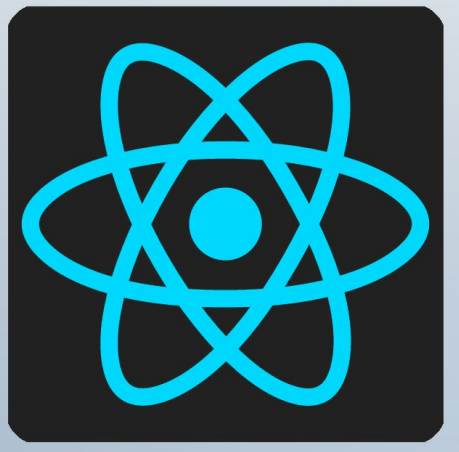
Manual de React



Miguel Angel Alvarez
Miguel Angel Durán
David García

desarrolloweb.com

desarrolloweb.com/manuales/manual-de-react.html

Introducción: Manual de React

Bienvenidos al Manual de React, en el que vamos a explicar el funcionamiento de la librería React, una herramienta estupenda para el desarrollo de interfaces de usuario y aplicaciones frontend (Javascript del lado del cliente). React fue creada por Facebook y actualmente es un software libre que se usa en cientos de páginas y aplicaciones de primer nivel. El desarrollo con React se basa en componentes, una arquitectura que nos permite una mayor reutilización del código, y la realización de aplicaciones complejas que más sencillas de programar y de mantener. React es una herramienta muy usada en el desarrollo del lado del cliente, no solamente en Facebook, sino también en miles de páginas y aplicaciones web. Es una evolución natural para aquellos que quieren llegar más lejos de lo que permiten otras librerías más básicas, como jQuery. En este manual iremos publicando los artículos sobre React, comenzando por una introducción general y una guía para los primeros pasos. Luego abordaremos asuntos tan importantes como el desarrollo de componentes, la gestión del estado y mucho más. Con este manual esperamos que aprender React se convierta en una experiencia sencilla y agradable para el desarrollador

Encuentras este manual online en:
http://desarrolloweb.com/manuales/manual-de-react.html

Autores del manual

Las siguientes personas han participado como autores escribiendo artículos de este manual.

Miguel Angel Alvarez

Miguel es fundador de DesarrolloWeb.com y la plataforma de formación online EscuelaIT. Comenzó en el mundo del desarrollo web en el año 1997, transformando su hobby en su trabajo.

David García

David García es Front-End web developer en Schibsted Spain y lleva ya más de diez años desarrollando proyectos web en agencias como HTTPcomunicació y Runroom o en plataformas de e-commerce como LetsBonus.

Miguel Angel Durán

Desarrollador apasionado de la programación, con tecnologías fullstack desde PHP a NodeJS. Su Javascript del lado del cliente lo prefiere con la librería React.

Introducción a React

En la primera parte de este manual aprenderemos qué es React, aclarando las características principales de esta librería para la creación de interfaces de usuario para la web. Además, a lo largo de los primeros artículos tendremos ocasión de hacer nuestros primeros ejemplos con React.

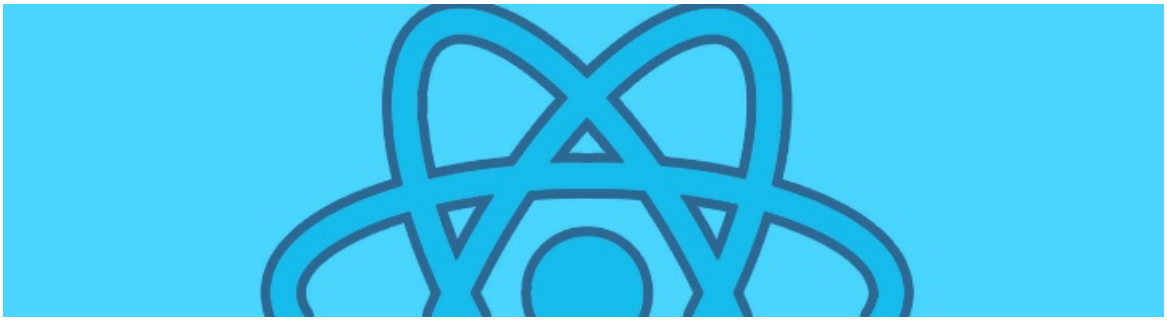
Tenemos dos acercamientos para la introducción a React, uno a partir de un paquete de software llamado Create React App, que sería la manera más aconsejable para comenzar un proyecto, y otra aproximación a los primeros pasos comenzando desde cero en un proyecto completamente vacío. Esta segunda introducción a los primeros pasos con React es más experimental y por tanto menos usada, pero que nos puede aportar datos interesantes para conocer mejor el funcionamiento interno de la librería y el sistema de anclaje para incorporar componentes en cualquier proyecto web.

Qué es React. Por qué usar React

Qué es React y los motivos por los que es una estupenda alternativa para el desarrollo de interfaces de usuario, o aplicaciones web completas del lado del cliente.

React es una librería Javascript focalizada en el desarrollo de interfaces de usuario. Así se define la propia librería y evidentemente, esa es su principal área de trabajo. Sin embargo, lo cierto es que en React encontramos un excelente aliado para hacer todo tipo de aplicaciones web, SPA (Single Page Application) o incluso aplicaciones para móviles. Para ello, alrededor de React existe un completo ecosistema de módulos, herramientas y componentes capaces de ayudar al desarrollador a cubrir objetivos avanzados con relativamente poco esfuerzo.

Por tanto, React representa una base sólida sobre la cual se puede construir casi cualquier cosa con Javascript. Además facilita mucho el desarrollo, ya que nos ofrece muchas cosas ya listas, en las que no necesitamos invertir tiempo de trabajo. En este artículo te ampliaremos esta información, aportando además diversos motivos por los que usar React como librería del lado del cliente.



Un poco de historia

React es una librería desarrollada inicialmente por Facebook. Es software libre y a partir de su liberación acapara una creciente comunidad de desarrolladores y entusiastas. Su creación se realizó en base a unas necesidades concretas, derivadas del desarrollo de la web de la popular red social. Además de facilitar el desarrollo ágil de componentes de interfaces de usuario, el requisito principal con el que nació React era ofrecer un elevado rendimiento, mayor que otras alternativas existentes en el mercado.

Detectaron que el típico marco de binding y doble binding ralentizaba un poco su aplicación, debido a la cantidad de conexiones entre las vistas y los datos. Como respuesta crearon una nueva dinámica de funcionamiento, en la que optimizaron la forma como las vistas se renderizaban frente al cambio en los datos de la aplicación.

A partir de ahí la probaron en su red social con resultados positivos y luego en Instagram, también propiedad de Facebook. Más adelante, después de su liberación y alentados por los positivos resultados en el rendimiento de React, muchas otras aplicaciones web de primer nivel la fueron adoptando. BBC, Airbnb, Netflix, Dropbox y un largo etc.

Cuál es el objetivo de React

Sirve para desarrollar aplicaciones web de una manera más ordenada y con menos código que si usas Javascript puro o librerías como jQuery centradas en la manipulación del DOM. Permite que las vistas se asocien con los datos, de modo que si cambian los datos, también cambian las vistas.

El código spaguetti que se suele producir mediante librerías como jQuery se pretende arquitecturizar y el modo de conseguirlo es a través de componentes. Una interfaz de usuario es básicamente creada a partir de un componente, el cual encapsula el funcionamiento y la presentación. Unos compoentes se basan además en otros para solucionar necesidades más complejas en aplicaciones. También permite crear otras piezas de aplicación cómodamente, como los test.

