cualquier atributo de nuestro estado con los valores de las propiedades. En el ejemplo concatenamos el *String* `Task name is` con el valor de la propiedad `this.props.name` en el atributo de estado `name`. Parece que todo esta bien, pero realmente estamos creando una duplicación de valor lo que se llama *Source of Truth* ya que el atributo de estado `name` nunca cambiará. Siempre que sea posible intenta calcular los valores *on-the-fly* para asegurarte que el valor no esta desincronizado, lo cual puede generar problemas. Recuerda que las propiedades de

tus componentes son valores inmutables y por lo tanto en todo el ciclo de vida de tu componente tendrán el mismo valor, a menos que el componente *Padre* establezca un nuevo valor. En cambio los estados son mutables y por lo tanto puede que en nuestro ejemplo el atributo `name` obtenga un nuevo valor y por lo tanto a la hora de ejecutarse el método `render()` no obtengamos lo que esperamos.

La forma recomendada es definir en la función *render* una variable con el valor que queremos. Es sencillo, seguro y fácil de entender:

```
render() {
  const name = `Task name is ${this.props.name}`;
  ...
}
```

De todas formas no siempre esta práctica es un *Anti-Pattern*, puede que en algunas ocasiones sea una buena técnica para inicializar atributos de estado. Veamos el siguiente ejemplo:

count.jsx

```
import React from 'react';

class Count extends React.Component {

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

  handleClick = (event) => {
    this.setState({count: this.state.count + 1});
  };

  render() {
    return <small onClick={this.handleClick}>{this.state.count}</small>
  }
};

export default Count;
```

Como vemos simplemente inicializamos el atributo de estado `count` con la propiedad `total`. Este atributo de estado puede que vaya acumulando nuevos valores, tantas veces como el usuario haga *click* sobre el componente llamaremos a la *callback* `handleClick` la cual modificará el estado con `this.setState`.

Para finalizar con este capítulo solo quiero que te quedes con una pequeña formula para no caer en el *Source of Truth*:

```
var state_attribute = props_attribute != future_state_attribute ? true : false;
```

Eventos Sintéticos

React te ofrece una capa de abstracción a la hora de utilizar la mayoría de eventos de cualquier navegador. Ya hemos visto en el anterior capítulo como utilizar el atributo `onClick`:

link.jsx

```
1  import React from 'react';
2
3  class Link extends React.Component {
4
5    handleClick = (event) => {
6      event.preventDefault();
7      alert('You clicked me!');
8    };
9
10   render() {
11     return (
12       <a href={this.props.url} onClick={this.handleClick}>
13         {this.props.caption}
14       </a>
15     )
16   }
17 };
18
19 export default Link;
```

En este ejemplo estamos capturando cuando el usuario haga *click* en nuestro componente `<Link>`. Al tratarse de una etiqueta `<a>` conoces que el navegador va a disparar la request a la url establecida en el atributo `href`, a menos que capturemos el evento y bloqueemos ese proceso. Para ello en la definición de la función `handleClick` nos llega como parámetro `event` y podremos bloquearlo usando el método `preventDefault`.

Vale, se lo que estás pensando, ¿eventos en linea? ¿hemos vuelto a los '90? o ¿estamos programando como con Angular?. Recuerda que React se diferencia del resto porque tiene una *cosita* llamada VirtualDOM. Todo indica a que es definición es en linea, pero nada más lejos de la realidad, simplemente estamos definiendo el comportamiento en el VirtualDOM. No te preocupes porque React usa por detrás un sistema de delegación de eventos y en la renderización final del componente no verás por ningún lado el atributo `onClick`. Con React no existe confusión alguna entre que elementos tienen eventos asignados y cuales no, y tampoco *ensuciamos* nuestro HTML con asignaciones innecesarias de `id` o `class` para utilizarlos como *hooks* para los eventos.

SyntheticEvent

Todos los eventos en React son instancias de `SyntheticEvent` un *wrapper cross-browser* del evento nativo del navegador. El evento sintético contiene la misma interfaz que el evento nativo del navegador, incluyendo `stopPropagation` y `preventDefault`, a diferencia de que funcionan idénticamente en todos los navegadores. En el caso de que necesites acceder al evento nativo del navegador simplemente tienes que utilizar el atributo `nativeEvent`.

Todos los eventos sintéticos tienen un objeto con los siguientes atributos:

```
boolean         bubbles
boolean         cancelable
DOMEventTarget  currentTarget
boolean         defaultPrevented
number          eventPhase
boolean         isTrusted
DOMEvent        nativeEvent
void            preventDefault()
void            stopPropagation()
DOMEventTarget  target
number          timeStamp
string          type
```

Eventos soportados

Como he dicho React normaliza todos los eventos para que tengan propiedades consistentes en todos los navegadores disponibles. Si te asusta la retrocompatibilidad te adelanto que React funciona desde IE8 en adelante ¿suficiente?.

Los manejadores de eventos serán disparados en la fase de propagación de burbuja, *bubbling phase*). Para registrar un manejador de evento en la fase de propagación de captura, *capture phase*, añade `Capture` al nombre del evento; por ejemplo en vez de usar `onClick` deberías utilizar `onClickCapture`. Ahora vamos a descubrir todos los eventos que React puede capturar de manera homogénea:

Eventos de Portapapeles

```
onCopy onCut onPaste
```

Las propiedades que vendrán en este tipo de eventos son: `DOMDataTransfer clipboardData`

Eventos de teclado

```
onKeyDown onKeyPress onKeyUp
```

Las propiedades que vendrán en este tipo de eventos son:

```
boolean altKey
Number charCode
boolean ctrlKey
function getModifierState(key)
String key
Number keyCode
String locale
Number location
boolean metaKey
boolean repeat
boolean shiftKey
Number which
```

Eventos de foco

```
onFocus onBlur
```

Las propiedades que vendrán en este tipo de eventos son:

```
DOMEventTarget relatedTarget
```

Eventos de formulario

Las etiquetas html utilizadas dentro un `<form>` como pueden ser `<input>`, `<textarea>` y `<option>` son diferentes del resto de etiquetas puesto que pueden ser mutadas por la interacción del usuario. React ofrece para este tipo de componentes una interfaz que te hará la vida más fácil a la hora de manejar formularios. Por ejemplo si nos suscribimos a `onChange` este trabajará igual en todos los navegadores y disparará este evento cuando:

- La propiedad `value` de un `<input>` o `<textarea>` cambie.
- La propiedad `checked` de un `<input>` cambie.
- La propiedad `selected` de un `<option>` cambie.

```
onChange onInput onSubmit
```

Eventos de ratón

```
onClick onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit
onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave
onMouseMove onMouseOut onMouseOver onMouseUp
```

Los eventos `onMouseEnter` y `onMouseLeave` se propagan desde el componente que se está dejando y hacia el que se está entrando y por lo tanto a diferencia del resto de eventos no tiene fase de captura. Las propiedades que vendrán en este tipo de eventos son:

```
boolean altKey
Number button
Number buttons
Number clientX
Number clientY
boolean ctrlKey
function getModifierState(key)
boolean metaKey
Number pageX
Number pageY
DOMEventTarget relatedTarget
Number screenX
Number screenY
boolean shiftKey
```

Eventos táctiles

Para habilitar los eventos táctiles dentro de tu aplicación React tendrás que utilizar `React.initializeTouchEvents(t` antes de renderizar cualquier componente.

```
onTouchCancel onTouchEnd onTouchMove onTouchStart
```

Las propiedades que vendrán en este tipo de eventos son:

```
boolean altKey
DOMTouchList changedTouches
boolean ctrlKey
function getModifierState(key)
boolean metaKey
boolean shiftKey
DOMTouchList targetTouches
DOMTouchList touches
```

Evento de Scroll

```
onScroll
```

Las propiedades que vendrán en este tipo de eventos son:

```
Number detail
DOMAbstractView view
```

Eventos de rueda

```
onWheel
```

Las propiedades que vendrán en este tipo de eventos son:

```
Number deltaMode
Number deltaX
Number deltaY
Number deltaZ
```

Autobinding y Delegación de eventos

Cuando definimos *callbacks* en JavaScript, normalmente tenemos que especificar el método de la instancia donde la queremos utilizar. Esto puede ser un foco de errores porque puede que en el momento de la asignación no exista ni tan siquiera la instancia. Con React todos los métodos están automáticamente bindeados a la instancia del componente donde son definidos. Estos métodos son cacheados consiguiendo menos proceso CPU y una gestión eficaz de la memoria.

Al contrario de lo que podrías pensar React no genera manejadores de evento por cada nodo de tus componentes. Cuando React arranca, comienza a escuchar todos los eventos en el nivel superior con un simple *listener* y cuando un componente se crea, o destruye, todos los manejadores definidos en el simplemente se añaden, o eliminan, dentro del mapa del *listener*. Esta solución es realmente rápida y al igual que el *Autobinding* es responsable con la memoria consumida por tu aplicación.

Ciclo de vida

Todos los componentes de React tienen un ciclo de vida realmente complicado, la propia naturaleza de React hace que este sea de esta manera. Para facilitarnos la vida React nos provee de una serie de eventos para capturar y actuar en determinados puntos de este ciclo de vida. Realmente son solo 7 puntos de captura, los cuales podemos dividir por puntos de construcción, actualización y destrucción.

A continuación podrás descubrir cuales son y para utilizarlos solo tendrás que incluirlos dentro de la definición de tu componente.

Construcción: componentWillMount

```
1 componentWillMount() {}
```

Este método se ejecutará solo una vez, tanto si estás utilizando React en el Cliente como en el Servidor, y lo hará una vez el proceso de renderización haya concluido. Ten en cuenta que si utilizas `setState` dentro de este método, el proceso `render()` se ejecutará solo una vez a pesar de haber disparado un cambio de estado.

Construcción: componentDidMount

```
1 componentDidMount() {}
```

Al igual que `componentWillMount` se ejecuta una sola vez, pero esta vez solo en aplicaciones Cliente. En este punto del ciclo de vida tu componente, este ya tiene una representación en el DOM a la cual tienes acceso mediante la función `ReactDOM.findDOMNode(this)`. Algo muy importante que tienes que tener en cuenta es que si tienes que:

- Integrarte con algún *Framework* JavaScript.
- Establecer algún *timer* con `setTimeout` o `setInterval`.
- Realizar alguna petición AJAX de precarga de datos.

Este es el método donde tendrás que incluir este tipo de operaciones.

Actualización: componentWillReceiveProps

```
componentWillReceiveProps(next_props) {}
```

Es un punto de captura de actualización y como su nombre indica este método se ejecutará cuando nuestro componente reciba nuevas propiedades. Al ser un punto de captura de actualización no se ejecutará con la primera renderización.

El método `componentWillReceiveProps` recibe como parámetro las futuras propiedades que va a contener nuestro componente y por lo tanto podemos hacer una comparación con las propiedades actuales utilizando `this.props`:

```
componentWillReceiveProps(next_props) {
  this.setState({
    changed: this.props.id !== next_props.id ? true : false
  });
}
```

En este ejemplo estamos comparando si el atributo `id` de nuestras propiedades cambia con la nueva asignación, conteniendo la booleana en un atributo de estado llamado `changed`. El uso de `setState()` dentro de este método nunca dispara una renderización adicional ya que React agrupa todos los cambios sean de propiedades o estado dentro de este método.

Usa este punto de captura como una oportunidad para generar una transición antes de que nuestro componente vuelva a ejecutar `render()`. Aquí tienes un ejemplo utilizando un cambio de atributo de estado:

```
componentWillReceiveProps(next_props) {
  this.setState({
    loading: true
  });
},

render() {
  className = this.state.loading ? 'loading' : 'loaded';
  return <div className={className}></div>
}
```

No existe un método tal que `componentWillReceiveState` para cuando un atributo de estado ha sido modificado. Un cambio de propiedad puede generar un cambio de estado, pero nunca ocurrirá lo contrario. Si necesitas realizar operaciones en respuesta a un cambio de estado tendrás que utilizar el punto de captura `componentWillUpdate`.

Actualización: componentWillUpdate

```
componentWillUpdate(next_props, next_state) {}
```

Este punto de captura de actualización se ejecutará inmediatamente antes de renderizar las nuevas propiedades o atributos de estado. Y al igual que `componentWillReceiveProps` nunca se ejecutará con la primera renderización. En esta función recibiremos dos parámetros:

- Las nuevas propiedades
- El nuevo estado

Al igual que `componentWillReceiveProps` este es un buen lugar para definir una transición antes de que se actualice nuestro componente. Lo que si tienes que tener en cuenta es que no puedes utilizar `setState()` dentro de este método, puesto que generarías una referencia cíclica. Si necesitas actualizar un atributo de estado en respuesta a un cambio de propiedad deberás utilizar forzosamente `componentWillReceiveProps`.

Actualización: shouldComponentUpdate

```
shouldComponentUpdate(next_props, next_state) {
  return true;
}
```

Este punto de captura de actualización, al igual que el anterior, se ejecuta antes de renderizar las nuevas propiedades o estado, y nunca se ejecutará con la renderización inicial.

A diferencia del resto de puntos de captura de actualización puedes utilizar este método para bloquear la renderización retornando un *Boolean* con `false`. Esto es realmente util cuando estás seguro de que los cambios en las nuevas propiedades o en el estado no necesitan ser renderizados. Resumiendo, si `shouldComponentUpdate` devuelve `false`, entonces `render()` será totalmente omitido hasta que exista un nuevo cambio de propiedades o estado y por lo tanto `componentWillUpdate` y `componentDidUpdate` no serán ejecutados.

Por defecto este método siempre devolverá `true` para prevenir errores sutiles cuando el estado ha sido modificado dentro de él. Pero si dentro de este método tratas los atributos de estado como inmutables puedes sobreescribir el retorno con lo que realmente necesites.

Todos sabemos que el rendimiento es muy importante en las aplicaciones webs y a veces puede convertirse en un verdadero cuello de botella. Imagina que tenemos cientos de componentes dentro de nuestra aplicación, el uso responsable de `shouldComponentUpdate` puede acelerar de manera considerable tu aplicación.

Actualización: componentDidUpdate

```
componentDidUpdate(prev_props, prev_state) {}
```

Este punto de captura de actualización se ejecuta inmediatamente después de que el componente haya sido renderizado. Al igual que todos los puntos de captura de actualización nunca se ejecutará con la primera renderización. En este caso recibimos dos parámetros:

- **`prev_props`**: Las propiedades anteriores
- **`prev_state`**: El estado anterior

Este es un buen lugar para operar con el DOM de nuestro componente, porque en este punto tenemos todos los cambios representados con la nueva renderización.

Destrucción: componentWillUnmount

```
componentWillUnmount() {}
```

Es el único punto de captura de destrucción que tenemos dentro del ciclo de vida de nuestro componente, y se ejecutará inmediatamente antes de que el componente se desmonte del DOM. Puede ser utilizado para realizar algún tipo de *limpieza*, como puede ser eliminar *timers* así como eliminar elementos del DOM que fueran creados dentro de `componentDidMount`.

Hijos dinámicos

Poco a poco vas adquiriendo el *Code Style* con React, ahora vas a conocer la enorme flexibilidad que esta librería te ofrece con la creación de componentes dinámicos. Ya conoces que todos los componentes de React pueden ser contenedores de otros componentes, componente *Padre*, o contenidos, componente *Hijo*. La asignación de este tipo de estructuras puede ser estática, cuando conocemos los valores exactos a representar, o dinámica; cuando la fuente de datos es incierta. Vamos a echar un vistazo a este ejemplo:

heroes.jsx

```
import React from 'react';

class Heroes extends React.Component {

  state = {
    list: []
  };

  render() {
    if (!this.state.list.length) {
      return ( <div>No Heroes!</div> );
    }

    return (
      <ul>
      {
        this.state.list.map(function(heroe, index) {
          return (
            <li key={index}>
              {heroe.name} the { heroe.power }!
            </li>
          );
        })
      }
      </ul>
    );
  }
};
```

```
29
30 export default Heroes;
```

Como puedes leer nuestro nuevo componente `<Heroes>` utiliza la función `render` comprobando si el atributo de estado `list` tiene contenido. Al estar vacío, ya que lo inicializamos en `state`, nuestro componente renderizará una etiqueta `<div>` con el contenido `No Heroes!`. En caso de que este atributo conteniese una lista de heroes devolvería una etiqueta `<ul>` y por cada héroe crearía una etiqueta `<li>`.