Comparación de React con otras librerías o frameworks

Con respecto a librerías sencillas como jQuery, React aporta una serie de posibilidades muy importantes. Al tener las vistas asociadas a los datos, no necesitamos escribir código para manipular la página cuando los datos cambian. Esta parte en librerías sencillas es muy laboriosa de conseguir y es algo que React hace automáticamente.

También en comparación con jQuery nos permite una arquitectura de desarrollo más avanzada, con diversos beneficios como la encapsulación del código en componentes, que nos ofrecen una serie de ventajas más importantes que los plugin, como la posibilidad de que esos componentes conversen e interaccionen entre si, algo que que sería muy difícil de conseguir con Plugins.

ReactJS solapa por completo las funcionalidades de jQuery, por lo que resulta una evolución natural para todos los sitios que usan esa librería. Podrían convivir pero no es algo que

realmente sea necesario y recargaría un poco la página, por lo que tampoco sería muy recomendable.

Ya luego en comparación con frameworks como es el caso de Angular o Ember, React se queda a mitad de camino, pues no incluye todo lo que suele ofrecer un framework completo. Pero ojo, a partir de todo el ecosistema de React se llega más o menos a las mismas funcionalidades, así que es una alternativa perfecta.

> Nota: Decimos que se queda a mitad de camino porque React por sí mismo es una librería y no un framework, puesto que React se ocupa de las interfaces de usuario. Quizás nos sirva decir que sería la "V" en un framework "MVC", aunque es solo una manera de hablar, puesto que React podría ocupar también parcelas de lo que sería la "C". Todo depende de nuestra manera de trabajar aunque, no obstante, esta posible carencia con respecto a los frameworks Javascript se soluciona con capas adicionales a React. Lo que podría interpretarse como una desventaja, muchos desarrolladores lo entienden como una ventaja con respecto a frameworks completos, ya que tú puedes desarrollar con React a tu gusto, aplicando aquellas herramientas y librerías adicionales que desees. Como resultado, es posible usar React de múltiples maneras y elegir aquella que mejor se adapte al proyecto o las costumbres de los desarrolladores.

No se puede decir de una manera objetiva si es ReactJS es mejor o peor que otras alternativas, porque eso ya entra más en el terreno de la opinión. Lo cierto es que muchas librerías se especializan en el "data-binding", pero React toma esa misma necesidad y la resuelve de otra manera. La diferencia es que React le pone más inteligencia a la necesidad de actualizar una vista cuando es necesario y lo consigue mediante el "DOM Virtual" o "Virtual DOM".

Qué es el Virtual DOM

A lo largo del Manual de React volveremos varias veces sobre el concepto de "Virtual DOM", que es una de las principales características de React. De momento, en líneas generales podemos decir que el virtual DOM es una representación del DOM pero en memoria, que usa React para aumentar sensiblemente el rendimiento de los componentes y aplicaciones front-end.

El Virtual DOM se basa en una idea bastante sencilla e ingeniosa. Básicamente hace que, cuando se actualiza una vista, React se encargue de actualizar el DOM Virtual, que es mucho más rápido que actualizar el DOM del navegador (DOM real). Cuando React compara el DOM Virtual con el DOM del navegador sabe perfectamente qué partes de la página debe actualizar y se ahorra la necesidad de actualizar la vista entera. Es algo muy potente, pero que se hace de manera transparente para el desarrollador, que no necesita intervenir en nada para alcanzar ese mayor rendimiento de la aplicación.

React es isomórfico

Éste es un concepto relativamente nuevo, pero muy interesante en el desarrollo de aplicaciones que se desean tengan un buen posicionamiento en buscadores. Básicamente se trata de, con un mismo código, renderizar HTML tanto en el servidor como en el cliente, rebajando la carga de trabajo necesaria para realizar aplicaciones web amigables para buscadores.

El problema de las aplicaciones Javascript es que muchas veces reciben los datos en crudo del servidor, o de un API o servicio web, en formato JSON. Las librerías Javascript y frameworks toman esos datos para producir el HTML que debe representar el navegador. Esta arquitectura representa la solución más adecuada para el desarrollo de aplicaciones web modernas, porque nos permite desacoplar el desarrollo del lado del servidor y el desarrollo del lado del cliente, pero se convierte en un aspecto negativo de cara al posicionamiento en buscadores como Google, debido a que el cuerpo de la página no tiene contenido.

> Nota: Al no tener contenido una página que recibe los datos en un JSON, Google no sabe qué palabras clave son interesantes y no otorga ranking para ellas. Con ello la aplicación o página no consigue posicionar. Google está haciendo cambios y ha comenzado a procesar el Javascript para saber los datos de una página, pero aún dista de las ventajas que supone que el contenido esté en el propio HTML que entrega el servidor.

React permite isomorfismo, lo que significa que, con el mismo código, somos capaces de renderizar tanto en el cliente como el servidor. Por tanto, cuando llega un buscador como Google, con la misma base de código se le puede entregar el HTML con el contenido ya renderizado, lo que lleva a que una aplicación React sea capaz de posicionarse tan bien como una aplicación web tradicional que renderice del lado del servidor, como es el caso de un desarrollo tradicional o un desarrollo basado en un CMS como WordPress.

Todo esto se consigue gracias a NodeJS y se puede reutilizar no solo la parte de la presentación, sino también la lógica de negocio. En resumen React permite isomorfismo, algo que le faltaba tradicionalmente a AngularJS 1.x. Ahora se ha resuelto en Angular 2.x. Aunque muchas librerías siguen sin ser capaces de soportar isomorfismo.

Ecosistema de React

Como hemos dicho, React en sí es una librería y, como tal, hay cosas que se deja del lado de fuera con respecto a soluciones aportadas por los frameworks MV* para Javascript. Sin embargo existe todo un ecosistema de herramientas, aplicaciones y librerías que al final equiparan React a un framework.

Hay herramientas que se usan en múltiples proyectos, como el caso de Redux o Flux, que aportan partes que React no se encarga. Éstos se ocupan del flujo de datos en React y lo resuelven de una manera optimizada, elegante, poniendo énfasis en la claridad de las aplicaciones. Como desarrolladores podemos escoger entre varios frameworks encargados del flujo de los datos, basados en React. Como otros ejemplos tenemos generadores de aplicaciones, sistemas de routing del lado del cliente, etc.

Por otra parte, al desarrollar en base a componentes reutilizables, permite que puedas usar el desarrollo de un proyecto en otro. Y por el mismo motivo, encuentras una amplia comunidad que libera sus propios componentes para que cualquier persona los pueda usar en cualquier proyecto. Por tanto, antes de desarrollar algo en React conviene ver si otro desarrollador ya ha publicado un componente que lo haga y en la mayoría de los casos, cuando se trata de cosas de ámbito general, veremos que siempre es así.

Hay componentes desde simples botones, sliders, tooltips, etc. Es muy sencillo que se pueda compartir, gracias a que los componentes son capaces de trabajar de manera independiente y que encapsulan funcionalidad para que no interaccionen con otros componentes si no se desea.

React Native es otra de las herramientas disponibles en el ecosistema, que permite llevar una aplicación escrita con Javascript y React como aplicación nativa para dispositivos iOS, Android, etc. Y se trata de aplicaciones nativas! el código Javascript con React se compila a nativo, en lugar de usar web views como ocurre generalmente en el desarrollo híbrido.