Ahora vamos a hacer que tras la primera renderización de nuestro componente, el atributo de estado `list` contenga una lista de heroes. Ahora que ya conoces el ciclo de vida de los componentes React, vamos a utilizar el método `componentDidMount`:

heroes.jsx

```
1  import React from 'react';
2
3  var dataSource = [
4    { name: 'Superman' , power: 'fly with underwear' },
5    { name: 'Batman'   , power: 'belt with gadgets' },
6    { name: 'Spiderman', power: 'Jump like a monkey' },
7    { name: 'Hulk'     , power: 'Angry with anyone' }
8  ];
9
10 class Heroes extends React.Component {
11
12   componentDidMount() {
13     setTimeout(function() {
14       this.setState({ list: dataSource });
15     }.bind(this), 2000);
16   }
17   ...
18 }
19 ...
```

Con este simple cambio hacemos que nuestro componente `<Heroes>` se comporte de manera dinámica. Ya que una vez finalice la primera renderización, se ejecutará el método `componentDidMount` y este ejecutará un cambio del atributo de estado `list` cuando transcurran 2 segundos. Ahora vamos a pararnos en este punto y vamos a analizar la sintaxis JSX que renderizamos por cada héroe:

```
<li key={index}>
  {heroe.name} the { heroe.power }!
</li>
```

Como ves dentro de la etiqueta `<li>` creamos una interpolación con el nombre del héroe y su superpoder, siendo en el caso de *Batman* algo como `Batman the belt with gadgets!`. Después de este chiste malo, muy malo, quiero que te fijes en el atributo `key` que defino en la etiqueta. React recomienda (casi obliga) el uso de este atributo cuando tengas estructuras repetitivas, donde sus hijos comparten estructura, ayudando a gestionar más eficientemente la reordenación y eliminación de cada hijo. React utiliza un algoritmo de reconciliación el cual optimiza el numero de renderizaciones de cada componente. Es importante que tengas claro que el atributo `key` tiene que asignarse siempre en la estructura superior de nuestra sintaxis JSX.

En nuestro caso el atributo `key` lo rellenamos con el `index` del array, pero puedes utilizar un `id` o cualquier valor que distinga a cada hijo del componente. En el caso de que no asignes este atributo aparecerá un *warning* en tu consola *JavaScript* por lo que intenta hacerlo siempre que puedas, además siempre podrás utilizar el `index` de tu grupo de datos.

Ahora vamos a finalizar nuestro ejercicio creando un poco más de dinamismo en nuestro componente `Heroes`:

heroes.jsx

```
import React from 'react';

var dataSource = [
  { name: 'Superman' , power: 'fly with underwear' },
  { name: 'Batman'   , power: 'belt with gadgets' },
  { name: 'Spiderman', power: 'Jump like a monkey' },
  { name: 'Hulk'     , power: 'Angry with anyone' }
];

class Heroes extends React.Component {

  state = {
    list: []
  };

  fetchData = () => {
    setTimeout(function() {
      this.setState({ list: dataSource });
    }.bind(this), 2000);
  };

```

```
22    handleReset = () => {
23      this.setState({ list: [] });
24    };
25
26    handleFetch = () => {
27      this.fetchData();
28    };
29
30    componentDidMount() {
31      this.fetchData();
32    }
33
34    render() {
35      if (!this.state.list.length) {
36        return (
37          <div>
38            No list!
39            <br />
40            <button onClick={this.handleFetch}>Fetch</button>
41          </div>
42        );
43      } else {
44        return (
45          <div>
46            <ul>
47            {
48              this.state.list.map(function(heroe, index) {
49                return (
50                  <li key={index}>{heroe.name} the {heroe.power}!</li>
51                );
52              })
53            }
54            </ul>
55            <button onClick={this.handleReset}>Reset</button>
56          </div>
57        );
58      }
59    }
60  }
61
62  export default Heroes;
```

Como ves ahora tenemos dos `<button>` a modo de accionadores de contenido, los cuales capturare-

mos cuando el usuario haga *click* sobre ellos:

- Cuando ejecutemos el evento `handleReset` eliminará el contenido del atributo de estado `list`.
- Cuando pulsemos sobre `handleFetch` llamaremos a una nueva función interna `fetchData` la cual asigna el contenido de `dataSource` al atributo `list`

Date cuenta que hemos refactorizado nuestro código y ahora nuestro método de ciclo de vida `componentDidMount` llama a la función interna `fetchData`.

Con esto hemos finalizado este escueto pero intenso capítulo, retómalo todas las veces que te haga falta porque esta es la base del mayor numero de aplicaciones web actuales, renderizar grupo de datos. Recuerda usar siempre el atributo `key` y aprovéchate del ciclo de vida de tus componentes para ofrecer un mayor dinamismo a la hora de representar tus datos.

Anidamiento de vistas

Gracias a que React es una librería orientada a componentes es muy fácil crear estructuras anidadas como las que harías con *HTML*. La gran diferencia con este lenguaje es que en vez de anidar etiquetas, estamos anidando componentes los cuales tienen una funcionalidad ya definida. Grábalo a fuego como lo hiciste con *JavaScript* y su *"todo es un objeto"*... en React *"todo es un componente"*.

Vamos a recuperar nuestro componente `<ToDo>` y ahora lo reconvertiremos en un componente algo más completo que contenga otros componentes:

todo.jsx

```
...
render() {
  return (
    <div>
      <Header />
      <Content>
        <Task name="Introduction" done />
        <Task name="Chapter 1 - First component" done />
        <Task name="Chapter 2 - Properties" />
      </Content>
      <footer>Copyright...</footer>
    </div>
  )
}
...
```

La definición de los nuevos componentes `<Header>` y `<Content>` será la siguiente:

header.jsx

```
import React from 'react';

class Header extends React.Component {

  render() {
    return (
      <header>
        <a href="#pending">pending</a>
        <a href="#done">done</a>
        <a href="#all">all</a>
      </header>
    )
  }
};

export default Header;
```

content.jsx

```
import React from 'react';

class Content extends React.Component {

  render() {
    return (
      <section>
        <ul>
          {this.props.children}
        </ul>
        <span>Tasks: {this.props.children.length}</span>
      </section>
    )
  }
};

export default Content;
```

Para poder utilizar los componentes `<Header>` y `<Content>` tendremos que importarlos en el fichero donde estamos creando el componente `<ToDo>`:

todo.jsx:

```
import Header from './header'
import Content from './content'
```

React tiene una convención que tienes que respetar en todo momento, utiliza *BumpyCase* para los nombres de tus componentes. De esta manera podrás diferenciarlos de las simples etiquetas *HTML*. Ahora si nos fijamos en el componente `<ToDo>` vemos que dentro de su componente *Hijo* `<Content>` instanciamos varias veces el componente `<Task>` del primer capítulo. En este caso lo importante es ver la estructura JSX de `<Content>` ya que utilizando la propiedad `this.props.children` podemos definir donde se renderizarán todos sus componentes *Hijo*:

```
<section>
  <ul>
    {this.props.children}
  </ul>
  <span>Tasks: {this.props.children.length}</span>
</section>
```

Además como puedes ver creamos una etiqueta `<span>` que contendrá el numero de hijos que contenga el componente. Por lo que para este ejemplo la estructura generada tras renderizarse todo el componente `<ToDo>` sería tal que así:

```
<div>
  <header>
    <a href="#pending">pending</a>
    <a href="#done">done</a>
    <a href="#all">all</a>
  </header>
  <section>
    <ul>
      <li><input type="checkbox" checked/>Introduction</li>
      <li><input type="checkbox" checked/>Chapter 1 - First component</li>
      <li><input type="checkbox"/>‘Chapter 2 - Properties</li>
    </ul>
    <span>Tasks: 3</span>
  </section>
  <footer>Copyright...</footer>
</div>
```

Como ves es realmente sencillo crear estructuras complejas de componentes React. Ten en mente siempre la reutilización de componentes definiendo una correcta estructura *HTML* y en el caso de que sea necesario estableciendo la posición concreta donde se renderizarán los componentes *Hijo*.

Extendiendo componentes con *Mixins*

Ya sabes como reutilizar componentes en diferentes areas de tu *App*, o incluso poder compartirlos entre diferentes aplicaciones. El equipo de React ha pensado en todo y también ofrece una interfaz sencilla para cuando necesitas compartir funcionalidad entre diferentes tipos de componentes.

Los componentes son la mejor manera de reutilizar tu código en React, pero en algunas ocasiones componentes concebidos para realizar tareas totalmente diferentes pueden compartir algún tipo de funcionalidad en común. A esto normalmente se llama el problema *cross-cuttings* y para ello React te proporciona los *Mixins* para resolver este problema.

De todas formas tómate este capítulo como una funcionalidad especial, ya que actualmente la comunidad está debatiendo mucho sobre los *Mixins*. Los podrás seguir utilizando mientras sigas utilizando el método `React.createClass` puesto que en *EcmaScript 6* no estarán disponibles. Parte de la comunidad parece estar inclinándose hacia el uso de *Componentes de Composición* o *Decoradores* (funciones o componentes que pueden manipular o alterar otros componentes).