Conclusión

React es una librería completa, adecuada en muchos tipos de proyectos distintos. Nos permite un desarrollo ágil, ordenado y con una arquitectura mantenible, focalizada en componentes y que nos ofrece un gran performance.

Aunque React no se encarga de todas las partes necesarias para hacer una aplicación web compleja, la serie de componentes y herramientas diversas que encontramos dentro del ecosistema React, nos permite beneficiarnos de alternativas capaces de desarrollar cualquier cosa que podríamos hacer con un complejo framework.

Todo esto te lo vamos a explicar en el Curso de desarrollo de aplicaciones con React, de EscuelaIT. Es una excelente alternativa para comenzar con React de la mano de expertos desarrolladores que ya están usando la librería desde hace tiempo.

Agradecemos a nuestros compañeros Gorka Laucirica y Miguel Angel Durán, que en un evento en directo en DesarrolloWeb y EscuelaIT nos informaron sobre todas estas características de React.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en *25/02/2019*
Disponible online en *http://desarrolloweb.com/articulos/que-es-react-motivos-uso.html*

Primeros pasos con React

Primeros pasos con React, la librería Javascript para el desarrollo de interfaces de usuario, a través del paquete Create React App.

React, a veces conocida también como ReactJS, es una librería Javascript potente, focalizada en la sencillez y el rendimiento, usada en cientos de proyectos de primer nivel como Facebook, Netflix o Airbnb. Merece la pena conocerla puesto que se postula como una evolución natural para miles de desarrolladores que usan librerías más sencillas como jQuery. React en resumen aporta diversas ventajas, permitiendo escribir menos código y aumentando su facilidad de mantenimiento de las aplicaciones.

En el artículo anterior del Manual de React, ya ofrecimos una serie de información útil y

general sobre React y los motivos por los que usar la librería. En este artículo pretendemos presentar React de una manera más práctica y para ello vamos a utilizar un paquete muy interesante, llamado "Create React App", puesto a disposición por el propio Facebook.

Qué es Create React App

A no ser que seas un experto desarrollador frontend, la mejor alternativa para dar los primeros pasos con React es usar el paquete create-react-app. Te permitirá empezar muy rápido y ahorrarte muchos pasos de configuración inicial de un proyecto.

Generalmente cuando se construye un sitio o app web se tiene que lidiar con una serie de herramientas que forman parte del tooling de un frontend developer, como gestores de paquetes, de tareas, transpiladores, linters, builders, live reload, etc. Toda esta serie de herramientas pueden tener su complejidad si se quieren aprender con el suficiente detalle como para comenzar a usarlas en un proyecto, pero son esenciales para un buen workflow.

Por tanto, si queremos ser detallistas y proveernos de las herramientas necesarias para ser productivos y eficientes, se puede hacer difícil la puesta en marcha en un proyecto Frontend en general. Ahí es donde entra Create React App, ofreciéndonos todo lo necesario para comenzar una app con React, pero sin tener que perder tiempo configurando herramientas. Comenzaremos una app con un par de comandos sencillos, obteniendo muchos beneficios de desarrolladores avanzados.

Instalamos Create React App con el siguiente comando de npm:

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

Una vez lo hemos instalado de manera global, nos metemos en la carpeta de nuestros proyectos y lanzamos el comando para comenzar una nueva aplicación:

```
create-react-app mi-app
```

> Nota: Por si no quedó claro "mi-app" será el nombre de tu aplicación React. Obviamente, podrás cambiar ese nombre por uno de tu preferencia.

Mediante el anterior comando se cargarán los archivos de un proyecto vacío y todas las dependencias de npm para poder contar con el tooling que hemos mencionado. Una vez terminado el proceso, que puede tardar un poco, podemos entrar dentro de la carpeta de nuestra nueva app.

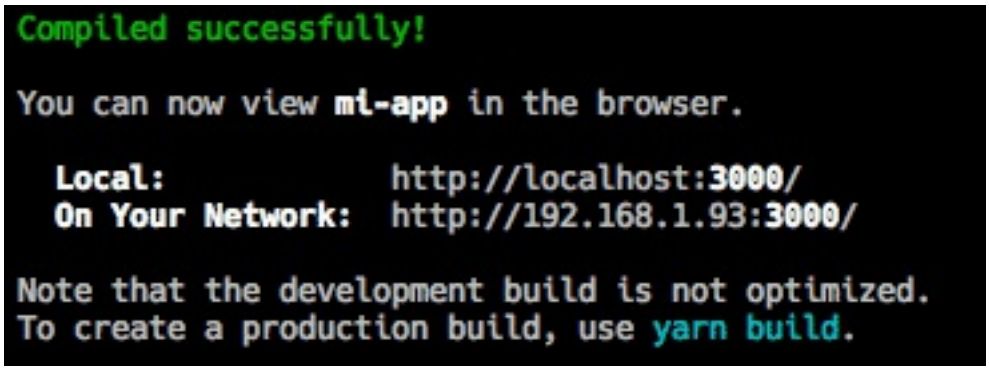
```
cd mi-app/
```

Y una vez dentro hacer que comience la magia con el comando:

```
npm start
```

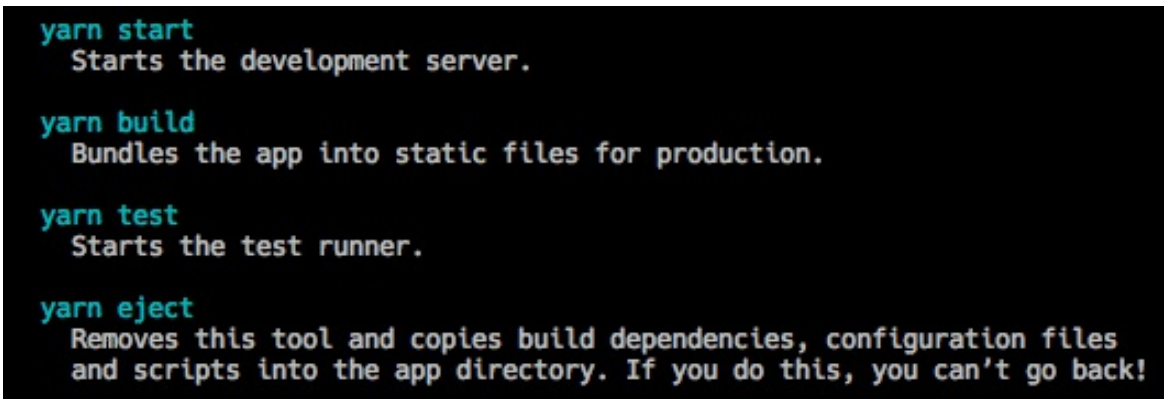
> Actualización: Al crear tu proyecto con create-react-app te informan de unos comandos ligeramente distintos, en los que se usa "Yarn" en lugar de "npm". Por si no lo sabes, Yarn es una herramienta equivalente a npm, creada por Facebook (como el propio React), que introduce algunos cambios en el gestor de dependencias que aumentan el rendimiento y la velocidad de la instalación de paquetes. Realmente es indiferente si usas Yarn o si usas npm, puesto que realmente los comandos funcionan con uno u otro sistema.