Mixins

Debemos tomar los *Mixins* como una muy buena estrategia para crear extensiones del ciclo de vida de nuestros componentes. Esto quiere decir que podemos hacer que un determinado componente ademas de tu propia definición en el ciclo de vida, pueda ejecutar los ciclos de vida de los *Mixins*. Veamos un ejemplo para enterarnos mejor:

hello.jsx

```
1  import React from 'react';
2
3  import MixinES from './mixin.es'
4  import MixinFR from './mixin.fr'
5
6  const Hello = React.createClass({
7
8    mixins: [MixinES, MixinFR],
9
10   componentDidMount: function() {
11     console.log('Hello World!');
12   },
13   ...
14 });
```

```
15
16 export default Hello;
```

mixin.es.js

```
1  const mixinES = {
2    componentDidMount: function() {
3      console.log('Hola Mundo!');
4    }
5  };
6
7  export default mixinES;
```

mixin.fr.js

```
1  const mixinFR = {
2    componentDidMount: function() {
3      console.log('Bonjour le Mounde!');
4    }
5  };
6
7  export default mixinFR;
```

En nuestro componente `<Hello>` definimos un atributo `mixins` con un *Array* de Mixins que queremos utilizar, en este caso queremos utilizar `MixinES` y `MixinFR`. Tras la ejecución del punto de captura del ciclo de vida `componentDidMount` la consola *JavaScript* mostrará:

```
1 Hola Mundo!
2 Bonjour le Mounde!
3 Hello World!
```

Una característica de los *Mixins* es que si un determinado componente utiliza múltiples *Mixins* estos se ejecutarán en el mismo ciclo de vida. Esto asegura que todos ellos se ejecuten en el orden definido y al finalizar todos ellos llamarán al *callback* principal del componente.

EcmaScript6 y *react-mixin*

Todos los métodos que definas en tus clases *EcmaScript 6* no tendrán un *binding* automático a la instancia como cuando utilizamos `React.createClass`. Esto quiere decir que tendrás que hacer uso de `bind(this)` o del llamado *FatArrow* `=>`. Es por esta simple razon por lo cual no podrás definir ningún *Mixin* ya que *EcmaScript 6* no los soporta de manera nativa.

En el caso de que quieras utilizar *mixins* te recomiendo utilizar el modulo `react-mixin` que te permite extender tus clases EcmaScript 6 con mixins orientados para `React.createClass`. Primero vamos a instalar este modulo en nuestro `package.json`:

```
npm install --save-dev react-mixin
```

Ahora vamos a modificar nuestro componente `<Hello>` para convertirlo en una clase EcmaScript 6 que utilice los mixins `MixinES` y `MixinES`:

```
import React from 'react';
import reactMixin from 'react-mixin';

import MixinES from './mixin.es.js'
import MixinFR from './mixin.fr.js'

class Hello extends React.Component {

  componentDidMount() {
    console.log('Hello World!')
  }
  ...
};

reactMixin(Hello.prototype, MixinES);
reactMixin(Hello.prototype, MixinFR);

export default Hello;
```

Si comparamos la definición de `<Hello>` con la anterior version, el único cambio que existe es el uso de `react-mixin` (ademas de la propia sintaxis EcmaScript 6). Como ves su uso es bastante sencillo:

```
reactMixin(Hello.prototype, MixinES);
reactMixin(Hello.prototype, MixinFR);
```

Enviamos al metodo `reactMixin` la *prototipica* de nuestra clase `<Hello>` junto con el *mixin* que necesitemos. La única diferencia es que si nos fijamos en la consola vemos que el orden ha mutado esto es asi por la propia naturaleza de `react-mixin` pero basta con que cambies el orden de la asignación:

```
Bonjour le Mounde!
Hola Mundo!
Hello World!
```

Higher-Order Components

Como te he introducido al principio de este capítulo, a veces los *Mixins* son realmente frágiles y no los podrás utilizar de manera nativa con *EcmaScript 6*. Veamos unas cuantas razones:

- **El contrato entre el componente y el mixin es implicito**. A veces los *Mixins* se basan en métodos definidos en el propio componente, y no hay manera de verlo desde el propio componente.
- **Cuando utilizas Mixins con el mismo nombre en un solo componente**. Por ejemplo, si utilizas algo como `StoreMixin(SomeStore)` y `StoreMixin(OtherStore)`, React disparará una excepción ya que dispones de dos versiones del *mismo* método.
- **Los Mixins tienden a agregar más atributos de estado a tu componente** cuando en realidad lo que buscamos es tener el menor número de atributos. Esto es así porque a mayor numero de atributos de estado mayores son las posibilidades de llevar a cabo un nuevo `render` y eso está totalmente relacionado con el rendimiento de nuestra *Hello*.
- **Los Mixins complican la optimización de rendimiento**. Si defines el método `shouldComponentUpdate` en tus componentes, significa que puedes llegar a tener problemas si alguno de los *Mixins* necesita su propia implementación de `shouldComponentUpdate` puesto que puede forzar una renderización cuando realmente no es así.

Por esta razón la comunidad ha definido un patrón denominado *Higher-order Component* el cual es solo una función que recibe un componente existente y devuelve un nuevo componente que encapsula al primero. Para entenderlo mejor vamos a crear una función `connectToStores`:

connect_to_stores.js

```
1  function connectToStores(Component, stores, getStateFromStores) {
2
3    const StoreConnection = React.createClass({
4      getInitialState() {
5        return getStateFromStores(this.props);
6      },
7
8      componentDidMount() {
9        stores.forEach(store =>
10         store.addChangeListener(this.handleStoresChanged)
11       );
12     },
13
14     componentWillUnmount() {
15       stores.forEach(store =>
16         store.removeChangeListener(this.handleStoresChanged)
```

```
      );
    },

    handleStoresChanged() {
      if (this.isMounted()) {
        this.setState(getStateFromStores(this.props));
      }
    },

    render() {
      return (<Component {...this.props} {...this.state} />);
    }
  });

  return StoreConnection;
};
```

Nuestra función recibe 3 parámetros:

- **component**, es el componente que queremos *extender*.
- **store**, el grupo de datos
- **getStateFromStores**, una función de callback

Se parece bastante a un *Mixin* pero en vez de gestionar el estado internamente, lo que hace es encapsular el componente en uno nuevo y establecer algunas propiedades más en él. De esta manera el ciclo de vida de nuestro componente funciona sin ningún tipo de fusión de eventos. Ahora vamos a crear un nuevo componente `ProfilePage` el cual utilizaremos para extender con la funcionalidad de `connectToStores`:

profile_page.jsx

```
var ProfilePage = React.createClass({

  propTypes: {
    userId: PropTypes.number.isRequired,
    user: PropTypes.object
  },

  render() {
    var { user } = this.props;
    return (<div>{user ? user.name : 'Loading'}</div>);
  }
});
```

Ahora extendemos `ProfilePage` simplemente llamando a `connectToStores` y enviándose a si mismo, así como la *Store* y el callback:

```
1 ProfilePage = connectToStores(ProfilePage, [UserStore], props => ({
2   user: UserStore.get(props.userId)
3 });
```

Como ves esta técnica es igual de efectiva y totalmente orientada a proyectos que utilicen `Ecmascript 6`. Mi recomendación es que a la hora de elegir una tecnología o arquitectura a la hora de diseñar una App no te dejes llevar por el *hype* o los *trending-frameworks*. Intenta ser sensato y objetivo en tu decisión por lo que si te sientes más cómodo con los *Mixins* que con los *Higher-Order Components* y no tienes la necesidad imperiosa de trabajar con EcmaScript 6, no lo dudes y utilízalos.

Renderizador Puro

Ahora que conoces el ciclo de vida de tus componentes React sabes que el rendimiento de tu *App* puede estar influenciado por el número de modificaciones de tus propiedades o atributos de estado. Esto hace que la función `shouldComponentUpdate` reciba un papel muy importante, ya que si siempre devolvemos `true` puede que tengamos serios problemas con los usuarios de tu aplicación.

Una manera muy sencilla de conseguir un considerable aumento del rendimiento es a través del *Mixin* `PureRenderMixin`. Si la función de tus componentes es *pura*, en otras palabras que el render siempre devolverá las mismas propiedades y atributos de estado, puedes usar este *Mixin* para obtener una mejora en el rendimiento. Internamente, lo que hace este *Mixin* es implementar el método `shouldComponentUpdate` y comparar automáticamente los valores actuales, tanto de propiedades como estado, con los nuevos valores y retornar `false` si son iguales.

Una cosa que tienes que tener en cuenta es que la comparación de valores solo se realiza de manera superficial. Si estos valores contienen estructuras de datos complejas, puede devolver `false` cuando en realidad no es así. Además, `shouldComponentUpdate` evita renderizaciones de todo el árbol de subcomponentes, por lo que todos los componentes *Hijo* deberán ser puros también. Utilizar el *Mixin* `PureRenderMixin` es realmente sencillo, simplemente tenemos que instalarlo en nuestro proyecto:

```
npm install --save-dev react-addons-pure-render-mixin
```

Una vez instalado lo importamos y utilizamos como cualquier otro *mixin*:

hello.jsx

```
import React from 'react';
import PureRenderMixin from 'react-addons-pure-render-mixin';

class Hello extends React.Component {

  render() {
    return <div className={this.props.className}>foo</div>
  }
};

reactMixin(Hello.prototype, PureRenderMixin);

export default Hello;
```

Esta es solo la asignación del *Mixin* a nuestro componente, pero no sabemos si realmente está funcionando, veamos un ejemplo completo:

```
...
class Hello extends React.Component {

  state = {
    message: 'Hi!'
  },

  componentDidUpdate() {
    console.log('Component updated!');
  },

  handleClick = (value, event) => {
    console.log('click event', value);
    this.setState({ message: value });
  },

  render() {
    console.log('Component rendered!');
    return (
      <div>
        <h1>Greeting: {this.state.message}</h1>
        <button onClick={this.handleClick.bind(null, 'Hi!')}>Hello</button>
        <button onClick={this.handleClick.bind(null, 'Bye!')}>Goodbye</button>
      </div>
    );
  }
};
...
});
```

El atributo de estado inicial `message` contiene el valor `"Hi!"`, y si pulsamos en el botón `Hello` nuestra consola *JavaScript* mostrará:

```
"click event"
"Hi!"
```

Como puedes comprobar no se ha ejecutado el método `render` puesto que el nuevo estado es el mismo que el definido al construir el componente. Ahora vamos a pulsar sobre el botón `Goodbye` y en este caso nuestra consola *JavaScript* devolverá un resultado diferente:

```
"click event"
"Bye!"
"Component rendered!"
"Component updated!"
```

Puesto que hemos modificado nuestro atributo de estado `message` al nuevo valor `Goodbye` nuestro *Mixin* `PureRenderMixin` decide que es momento de volver a renderizar nuestro componente. Mi recomendación es que a menos que necesites gestionar por tu cuenta `shouldComponentUpdate` intentes implementar siempre `PureRenderMixin` y dejar que React decida cuando volver a renderizar el componente.

Encapsulando librerías

Una de las grandes características de React es que no tienes porque utilizar esta librería en toda tu *App*, recuerda *solo es la V*. Puedes comenzar utilizando React en pequeñas partes de tu App... hasta que al final termines por escribir todo tu código con React.

En este capítulo vas a aprender a utilizar librerías de terceros dentro de React, librerías que no fueron desarrolladas para funcionar con React (tal vez porque incluso React no existía cuando se crearon). Vamos a aprender como incluir MomentJS[18], una de las más famosas librerías para el tratamientos de fechas con *JavaScript*.

Vamos a crear un nuevo componente `<MomentWrapper>` el cual encapsulará la librería MomentJS y la cual podremos utilizar por medio del uso de propiedades. Comencemos creando la estructura básica:

moment_wrapper.jsx

```
1  import React from 'react';
2  import moment from 'moment';
3
4  class MomentWrapper extends React.Component {
5
6    render() {
7      return <span></span>
8    }
9  };
10
11 export default MomentWrapper;
```

Como ves nuestro componente `<MomentWrapper>` no hace nada especial únicamente requerimos la librería `moment`. Ahora vamos a definir las primeras propiedades y utilizar por primera vez la librería:

moment_wrapper.jsx

[18] http://momentjs.com

```
...
class MomentWrapper extends React.Component {

  static propTypes = {
    date: React.PropTypes.any
  };

  static defaultProps = {
    date: new Date()
  };

  render() {
    const date = moment(this.props.date).format('MMMM Do YYYY');
    return <span>{date}</span>
  }
};
...
```

Hemos definido una propiedad `date` la cual es de tipo `React.PropTypes.any` lo cual nos permitirá recibir cualquier tipo de dato. Por defecto nuestra propiedad `date` contendrá la fecha actual. Ahora si echas un vistazo a la función `render` ves que utilizamos por primera vez la librería MomentJS realizando un cambio de formato:

```
const date = moment(this.props.date).format('MMMM Do YYYY');
```

Ahora si quisiéramos utilizar nuestro componente `<MomentWrapper>` podríamos hacerlo de la siguiente manera:

```
<MomentWrapper />
<MomentWrapper date={new Date("04/10/1980")} />
```

Lo que daría como resultado en nuestra consola *JavaScript*:

```
"January 7th 2016" /* en el momento de la ejecución ;) */
"April 10th 1980"
```

Ahora vamos a definir una nueva propiedad `format` la cual nos de total libertad a elegir el formato de fecha que necesitamos utilizar:

moment_wrapper.jsx

```
...
class MomentWrapper extends React.Component {

  static propTypes = {
    date: React.PropTypes.any,
    format: React.PropTypes.string
  };

  static defaultProps = {
    date: new Date(),
    format: 'MMMM Do YYYY, h:mm:ss'
  };

  render() {
    const date = moment(this.props.date).format(this.props.format);
    return <span>{date}</span>
  }
};
...
```

Nuestra nueva propiedad `format`, en el caso de que no definamos ningún valor, contendrá el formato por defecto `'MMMM Do YYYY, h:mm:ss'`. Si ahora volvemos a usar nuestro componente:

```
<MomentWrapper format="YYYY/MM/DD"/>
<MomentWrapper date={new Date("04/10/1980")} format="MMMM" />
```

El resultado en nuestra consola *JavaScript* usando las propiedades `date` y `format` sería tal que así:

```
"January 7th 2016, 9:27:44" /* en el momento de la ejecucion ;) */
"April 10th 1980, 12:00:00"
```

Este es un ejemplo muy sencillo de encapsulamiento de librerías *JavaScript* dentro de componentes React, pero hemos conseguido generar una funcionalidad transparente para el resto de componentes. Esto es, si en el futuro decides no usar MomentJS y prefieres utilizar otra librería simplemente tendrás que respetar la interfaz de propiedades generada.

Comunicación entre componentes

Como hemos aprendido React nos ofrece un sistema de comunicación de datos entre componentes por medio de propiedades. Esto es, el componente *Padre* siempre podrá establecer nuevas propiedades y comportamientos pero el componente *Hijo* no podrá hacer lo mismo. Seguro que te estarás preguntando como podríamos modificar cualquier atributo de estado de nuestro *Padre*. La solución es bastante sencilla es estableciendo un *callback* en el padre que será ejecutado por el *hijo*. Veamos un ejemplo bien sencillo:

cart.jsx

```
1  import React from 'react';
2  import Product from './product'
3
4  class Cart extends React.Component {
5
6    handleClick = (id, event) => {
7      console.log(id, event);
8    };
9
10   render() {
11     return (
12       <ul>
13         <Product id={1} name="Macbook Air 11" onClick={this.handleClick} />
14         <Product id={2} name="Macbook 12'" onClick={this.handleClick} />
15         <Product id={3} name="Macbook Air 13" onClick={this.handleClick} />
16       </ul>
17     );
18   }
19 };
20
21 export default Cart;
```

product.jsx

```
import React from 'react';

class Product extends React.Component {

  static propTypes = {
    id: React.PropTypes.number.isRequired,
    name: React.PropTypes.string.isRequired,
    onClick: React.PropTypes.func.isRequired
  };

  render() {
    return (
      <li onClick={this.props.onClick.bind(null, this.props.id)}>
        {this.props.name}
      </li>
    );
  }
};

export default Product;
```

Como vemos tenemos dos componentes: `<Cart>` y `<Product>`, siendo el primero de ellos el contenedor *Padre*. Si vemos la definición de `<Product>` vemos que tiene una serie de propiedades, y en este caso nos vamos a fijar en la propiedad `onClick`. Esta actuara como referencia al *callback* que hemos definido en `<Cart>` con el nombre `handleClick`. Si nos fijamos en este *callback* vemos que recibe dos argumentos: `id` y `event`, esto lo conseguimos gracias a utilizar la extension de función `.bind()` y el `id` asignado por propiedad:

```
<li onClick={this.props.onClick.bind(null, this.props.id)}>...</li>
```