Observarás que, una vez lanzas el comando para iniciar la app, se abre una página en tu navegador con un mensaje de bienvenida. Ese es el proyecto que acabamos de crear. No obstante, en el propio terminal nos indicarán la URL del servidor web donde está funcionando nuestra app, para cualquier referencia posterior. De manera predeterminada será el http://localhost:3000/, aunque el puerto podría cambiar si el 3000 está ocupado.



Otros comandos disponibles en el proyecto con create-react-app

Create React App configura otra serie de comandos interesantes para el desarrollo del proyecto. Al terminar la instalacion del proyecto nos informan sobre ellos, tal como aparece en la siguiente imagen.



Uno de los comandos especialmente útil es "yarn build" (o su equivalente "npm build"), que sirve para generar los archivos de nuestro proyecto optimizados para producción.

```
npm run build
```

Otro comando super interesante es "yarn eject", que sirve para liberar todas las herramientas que están detrás de Create React App, de modo que puedas personalizar su funcionamiento. Create React App usa por debajo varias herramientas como Webpack, el transpilador Babel, un Linter, etc. Sin embargo, todas estas herramientas permanecen escondidas y no se pueden personalizar. Si tu proyecto necesita una configuración más específica de cualquiera de estas herramientas, puedes lanzar el comando "eject", con lo que consigues que toda la configuración escondida de Create React App salga a la luz y puedas trabajar de manera normal. Solo ten en cuenta que la acción "eject" no se puede deshacer!

```
yarn eject
```

Para mayores informaciones consultar la página del repositorio en Github: https://github.com/facebook/create-react-app

Carpetas de nuestra app React

El listado de nuestra app recién creada es bastante sencillo. Observarás que tenemos varias carpetas:

- node_modules: con las dependencias npm del proyecto
- public: esta es la raíz de nuestro servidor donde se podrá encontrar el index.html, el archivo principal y el favicon.ico que sería el icono de la aplicación.
- src: aquí es donde vamos a trabajar principalmente para nuestro proyecto, donde vamos a colocar los archivos de nuestros componentes React.

Además encontrarás archivos sueltos como:

- README.md que es el readme de Create React App, con cantidad de información sobre el proyecto y las apps que se crean a partir de él.
- package.json, que contiene información del proyecto, así como enumera las dependencias de npm, tanto para desarrollo como para producción. Si conoces npm no

necesitarás más explicaciones.
- .gitignore que es el típico archivo para decirle a git que ignore ciertas cosas del proyecto a la hora de controlar el versionado del código.
- yarn.lock Este archivo no se debe tocar, puesto que es código generado por Yarn. Su utilidad es ofrecer instalaciones de dependencias consistentes a lo largo de todas las instalaciones de un proyecto.

Componente raíz del proyecto con create react app

Para nuestros primeros pasos con React no necesitamos más que entrar en la carpeta src y empezar a editar su código. De entre todos los archivos que encuentras en la carpeta src por ahora nos vamos a quedar con uno en concreto, src/App.js, en el que se crea el componente principal de nuestra app.

De momento en nuestra aplicación sólo tenemos un componente, el mencionado componente raíz localizado en src/App.js. Sin embargo a medida que se vaya desarrollando la aplicación se irán creando nuevos componentes para realizar tareas más específicas, e instalando componentes de terceros, que nos ayuden a realizar algunas tareas sin necesidad de invertir tiempo en programarlas de nuevo. Así es como llegamos a la arquitectura frontend actual, basada en componentes.

> Nota: no lo hemos comentado, pero React fue una de las primeras librerías que están orientadas a componentes. Con React debemos construir componentes que son como etiquetas nuevas de HTML, que encapsulan un contenido, presentación y funcionalidad. A base de tener unos componentes que se apoyan en otros es como se consigue la arquitectura de componentes, que es la base del desarrollo más actual en las librerías y frameworks frontend más destacados.

Si abres src/App.js verás que la mayoría del código que aparece es como si fuera HTML, aunque encerrado dentro de un script Javascript. Es la manera en la que ReactJS trabaja con las vistas de la aplicación. En realidad no es código HTML sino "JSX", una extensión a la sintaxis de Javascript que nos permite de una manera amigable crear y mantener el HTML que necesitas para "renderizar" (pintar) los componentes. JSX tiene muchas cosas interesantes que comentar, pero no vamos a entretenernos hasta más adelante.

Hola Mundo en React

En el mencionado archivo src/App.js nos encontraremos poco código, pero para no despistarnos en nuestro primer "Hola Mundo", podemos borrar gran parte y quedarnos solamente con esto:

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

class App extends Component {
  render() {
    return (
      <div>
        <h1>Hola Mundo!</h1>
```

```
        <p>Bienvenidos a los primeros pasos con React</p>
      </div>
    );
  }
}

export default App;
```

En la primera línea "import React..." se está importando la librería React y la clase Component, que es la que usamos para, en la segunda línea, extender la clase App.

> Nota: En React hay varias alternativas para crear componentes. Esta que se ha utilizado es la forma de ECMAScript 6, que sería la recomendada. No necesitas preocuparte porque ese Javascript no sea compatible con todos los navegadores, ya que el propio Create React App nos ha configurado ya el transpilador para convertirlo a Javascript compatible con ECMAScript 5.

Las clases que usamos para implementar componentes React solo necesitan un método render para poder funcionar. En nuestro caso observarás que el componente que estamos creando, llamado App (como el nombre de la clase), solamente tiene ese método render().

```
render() {
  return (
    <div>
      <h1>Hola Mundo!</h1>
      <p>Bienvenidos a los primeros pasos con React</p>
    </div>
  );
}
```

Todo el código que coloques dentro del método render() será el marcado que usará el componente para su representación en la página.

> Nota: Es importante fijarse en una cosa: render() debe devolver el contenido con la palabra "return", colocando entre paréntesis el HTML necesario para pintar el componente en la página. Aunque lo parezca, el marcado devuelto por el método render no es exactamente HTML, sino JSX, como ya se adelantó en este artículo.

Guardando el archivo podrás observar como el navegador refresca la página automáticamente, gracias a las herramientas incluidas de serie dentro de Create React App.

Arquitectura de componentes

Quizás sea un poco ambicioso comenzar ahora a hablar de arquitectura de componentes, ya que es un concepto muy amplio, pero nos viene bien para ver cómo se puede conseguir desarrollar una aplicación a base de usar varios componentes.

Vamos a mejorar un poquito nuestro ejemplo anterior y para ello, dentro del mismo App.js, vamos a crear un segundo componente para ver cómo unos componentes se pueden basar en otros. El código de nuestro nuevo componente será el siguiente:

```
class OtroSaludo extends Component {
  render() {
    return (
      <p>Hola desde otro componente</p>
    )
  }
}
```

Usar este componente nuevo es tan simple como colocar una etiqueta en el componente que lo necesite usar, que tendrá el mismo nombre que el de la clase que acabamos de crear para este componente.

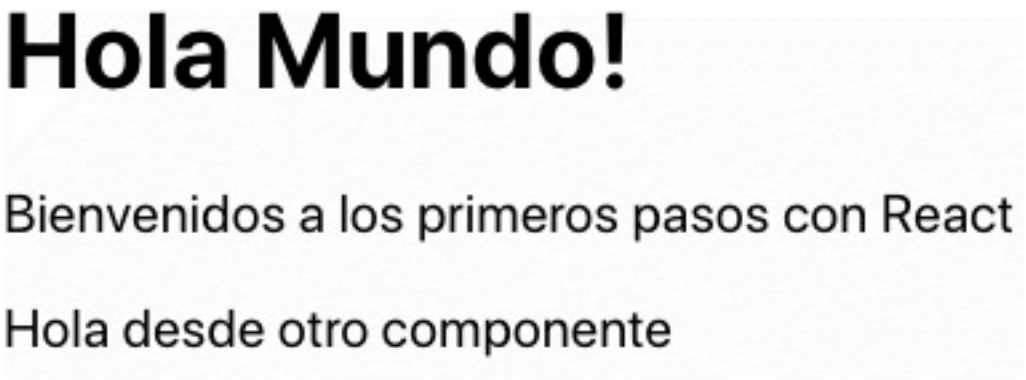
```
<div>
  <h1>Hola Mundo!</h1>
  <p>Bienvenidos a los primeros pasos con React</p>
  <OtroSaludo />
</div>
```

Debes observar cómo hemos usado la etiqueta "OtroSaludo" en la vista del componente App. Pondremos todo de momento dentro del mismo App.js, con lo que el código completo del archivo nos quedará como esto:

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

class App extends Component {
  render() {
    return (
      <div>
        <h1>Hola Mundo!</h1>
        <p>Bienvenidos a los primeros pasos con React</p>
        <OtroSaludo />
      </div>
    );
  }
}

class OtroSaludo extends Component {
  render() {
    return (
      <p>Hola desde otro componente</p>
    )
  }
}

export default App;
```

> Nota: obviamente, podríamos (deberíamos) separar el código de cada componente en archivos distintos. No encierra mucha dificultad, pero no vamos a entrar en ese detalle en este ejemplo, para facilitarnos las cosas en estos primeros pasos con React. Comenzaremos a separar el código de los componentes en archivos (un archivo ".js" para cada componente) en el artículo Componentes React mediante clases ES6.

En este momento nuestra aplicación te mostrará otro resultado cuando la veas en el navegador. Debería aparecer más o menos como se puede ver en la siguiente imagen:



Conclusión y vídeo

Esto es solo una pequeña muestra de lo que se puede hacer con un mínimo conocimiento de React. Cabe señalar que React es válido para hacer componentes que se pueden usar dentro de aplicaciones ya existentes, por ejemplo un proyecto PHP tradicional, que renderiza HTML del lado del servidor, como en aplicaciones enteramente del lado del cliente, que renderizan el HTML por medio de vistas React y con datos que traerás con JSON de un API REST o de cualquier otro medio.

Todo esto lo veremos en el futuro, pero si te interesa contar ya mismo con una formación completa y tutorizada para aprender React, te recomendamos participar en el Curso de React de EscuelaIT.

Como presentación de estas funcionalidades básicas de React y otras que no hemos tratado en este artículo te recomendamos asistir al vídeo siguiente, de los primeros pasos en React.

Para ver este vídeo es necesario visitar el artículo original en:
https://desarrolloweb.com/articulos/primeros-pasos-react.html

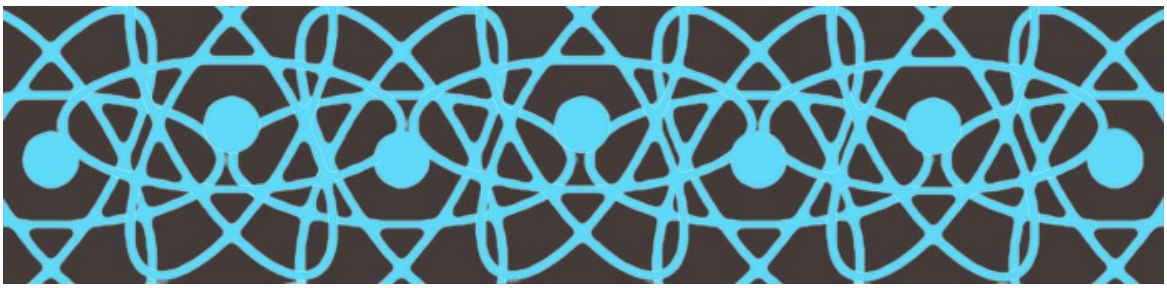
Este artículo es obra de *Miguel Angel Durán*
Fue publicado / actualizado en *26/02/2019*
Disponible online en *http://desarrolloweb.com/articulos/primeros-pasos-react.html*

Características de React

En este artículo vamos a conocer algunas de las principales características de React.

Antes de comenzar a programar nuevos ejemplos con React nos vamos a tomar nuevamente algo de tiempo para la teoría, en concreto para analizar la librería y así conocer más de cerca algunas de sus características. Ya tendremos tiempo en el Manual de React de ponernos algo más prácticos.

React es una librería para crear aplicaciones web, con Javascript del lado del cliente y del lado del servidor. Esa definición hace hincapié en una dualidad (cliente + servidor) que resulta una de las principales características de la librería, de las que ya hemos hablado anteriormente, en el artículo de Qué es React y por qué usarlo. Aunque volveremos sobre ese punto, ahora vamos a conocer otros datos de interés.



Composición de componentes

Así como en programación funcional se pasan funciones como parámetros para resolver problemas más complejos, creando lo que se conoce como composición funcional, en ReactJS podemos aplicar este mismo patrón mediante la composición de componentes

Las aplicaciones se realizan con la composición de varios componentes. Estos componentes encapsulan un comportamiento, una vista y un estado. Pueden ser muy complejos, pero es algo de lo que no necesitamos preocuparnos cuando estamos desarrollando la aplicación, porque el comportamiento queda dentro del componente y no necesitamos complicarnos por él una vez se ha realizado.

Esto es algo sobre lo que ya hemos hablado. En resumen, al desarrollar crearemos componentes para resolver pequeños problemas, que por ser pequeños son más fáciles de resolver y en adelante son más fáciles de visualizar y comprender. Luego, unos componentes se apoyarán en otros para resolver problemas mayores y al final la aplicación será un conjunto de componentes que trabajan entre sí. Este modelo de trabajo tiene varias ventajas, como la facilidad de mantenimiento, depuración, escalabilidad, etc.

Desarrollo Declarativo Vs Imperativo

En la experiencia de desarrollo con librerías más sencillas como jQuery, o el propio "Vanilla Javascript" realizamos un estilo de programación imperativo. En ese estilo se realizan scripts que paso por paso tienen que informar sobre qué acciones o cambios en el DOM se deben realizar. Hay que ser muy concisos en esas acciones, especificando con detalle cada uno de los cambios que se quieren realizar. La forma imperativa de declarar nos obliga a escribir mucho código, porque cada pequeño cambio se debe definir en un script y cuando el cambio puede ser provocado desde muchos sitios, cuando agregamos eventos, el código comienza a ser poco mantenible.

Sin embargo, el estilo de React es más declarativo, en el que nosotros contamos con un estado de la aplicación y sus componentes reaccionan ante el cambio de ese estado. Los componentes tienen una funcionalidad dada y cuando cambia una de sus propiedades ellos producen un cambio. En el código de nuestra aplicación tendremos ese componente, y en él se declarará de donde vienen los datos que él necesita para realizar su comportamiento. Podremos usarlo tantas veces como queramos declarando que lo queremos usar y declarando también los datos que él necesita para funcionar.

Quizás ahora, si no tenemos experiencia con otras librerías o frameworks de enfoque declarativo la idea pueda quedar algo confusa, pero lo iremos viendo mejor con los ejemplos.