Como ves es realmente sencillo comunicar a nuestro componente `<Cart>` que componente `<Product>` ha sido clickado. Ahora vamos a mejorar esta estructura definiendo algo mas escalable y mejor organizado, ya que ahora debemos definir la propiedad `onClick` por cada componente `<Product>`. Nuestro nuevo acercamiento se basara en `<Cart>` y `<Products>`:

cart.jsx

```
import React from 'react';
import Products from './products'

const store = [
  { id: 1, name: 'Macbook Air 11'},
  { id: 2, name: 'Macbook 2015'},
  { id: 3, name: 'Macbook Air 13'}
];

class Cart extends React.Component {

  handleClick = (id, event) => {
    console.log(id, event);
  };

  render() {
    return <Products store={store} onClick={this.handleClick} />
  }
};

export default Cart;
```

products.jsx

```
import React from 'react';

class Products extends React.Component {

  static propTypes = {
    store: React.PropTypes.array.isRequired,
    onClick: React.PropTypes.func.isRequired
  };

  render() {
    return (
      <ul>
        {
          this.props.store.map(function(product, index) {
            return (
              <li onClick={this.props.onClick.bind(null, product.id)}>
                {product.name}
              </li>
```

```
19              );
20            }.bind(this))
21          }
22        </ul>
23      );
24    }
25  };
26
27  export default Products;
```

Como vemos `<Products>` solo se define una única vez dentro de `<Card>` y sus propiedades pasan a ser:

- **`items`**: Conteniendo la lista de productos a representar.
- **`onClick`**: El *callback* que sera asignado a cada uno de los productos.

Acostumbrate a generar componentes de esta manera, para no tener que sufrir de *DRY* (*Don't repeat yourself*) creando definiciones de eventos sintéticos repetidas.

Componentes *Stateless*

Dentro de la comunidad de React cada vez suena mas fuerte y continuado, utilizar components *stateless*. Esto quiere decir que puedes definir tus clases React como una sencilla función JavaScript. Por ejemplo algo como esto:

```
function HelloStateless(props) {
  return <div>Hello {props.name}!</div>;
}

ReactDOM.render(<HelloStateless name="world" />, ...);
```

Utilizando la nueva sintaxis de funciones en EcmaScript 6 seria algo tal que así:

hello-stateless.jsx

```
const HelloStateless = props => {
  return (
    <small>Hello {props.caption}!</small>
  );
};

export default HelloStateless;
```

En el caso de que queramos definir propiedades en nuestro componente `<HelloStateless>` lo haríamos de la siguiente forma:

```
import React from 'react';

const HelloStateless = props => {
  return (
    <small>Hello {props.caption}!</small>
  );
};

HelloStateless.propTypes = {
  caption: React.PropTypes.string
};
```

```
12
13 HelloStateless.defaultProps = {
14   caption: ''
15 };
16
17 export default HelloStateless;
```

Como ves no tiene ningún misterio, simplemente una vez definido el retorno `<HelloStateless>` asignamos los dos atributos `propTypes` y `defaultProps`. Utilizar componentes *stateless* hace que simplifiquemos consideradamente la API de los mismos puesto que:

- No almacenan ningún tipo de estado interno
- No contienen instancia
- No tienen definidas las funciones de ciclo de vida

Estos componentes son transformaciones puras de su definición JSX. Sin embargo, y como hemos visto podemos seguir especificando `propTypes` y `defaultProps` como propiedades de la función.

Debido a que las funciones *stateless* no tienen una instancia propia, no puedes adjuntar una referencia `ref` a un componente *stateless*. Normalmente esto no es un problema, ya que este tipo de funciones no proporcionan una API imperativa. Sin embargo, si quieres buscar el nodo DOM de un componente *stateless* (algo muy común por otra parte) solo tienes una única solución: contenerlo dentro de un componente con estado.

En un mundo ideal, la mayoría de tus componentes React deberían de ser funciones *stateless* puesto que este tipo de componentes son mucho mas rápidos dentro del core de React. Este debe ser el patrón recomendado, siempre que sea posible.

TodoMVC con React

Ahora que ya has interiorizado todos los *internals* de React llega el momento de poner en práctica todos tus conocimientos. Vamos a desarrollar por completo el archiconocido TodoMVC[19] y lo vamos a hacer en menos de 250 lineas de código JavaScript. Para ello ademas de utilizar React vamos a hacer uso de Hamsa[20] para gestionar los datos dentro de esta *App*. Recuerda que React te da la libertad a elegir como gestionar tus modelos, en el caso de que sea necesario, Facebook ofrece una arquitectura llamada Flux[21] pero resulta demasiado pesada para *Apps* como TodoMVC.

La estructura de archivos y directorios que vamos a necesitar es la siguiente:

- **[components]**: Directorio con los componentes *Hijo*.
- **[models]**: Contendrá nuestra modelo de clase *Hamsa*.
- **app.jsx**: Componente *Padre*.
- **index.html**: Fichero HTML por defecto de TodoMVC.
- **package.json**: Definición de módulos necesarios.
- **webpack.config.js**: Configuración de tareas para el desarrollo de nuestra App.

Configurando webpack

Como gestor de procesos vamos a seguir utilizando Webpack por lo que antes de comenzar a desarrollar vamos a definir los módulos necesarios para llevar a cabo nuestro TodoMVC. En este caso utilizaremos los siguientes:

- **babel**: Modulo para poder transpilar el código EcmaScript 6 a React JavaScript
- **hamsa**: Modulo para poder crear modelos con la especificación `Object.Observe` de EcmaScript 7 (si has leido bien, ES7).
- **react** y **react-dom**: No hace falta explicacion ;)
- **spa-router**: Modulo para poder crear un *Router* en *SPA* (Single Page Applications).
- **todomvc-app-css** y **todomvc-common**: Modulos con todos los stylesheets por defecto de TodoMVC.
- **webpack**: Modulo gestor de tareas (es recomendable tenerlo instalado de manera global con `npm install webpack -g`).
- **webpack-dev-server**: Modulo para poder crear un servidor http local con *LiveReload*.

Con estos requisitos nuestro fichero `package.json` quedaría de la siguiente manera:

package.json

[19] http://todomvc.com

[20] https://github.com/soyjavi/hamsa

[21] http://facebook.github.io/flux/

```
{
  ...
  "devDependencies": {
    "babel-core": "^6.1.4",
    "babel-loader": "^6.0.1",
    "babel-plugin-react-transform": "^1.1.1",
    "babel-preset-es2015": "^6.1.4",
    "babel-preset-react": "^6.1.4",
    "babel-preset-stage-0": "^6.1.4",
    "hamsa": "*",
    "react": "^0.14.0",
    "react-dom": "^0.14.0",
    "spa-router": "*",
    "todomvc-app-css": "*",
    "todomvc-common": "*",
    "webpack": "^1.9.10",
    "webpack-dev-server": "*"
  },
  "scripts": {
    "start"               : "npm run build & npm run server",
    "server"              : "webpack-dev-server --hot",
    "build"               : "webpack"
  }
}
```

Fíjate que para facilitar un mejor flujo de trabajo he definido varios shortcuts NPM: `start`, `server` y `build`. Acostúmbrate a definir tus propios shortcuts que te ayuden a ti, y a tu equipo, a ser mas eficientes.

Ahora vamos a configurar nuestro sistema de tareas con WebPack[22]. Podríamos haber utilizado GruntJS[23] o Gulp[24], yo los he utilizado en el pasado, pero puedo asegurarte que WebPack es mucho mas eficiente, rápido y orientado a React que el resto de soluciones. En nuestro caso vamos a hacer que WebPack realice las siguientes tareas por nosotros:

- Analizar todos los archivos con extension `jsx` o `js`.
- Los ficheros `jsx` serán transpilados a código React *JavaScript*.
- Utilizara como componente *Padre* el fichero `app.jsx`
- Con cada cambio en el fichero `app.jsx` y todas sus dependencias con `import` ejecutaran una nueva compilación.