Flujo de datos unidireccional

Ésta es otra de las cosas que facilita React, aunque no es exclusivo. En este modelo de funcionamiento, los componentes de orden superior propagan datos a los componentes de orden inferior. Los de orden inferior trabajarán con esos datos y cuando cambia su estado podrán propagar eventos hacia los componentes de orden superior para actualizar sus estados.

Este flujo tiende a ser unidireccional, pero entre componentes hermanos muchas veces se hace más cómodo que sea bidireccional y también se puede hacer dentro de React. Sin embargo, si tratamos siempre de mantener el patrón de funcionamiento unidireccional, nos facilitará mucho el mantenimiento de la aplicación y su depuración.

Performance gracias al DOM Virtual

El desempeño de React es muy alto, gracias a su funcionamiento. Nos referimos al desempeño a la hora del renderizado de la aplicación. Esto se consigue por medio del DOM Virtual. No es que React no opere con el DOM real del navegador, pero sus operaciones las realiza antes sobre el DOM Virtual, que es mucho más rápido.

El DOM Virtual está cargado en memoria y gracias a la herramienta que diferenciación entre él y el real, el DOM del navegador se actualiza. El resultado es que estas operaciones permiten actualizaciones de hasta 60 frames por segundo, lo que producen aplicaciones muy fluidas, con movimientos suavizados.

Isomorfismo

Es la capacidad de ejecutar el código tanto en el cliente como el servidor. También se conoce como "Javascript Universal". Sirve principalmente para solucionar problemas de posicionamiento tradicionales de las aplicaciones Javascript.

Como anteriormente ya hemos hablado sobre este punto, lo vamos a dejar por aquí.

Elementos y JSX

ReactJS no retorna HTML. El código embebido dentro de Javascript, parece HTML pero realmente es JSX. Son como funciones Javascript, pero expresadas mediante una sintaxis propia de React llamada JSX. Lo que produce son elementos en memoria y no elementos del

DOM tradicional, con lo cual las funciones no ocupan tiempo en producir pesados objetos del navegador sino simplemente elementos de un DOM virtual. Todo esto, como hemos dicho, es mucho más rápido.

React DOM y la herramienta de diffing, se encargarán más tarde de convertir esos elementos devueltos por JSX en DOM Real, sin que nosotros tengamos que intervenir.

Componentes con y sin estado

React permite crear componentes de diversas maneras, pero hay una diferencia entre componentes con y sin estado.

Los componentes stateless son los componentes que no tienen estado, digamos que no guardan en su memoria datos. Eso no quiere decir que no puedan recibir valores de propiedades, pero esas propiedades siempre las llevarán a las vistas sin producir un estado dentro del componente. Estos componentes sin estado se pueden escribir con una sencilla función que retorna el JSX que el componente debe representar en la página.

Los componentes statefull son un poco más complejos, porque son capaces de guardar un estado y mantienen lógica de negocio generalmente. Su principal diferencia es que se escriben en el código de una manera más compleja, generalmente por medio de una clase ES6 (Javascript con ECMAScript 2015), en la que podemos tener atributos y métodos para realizar todo tipo de operaciones. Los componentes statefull, con estado, necesitan tener un método render() que se encarga de devolver el JSX que usar para representarlo en la página.

Podemos decir que los componentes con estado son componentes "listos" y los que no tienen estado son "tontos". Los statefull se usan para resolver problemas mayores, con lógica de negocio, mientras que los stateless se usan más para interfaces de usuario.

Ciclo de vida de los componentes

React implementa un ciclo de vida para los componentes. Son métodos que se ejecutan cuando pasan cosas comunes con el componente, que nos permiten suscribir acciones cuando se produce una inicialización, se recibe la devolución de una promesa, etc.

En definitiva, todas estas cosas las iremos conociendo a lo largo del Manual de React de DesarrolloWeb.com. Aunque recuerda que si quieres aprender mucho mejor, con menos tiempo y ayudado por profesores, tienes el Curso de React de EscuelaIT.

Este artículo es obra de *David García*
Fue publicado / actualizado en *20/10/2016*
Disponible online en *http://desarrolloweb.com/articulos/caracteristicas-react.html*

Comenzar con React en un entorno vacío

Práctica de un proyecto React básico y simplificado al máximo, mediante un entorno vacío, que

comenzaremos con un archivo html en blanco.

Antes de comenzar la lectura: Este artículo aborda una posible forma de iniciar un proyecto React, sin embargo no es la más fácil y tampoco sería muy recomendada para una aplicación real. A nivel didáctico está bien, porque nos permite ir conociendo algunas de las piezas que forman una aplicación React, sin la envoltura de un entorno de desarrollo ni la necesidad de configurar herramienta alguna. Si estás aprendiendo React desde cero y no tienes mucha prisa, es una buena lectura para ir familiarizándote con algunas de las cosas que React dispone para desarrollar componentes y aplicaciones, como el método ReactDOM.render(). Si tienes prisa por aprender React en un entorno de desarrollo mejor preparado y más apropiado para una aplicación real, la recomendación sería comenzar tu aplicación con Create React App. Puedes ver esa introducción en el artículo de primeros pasos en React con Create React App.

Aclarado lo anterior, vamos a dar los primeros pasos con React, pero desde un proyecto completamente vacío, es decir, una página en blanco y sin la necesidad de descargar ninguna herramienta. Solo con un index.html y un poco de código.

Pero, si no es la manera más adecuada para desarrollar React, ¿Por qué explicamos cómo comenzar con React usando esta alternativa?. Para responder la pregunta aclararemos el objetivo del artículo: ofrecer una vía sencilla para poner en marcha React sin las complejidades de un entorno de desarrollo avanzado. Hay personas que prefieren enfrentarse a una nueva tecnología desde un lienzo vacío, o una página en blanco, donde tengamos pocas cosas y nos sean familiares. Además, al comenzar desde un proyecto completamente en blanco, tendremos ocasión de familiarizarnos mejor con las partes de una aplicación React.

> Nota: Si no te asusta tener un entorno complejo y prefieres código ES6 te interesa mejor ver cómo comenzar React a través de Create React App. Esa aproximación es la que vimos en el artículo Primeros pasos con React.

Además, si alguien ya tiene una estructura de aplicación y desea integrar React en ella, este artículo le dará una buena idea de cómo se podría comenzar. El problema en este caso es que React usa JSX, que requiere de un compilado por medio de Babel. A esta operación se la conoce como "transpilado", término a medio camino entre compilación y traducción. Por ese motivo al menos, para integrar Babel en un proyecto existente, estaríamos obligados a realizar una transpilación antes de poner en producción el código, lo que nos obligaría a definir nuestro entorno necesariamente.

CDN de React

Comenzaremos entonces cargando React desde el CDN, de modo que no tengamos que instalar en nuestro proyecto ninguna librería. React se ofrece con un CDN sobre el que podemos encontrar información y las versiones más actuales desde la página de links a CDN.

El código que deberemos usar a partir del CDN está dividido en varios scripts. Usaremos los tres siguientes:

```
<script crossorigin src="https://unpkg.com/react@16/umd/react.development.js"></script>
<script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
<script src="https://unpkg.com/babel-core@5.8.38/browser.min.js"></script>
```

Por orden de aparición, estos scripts sirven para lo siguiente:

- react es la propia librería React
- react-dom nos permite comenzar con react usando el lenguaje JSX, para escribir algo parecido a HTML / XML cuando queremos definir la salida de la aplicación
- babel-core sirve para convertir el JSX en código Javascript compatible con todos los navegadores

Como recomendación es interesante que usemos, en la etapa de desarrollo, los archivos de React y React-DOM no minimizados. Así podrás obtener errores más claros cuando algo no funcione en tu página. Los anteriores scripts eran las versiones para desarrollo, pero en la página de descargas de React encontrarás esos mismos scripts minimizados y adecuados para entornos de producción.

> Nota: Sobre JSX hemos hablado poco. Es un superset de Javascript que permite escribir código HTML dentro de un script Javascript. Es algo que se hace extraño al principio, pero que resulta muy potente. Puedes verlo inicialmente como un sistema de templating, donde de una manera clara y mantenible puedo generar salida en HTML, interpolando datos e incluso funcionalidad que viene de Javascript. Tampoco vamos a introducirnos ahora al detalle. Una de las grandes novedades de React en el panorama de desarrollo frontend es justamente el JSX.

Anclaje de la salida de React

Ahora que ya tenemos nuestras tres librerías necesarias, podemos crear un script que use react para producir una pequeña salida. Sin embargo, necesitamos algún sitio de nuestro HTML donde volcar esta salida. Este sitio será una sencilla etiqueta HTML a la que le pondremos un identificador para poder referirnos a ella.

La salida con React será inyectada en esa etiqueta. Nosotros simplemente al ejecutar React especificaremos el id que tiene el elemento donde inyectar esa salida. React se encargará de lo demás.

El código de nuestra página HTML, una vez colocados los scripts del CDN y el elemento de

anclaje quedaría de esta manera.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Test React desde una página en blanco</title>
</head>
<body>
  <div id="anclaje"></div>

  <script crossorigin src="https://unpkg.com/react@16/umd/react.development.js"></script>
  <script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
  <script src="https://unpkg.com/babel-core@5.8.38/browser.min.js"></script>
</body>
</html>
```

Como habrás observado, el elemento de anclaje es el DIV con id="anclaje". A continuación ya podremos poner en marcha React para hacer un sencillo "Hola mundo".

Script React

Ahora vamos con lo que todos estarán esperando ver, cómo ejecutar React para producir esa salida, que se inyectará en el elemento de anclaje. Es bien sencillo, a través del objeto ReactDOM.

ReactDOM permite hacer cosas como renderizar HTML en un lugar de la página. La gracia es que ReactDOM soporta escribir código en JSX, de modo que nos simplifica escribir HTML dentro del propio Javascript. El método que usamos de ReactDOM es render(), el cual requiere dos parámetros: 1) El JSX con el HTML a generar. 2) el objeto del DOM donde inyectar la salida.

El ejemplo podría ser algo como esto, aunque luego veremos alguna variación:

```
<script type="text/babel">
    var contenido = <h1>Hola Mundo!!</h1>;
    var anclaje = document.getElementById('anclaje');
    ReactDOM.render(contenido, anclaje);
</script>
```

El método "render" de ReactDom recibe dos parámetros. El contenido que debe renderizar y el objeto del DOM donde volcar el resultado de esa renderización. Esos dos parámetros son los que hemos generado en las variables contenido y anclaje generadas en las líneas anteriores.

Observarás, y seguramente te llamará la atención, la variable "contenido". A ella le hemos asignado algo que parece una cadena de texto con código HTML. Sin embargo, no es un string de código HTML (porque en ese caso debería ir entre comillas), sino código JSX. El JSX no se indica entre comillas, porque no es una cadena, sino código Javascript.

Como se ha dicho, conseguimos que el navegador entienda el código JSX gracias a la conversión de este script mediante babel-core. Es por ello que el script que estamos utilizando tiene un atributo type="text/babel", que indica que es un código que BabelJS debe procesar

antes de que el navegador lo interprete.

> Nota: Es muy importante el type="text/babel" porque si no lo pones el navegador te arrojará un error de sintaxis en el anterior código. Pero ojo, como hemos advertido antes, tampoco es la manera adecuada de trabajar con Babel, porque de este modo estaríamos poniendo la conversión del código del lado del cliente y eso es algo que se debe hacer en el proceso de despliegue. Si lo haces así como está en este ejemplo estarás obligando al navegador a dos cosas que producirán una aplicación poco optimizada: 1) descargar el código de Babel-core, con el peso que ello conlleva para la página y 2) procesar esa transpilación (traducción + compilación) del código JSX en tiempo de ejecución, haciendo el que procesado de la página sea más lento. Quizás no lo notes con una aplicación pequeña que se ejecuta en local, pero sería patente con una aplicación mayor ejecutada en remoto.

Como alternativa, veamos ahora el mismo código pero expresado de otra manera. En vez de generar variables en pasos anteriores enviamos los valores directamente al método render().

```
<script type="text/babel">
  ReactDOM.render(
    <h1>Hola Mundo React</h1>,
    document.getElementById('example')
  )
</script>
```

Ejemplo completo del hola mundo react sin setup del entorno de desarrollo

Ahora puedes ver el código de nuestra página, un archivo HTML que podrías poner en cualquier lugar de tu disco duro y ejecutar directamente con un doble clic. Sin dependencias de ningún otro paso que debas realizar para que esto funcione correctamente.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Test React desde una página en blanco</title>
</head>
<body>
  <div id="anclaje"></div>

  <script crossorigin src="https://unpkg.com/react@16/umd/react.development.js"></script>
  <script crossorigin src="https://unpkg.com/react-dom@16/umd/react-dom.development.js"></script>
  <script src="https://unpkg.com/babel-core@5.8.38/browser.min.js"></script>
  <script type="text/babel">
    var contenido = <h1>Hola Mundo!!</h1>;
    var anclaje = document.getElementById('anclaje');
    ReactDOM.render(contenido, anclaje);
  </script>
</body>
</html>
```

Es un ejemplo de React básico, pero como hemos dicho poco apropiado por varios motivos. Además, aunque el archivo se ejecute correctamente haciendo doble clic sobre él, te conviene acceder a los ejemplos a través de un servidor web, al menos ir acostumbrándote a ello, porque más adelante las cosas no funcionarán desde file:// y sí desde http://.

Espero que te haya parecido interesante esta práctica para comenzar con ReactJS sin tener que preocuparte de la creación de un entorno de desarrollo y una carpeta de aplicación.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en *25/10/2016*
Disponible online en *http://desarrolloweb.com/articulos/comenzar-con-react-entorno-vacio.html*

Creación de componentes con React

En la segunda sección del Manual de React vamos a abordar con mayor detalle el desarrollo de componentes. Recordemos que React es una de las primeras librerías que nos permiten el desarrollo basado en componentes, lo que nos ofrece muchas ventajas a la hora de realizar la arquitectura de la aplicación frontend.

En los próximos artículos abordaremos diversas maneras de crear componentes con React, basados en código ES5 (el método de React Create Class) y basados en código ES6 (el método más nuevo y recomendado para la creación de componentes en React).