[22] http://webpack.github.io/
[23] http://gruntjs.com/
[24] http://gulpjs.com/

- La compilación se dejara en el directorio `build` y tendrá el nombre que hemos definido en el fichero `package.json`.
- Creara un servidor http local en `http://localhost:8080` y ejecutara un *reload* con cada nueva compilación.

webpack.config.js

```
var pkg = require('./package.json');

module.exports = {
  cache: true,

  resolve: { extensions: ['', '.jsx', '.js'] },

  context: __dirname,

  entry: { app: ['webpack/hot/dev-server', './app.jsx'] },

  output: {
    path: './build',
    filename: pkg.name + '.[name].js',
    publicPath: '/build/'
  },

  devServer: {
    host        : '0.0.0.0',
    port        : 8080,
    inline      : true
  },

  module: {
    loaders: [
      {
        test: /(\.js|\.jsx)$/,
        loader: 'babel',
        query: { presets: ['es2015', 'stage-0', 'react'] }
      }
    ]
  }
};
```

Para finalizar con esta sección fíjate que en nuestro fichero `webpack.config.js` he definido dos atributos `cache` y `context`. Con el primero de ellos incrementas considerablemente la velocidad de

compilado puesto que solo concatena al fichero resultante los ficheros que hayas modificado. Esto es realmente importante cuando haces `import` de librerías *grandes* como pueden ser MomentJS o la propia ReactJS. El segundo atributo, `context`, ayuda a Webpack a buscar todas las dependencias desde un determinado directorio.

Creando nuestra clase *Hamsa*

Antes de comenzar a desarrollar nuestros componentes React vamos a aprender que es Hamsa[25]. Con la futura especificación EcmaScript 7 aparece `Object.observe` una de las funcionalidades que va a facilitar mucho la forma de gestionar datos dentro de nuestras Apps. `Object.observe` te va a permitir poder observar las mutaciones de cualquier objeto nativo *JavaScript*, incrementando considerablemente el rendimiento y la gestión de memoria de tus *Apps*.

Hamsa[26] te permite utilizar hoy mismo esta implementación teniendo un *polyfill* para los navegadores que todavia no soporten `Object.observe` natívamente. Ademas Hamsa[27] te ofrece una seria de funcionalidades que te facilitaran mucho la gestión de las instancias de modelo que hayas creado, por ejemplo:

- Crear estructuras solidas de datos.
- Crear *observers* tanto para la clase base como para cada una de las instancias.
- Métodos para poder buscar, modificar o eliminar nuestras instancias.

Para nuestro TodoMVC vamos a crear una clase `Task` que tendrá únicamente dos campos:

- **`name`**: Contendrá el nombre de la tarea y sera de tipo *String*.
- **`completed`**: Contendrá el valor de si la tarea ha sido realizada, sera de tipo *Boolean* y por defecto sera *false*.

Ademas nuestra clase tendrá una serie de métodos estáticos:

- **`all()`**: Devuelve todas las instancias creadas utilizando el método implícito de Hamsa `find()`.
- **`active()`** y **`completed()`**: Devuelve las tareas pendientes utilizando el método implícito de Hamsa `find()` filtrando la propiedad `completed`.

task.js

[25] https://github.com/soyjavi/hamsa/
[26] https://github.com/soyjavi/hamsa/
[27] https://github.com/soyjavi/hamsa/

```
import Hamsa from 'Hamsa';
var Task;

module.exports = (function() {

  _extends(Task, Hamsa);

  function Task() {
    return Task.__super__.constructor.apply(this, arguments);
  }

  Task.define({
    name: { type: String },
    completed: { type: Boolean, default: false },
    created_at: { type: Date, default: new Date() }
  });

  Task.all = function() {
    return this.find();
  }

  Task.active = function() {
    return this.find({ completed: false });
  }

  Task.completed = function() {
    return this.find({ completed: true });
  }

  return Task;

})();
```

NOTA: En el caso de que quieras conocer todas las funcionalidades y capacidades de *Hamsa* te recomiendo que te pases por el site oficial en https://github.com/soyjavi/hamsa[28].

<App>: El componente *Padre*

Vamos a comenzar a diseñar nuestro componente *Padre* `<App>`, y comenzaremos definiendo sus dependencias:

app.jsx

[28] https://github.com/soyjavi/hamsa

```
import React from 'react';
import ReactDOM from 'react-dom';
import SPARouter from 'spa-router';
import Header from './components/header';
import Content from './components/content';
import Footer from './components/footer';
import Model from './models/task';
```

Como vemos requerimos los módulos generales `React`, `ReactDOM` y `SPARouter` definidos en el `package.json`, los componentes *Hijo* que vamos a desarrollar en este capitulo (`<Header>`, `<Content>` y `<Footer>`) y la clase Hamsa `Task` que hemos desarrollado en la sección anterior. Ahora debemos crear la definición de nuestro componente `<App>` siendo su responsabilidad:

- Debemos saber en que *url* estamos para mostrar las tareas dependiendo del filtro seleccionado.
- Debemos observar cualquier cambio en nuestras instancias de clase *Hamsa* `Task` para mostrar el numero de tareas pendientes.
- Dependiendo del filtro seleccionado enviaremos las tareas al componente `<Content>`.

app.jsx

```
...
class App extends React.Component {

  state = {
    context: 'all',
    pending: 0
  };

  componentDidMount() {
    Model.observe(function(state) {
      this.setState({ pending: Model.active().length });
    }.bind(this));

    SPARouter.listen({
      '/': this.setState.bind(this, { context: 'all' }),
      '/active': this.setState.bind(this, { context: 'active' }),
      '/completed': this.setState.bind(this, { context: 'completed' })
    });
    SPARouter.path('');
  }

```

```
22    render() {
23      return (
24        <div>
25          <Header />
26          <Content dataSource={Model[this.state.context]()} />
27          <Footer context={this.state.context} pending={this.state.pending} />
28        </div>
29      )
30    }
31  };
32
33  ReactDOM.render(<App />, document.getElementsByClassName('todoapp')[0]);
```

Como puedes ver hemos creado un atributo de estado `context` que contendrá el contexto de datos que estamos visualizando. Este atributo se modificara con cada cambio de url y es ahi donde entra en acción el método `SPARouter.listen()` el cual dependiendo en que *hashURL* estemos definirá el cambio en este atributo de estado. Ademas definimos un *Observer* para nuestra clase Hamsa `Task` el cual modificara el numero de tareas pendientes gracias al método estático `active()`.

Como sabes con cada cambio de estado React vuelve a renderizar el componente, a menos que modifiquemos el ciclo de vida, y nuestro componente `<App>` enviara la información necesaria a sus componentes *Hijo*.

```
1  <Header />
2  <Content dataSource={Task[this.state.context]()} />
3  <Footer context={this.state.context} pending={this.state.pending} />
```

Nuestro componente *hijo* `<Content>` establece una propiedad `dataSource` con las tareas que tiene que mostrar dependiendo del contexto en el que estemos. Si te fijas `Task[this.state.context]()` no deja de ser el nombre del método estático de nuestra clase *Hamsa*.

El componente *hijo* `<Footer>` recibirá dos propiedades `context` y `pending` rellenadas con los atributos de estado de nuestro componente *Padre*.

<Header>, componente *Hijo*

El componente *Hijo* `<Header>` solo tiene una responsabilidad, crear nuevas instancias de nuestra clase *Hamsa* `Task`.

header.cjsx

```
import React from 'react';
import Task from '../models/task';

class Header extends React.Component {

  handleKeyDown = (event) => {
    if (event.keyCode === 13) {
      event.preventDefault()
      new Task({ name: event.target.value });
      event.target.value = '';
    }
  };

  render() {
    return (
      <header className='header'>
        <h1>todos</h1>
        <input
          autofocus
          className='new-todo'
          onKeyDown={this.handleKeyDown}
          placeholder='What needs to be done?'
        />
      </header>
    )
  }
};

export default Header;
```

Como puedes comprobar no hace uso de ninguna propiedad ni estado, no todos los componentes de React lo necesitan. Simplemente creamos un contenedor con la etiqueta `<header>` el cual ademas de una cabecera `<h1>` contiene un elemento `<input>`. Cuando pulsemos cualquier tecla dentro de `<input>` se llamara a la función `handleKeyDown` la cual comprobara si estamos pulsando *return*. En caso de que así sea recogemos el valor actual y creamos una nueva instancia de `Task` con:

```
new Task({ name: event.target.value });
```

Al crearse una mutación en nuestra clase `Task` el *observer* que definimos en el componente *Padre* `<App>` se ejecutara lanzando una nueva renderizacion de todo el componente. Siendo este un gran ejemplo de lo que `Object.Observe` va a suponer a la hora de crear *Apps*, como ves no hace falta que `<Header>` comunique (por medio de una *callback*) nada a `<App>`.

<Content>, componente *Hijo*

El componente *Hijo* `<Content>` tiene la responsabilidad de representar todas las instancias de nuestra clase *Hamsa* `Task`.

content.cjsx

```
import React from 'react';
import Todo from './todo';

class Content extends React.Component {

  static propTypes = {
    dataSource: React.PropTypes.array
  };

  handleToggle = (event) => {
    let items = this.props.dataSource;
    for (let i = 0, len = items.length; i < len; i++) {
      items[i].completed = event.target.checked
    }
  };

  render() {
    return (
      <section className='main'>
        <input className='toggle-all' type='checkbox' onClick={this.handleToggle\
} />
        <label htmlFor='toggle-all'>Mark all as complete</label>
        <ul className='todo-list'>
        {
          this.props.dataSource.map(function(item, index) {
            return ( <Todo key={item.uid} data={item} /> );
          })
        }
        </ul>
      </section>
    )
  }
};

export default Content;
```

Como ves hacemos uso de la propiedad `dataSource` la cual por medio de la función `map()` nos ayuda crear tantas instancias del componente *Hijo* `<Todo>`. Fíjate que en `<Todo>` establecemos la propiedad key con el valor de `item.uid`, este es un identificador único que genera Hamsa, en el caso de que no vayas a usar Hamsa podrías utilizar el *index* del *Array*.

El ejercicio TodoMVC expone que tiene que existir la capacidad de poder modificar el estado de todas las tareas del contexto actual. Para ello hemos definido un método `handleToggle` el cual recogerá el valor actual del *CheckBox* y lo establecerá a todas las instancias contenidas en `dataSource`. Al igual que con el componente `<Header>` esto disparara el *observer* del componente *Padre* `<App>`.

<Todo>, componente *Hijo*

El componente *Hijo* `<Todo>` tiene la responsabilidad de representar toda la información de cada tarea. A su vez se podrá modificar tanto el nombre del como el estado de la misma, así como eliminar la tarea.

todo.cjsx

```
1  import React from 'react';
2
3  class Todo extends React.Component {
4
5    static propTypes = {
6      data: React.PropTypes.object
7    };
8
9    state = {
10     data: this.props.data,
11     editing: false,
12     value: this.props.data.name
13   };
14
15   handleToggle = (event) => {
16     this.state.data.completed = !this.state.data.completed
17   };
18
19   handleEdit = (event) => {
20     this.setState({ editing: true });
21   };
22
23   handleDestroy = (event) => {
24     this.state.data.destroy()
25   };
```

```
26
27    handleKeyDown = (event) => {
28      if (event.keyCode === 13) {
29        this.state.data.name = event.target.value;
30        this.setState({ editing: false });
31      }
32    };
33
34    handleChange = (event) => {
35      this.setState({ value: event.target.value });
36    };
37
38    render() {
39      let className = '';
40      if (this.state.data.completed) { className += ' completed' }
41      if (this.state.editing) { className += ' editing' }
42
43      return (
44        <li className={className}>
45          <div className='view'>
46            <input
47              className='toggle'
48              checked={this.state.data.completed}
49              onChange={this.handleToggle}
50              type='checkbox'
51            />
52            <label onDoubleClick={this.handleEdit}>{this.state.value}</label>
53            <button className='destroy' onClick={this.handleDestroy}></button>
54          </div>
55          <input
56            className='edit'
57            onChange={this.handleChange}
58            onKeyDown={this.handleKeyDown}
59            type='text'
60            value={this.state.value}
61          />
62        </li>
63      )
64    }
65  };
66
67  export default Todo;
```

Como ves solo disponemos de una propiedad `data` la cual contiene una instancia de *Hamsa* `Task` y es establecida como atributo de estado con el mismo nombre. Ademas disponemos de dos atributos de estado mas:

- **`editing`**: Define si estamos en modo edición de tarea o visualización.
- **`value`**: El valor actual del nombre de la tarea, el cual se modificara cuando estemos en modo edición.

`<Todo>` es un buen ejemplo de componente guiado por eventos, analicemos que es lo que hacemos con cada evento:

- **`handleToggle`**: Cuando se cambie el estado del *CheckBox* se modificara el atributo `completed` de la instancia *Hamsa*.
- **`handleEdit`**: Cuando se haga *DoubleClick* se activara el atributo de estado `editing` a `true`.
- **`handleDestroy`**: Ejecutara el método `destroy()` de la instancia *Hamsa*.
- **`handleKeyDown`**: Si pulsamos *return* finalizara el estado de edición y modificara el atributo name de la instancia *Hamsa*.
- **`handleChange`**: Con cada cambio en `<input>` se modificará el atributo de estado `value`.

Ten en cuenta que al estar trabajando directamente con una instancia *Hamsa* de `Task` cualquier modificación en sus atributos `name` o `completed` disparara el observer definido en el componente *Padre* `<App>`. Resumiendo cuando el *nieto* `<Todo>` modifica la instancia `Task` el *abuelo* `<App>` se entera de esos cambios y todo ellos sin utilizar ningún tipo de *callback* entre componentes. Esta parte es realmente importante, tienes que utilizar una buena estrategia

<Footer>, componente *Hijo*

La responsabilidad del componente `<Footer>` es filtrar las tareas por los diferentes contextos así como mostrar el numero total de tareas pendientes.

footer.js

```
import React from 'react';
import Task from '../models/task';

class Footer extends React.Component {

  static propTypes = {
    context: React.PropTypes.string,
    pending: React.PropTypes.number
  };

  static defaultProps = {
    pending: 0
  };

  state = {
    contexts: [
      {href: '#/', caption: 'All'},
      {href: '#/active', caption: 'Active'},
      {href: '#/completed', caption: 'Completed'}
    ]
  };

  handleClearCompleted = (event) => {
    var tasks = Task.completed();
    for (var i = 0, len = tasks.length; i < len; i++) {
      tasks[i].destroy();
    }
  };

  render() {
    var context = this.props.context;
    return (
      <footer className='footer'>
        <span className='todo-count'><strong>{this.props.pending}</strong> item \
left</span>
        <ul className='filters'>
        {
          this.state.contexts.map(function(item, index) {
            let className = item.caption.toLowerCase() == context ? 'selected' :\
 '';
            return (
              <li key={index}><a className={className} href={item.href}>{item.ca\
```

```
ption}</a></li>
            )
          })
        }
        </ul>
        <button
          className='clear-completed'
          onClick={this.handleClearCompleted}
        >
          Clear completed
        </button>
      </footer>
    )
  }
};

export default Footer;
```

Ademas de las propiedades `context` y `pending` establecidas desde el componente `<App>` definimos un atributo de estado `contexts` que contiene los filtros de tareas disponibles. Si nos fijamos en el método `render()` vemos que marcamos con el *className* `selected` el contexto actual:

```
{
  this.state.contexts.map(function(item, index) {
    className = item.caption.toLowerCase() == context ? 'selected' : '';
    return (
      <li>
        <a className={className} href={item.href}>{item.caption}</a>
      </li>
    )
  })
}
```

Otro de los requisitos del ejercicio TodoMVC es poder eliminar todas las tareas completadas. Para ello hemos definido el método `handleClearCompleted` el cual ejecuta el método estático `completed()` de nuestra clase *Hamsa* `Task`. En este método simplemente ejecutamos al método `destroy()` de cada instancia de `Task` disparando por tanto una mutación en `Task` y haciendo que nuestro componente *Padre* `<App>` vuelva a renderizarse.

Código fuente

Si quieres obtener todo el código fuente del ejercicio TodoMVC puedes hacer un *fork* de mi repositorio https://github.com/soyjavi/react-hamsa-todomvc[29]. Siéntete con la libertad de ofrecer mejoras al código así como de realizar tus propias soluciones utilizando otras librerías en vez de Hamsa y SPARouter.

[29] https://github.com/soyjavi/react-hamsa-todomvc

Siguientes pasos

Con esto hemos finalizado la segunda edición de este libro. React esta madurando continuamente y todos los días aparecen nuevas características y *addons*. En esta segunda edición podría haber hablado de muchas mas cosas pero considero que hay que dar mas tiempo.

Por ejemplo he probado 3 sistemas de testing, 2 librerías de componentes React y varias implementaciones de Apps Isomórficas. Como el buen software, los libros también pueden iterar y mejorar con el tiempo, por lo que en posibles ediciones de este libro me centraré en:

- **Apps Isomórficas**: Crear aplicaciones React desde NodeJS.
- **Testing**: Testear tus componentes React tanto a nivel de negocio como visualmente.
- **React-toolbox**: Utilizaremos los componentes contenidos en esta librería para crear aplicaciones con estilo Material Design.