Componentes React con createClass

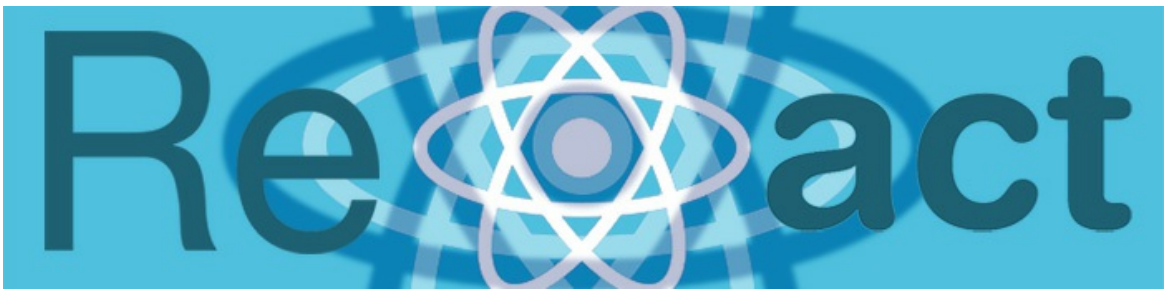
Cómo crear componentes ReactJS con el método más tradicional, el método createClass() del objeto React.

Comenzamos un tema importante en React, ya que en esta librería realizamos un desarrollo basado en componentes. Básicamente quiere decir que, mediante anidación de componentes podremos crear aplicaciones completas de una manera modular, de fácil mantenimiento a pesar de poder ser complejas.

Como ya hemos comentado anteriormente sobre las bondades del desarrollo basado en componentes, vamos a abordar más técnicamente este apartado, observando algunas cosas interesantes sobre cómo debemos desarrollar componentes mediante la alternativa más clásica.

Los componentes se definen de diversas maneras. Lo ideal es que encontremos la que más se adapta a nuestros gustos o necesidades. Veremos en esta ocasión un método llamado createClass(), que es el más tradicional a la hora de construir componentes, y que nos permite empezar sin la necesidad de crear un entorno de desarrollo complejo para producir nuestras aplicaciones.

> Nota: Este artículo es interesante porque createClass() es una manera bastante tradicional de crear componentes React. Hoy, dado que todos los navegadores modernos aceptan programación con Javascript ES6, no es tan relevante. Sin embargo, sigue siendo importante que conozcas este mecanismo, ya que seguramente lo encontrarás en tutoriales y código heredado de proyectos antiguos. Por tanto, te vendrá bien conocer los detalles de la creación de componentes con createClass. En futuros artículos estudiaremos alternativas que a nuestro juicio son más recomendables, basicamente la creación de componentes React con clases de ES6.



Actualización: Es importante aclarar que en la versión más nueva de React han cambiado la locaclización del método createClass(). El código de este ejercicio funciona, porque traemos React desde un CDN en la versión 15, pero en la versión 16 quitaron ese método del núcleo de React. Es todavía posible crear componentes con createClass(), pero hay unas pequeñas diferencias para acceder a esa funcionalidad. Lo puedes ver resumido en el artículo React Without ES6 de la documentación de React. De todos modos, como se ha mencionado, la manera recomendada actualmente para crear componentes en React es a partir de clases ES6, por lo que quizás sería más útil ir directamente a ese artículo para continuar con el manual.

Crear un componente con createClass()

Comencemos creando un componente por medio de un método que pertenece al objeto React, llamado createClass(). No es la manera más novedosa de crear componentes pero sí la más tradicional. Luego veremos cómo crear los componentes con clases de verdad, de las que nos permite realizar Javascript con ES6, pero createClass() es interesante de ver por dos motivos.

1. Porque nos permite crear componentes sin necesidad de crear un entorno de desarrollo complicado.
2. Porque hay muchos tutoriales más antiguos que éste donde podrás encontrar código como el que te vamos a proponer ahora. Así te será más fácil entender otras posibles referencias de estudio.

Nota: Este modo de desarrollo haría continuación al artículo Comenzar con React en un entorno vacío. Sería interesante echarle un vistazo a ese artículo para entender algunas cosas que ahora no vamos a explicar de nuevo.

En el código que encuentras a continuación observarás que para crear un componente usamos React.createClass(), enviando como parámetro la configuración del componente, indicada por medio de un literal de objeto Javascript.

```
var MiComponente = React.createClass({
  render: function() {
    return (
      <div>
        <h1>componente con createClass()</h1>
        <p>Este componente React lo creo como simple prueba. Lo hago con CreateClass porque quiero centrarme en ES5</p>
      </div>
```

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

Dentro del objeto de configuración enviado colocamos todas las cosas que debería tener nuestro componente. De momento observarás que le hemos pasado único método: render(), que es lo mínimo que debería tener todo componente básico.

El método render() contiene el código que debe usarse para representar el componente en la página. Como estamos dentro de React podemos usar la sintaxis JSX.

Como render() solamente sirve para devolver código que representar en la vista, lo único que encontramos es un return. Ya dentro del return estará el código JSX que construye la vista a usar para la representación del componente.

Usar un componente

Un componente que hemos creado mediante createClass() no produce ninguna salida. Tendremos que usar el componente para que realmente podamos verlo en funcionamiento en la página. Esta parte requiere de la otra pieza de React que necesitamos para funcionar: React-DOM.

> Nota: Volvemos a referirnos al artículo de hola mundo en React desde un archivo en blanco y entorno vacío, donde explicamos conceptos como React-DOM y el modo de usar estas piezas de React.

En este caso, como contenido a representar a React-DOM, le indicamos el código del componente que acabamos de crear. Recuerda que como segundo parámetro tenemos que indicar un objeto del DOM donde anclar la visualización del componente.

```
ReactDOM.render(<MiComponente />, anclaje);
```

Una de las posibilidades de desarrollar con componentes es que, una vez definido el componente, lo podemos usar varias veces en el código de nuestra aplicación. Para ello simplemente colocamos varias veces su etiqueta. Su contenido entonces se renderizaría en cada sitio donde lo estamos usando.

Sin embargo, esto así no te funcionará:

```
ReactDOM.render(<MiComponente /><MiComponente />, anclaje);
```

El contenido del método render() no es posible escribirlo con un número de etiquetas hermanas. Si quieres usar cualquier número de veces ese componente tendrás que envolverlo en una etiqueta cualquiera. Esto es porque JSX es como si produjera un return por cada

etiqueta que tiene y no es posible que una función devuelva dos return diferentes. La envoltura nos permite que solo se devuelva una única salida, aunque luego esa etiqueta pueda tener hijos, nietos, etc. en cualquier estructura de árbol.

Ahora con el siguiente código sí obtendremos resultados positivos.

```
ReactDOM.render(
  <div>
    <MiComponente />
    <MiComponente />
  </div>,
  anclaje
);
```

Observarás como hemos colocado unos saltos de línea en el código, simplemente para que se pueda leer mejor por el ojo humano. Estos saltos de línea son perfectamente válidos y dulcifican la sintaxis de JSX, a la vez que producen un código más mantenible y similar al HTML.

Ejemplo completo componente react con createClass()

El ejemplo completo de este ejercicio lo tienes a continuación.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Test React desde una página en blanco</title>
</head>
<body>
  <div id="anclaje"></div>

  <script src="https://unpkg.com/react@15.3.2/dist/react.js"></script>
  <script src="https://unpkg.com/react-dom@15.3.2/dist/react-dom.js"></script>
  <script src="https://unpkg.com/babel-core@5.8.38/browser.min.js"></script>
  <script type="text/babel">
  var MiComponente = React.createClass({
    render: function() {
      return (
        <div>
          <h1>componente con createClass()</h1>
          <p>Este componente React lo creo como simple prueba. Lo hago con CreateClass porque quiero centrarme en ES5</p>
        </div>
      );
    }
  });

  ReactDOM.render(
    <div>
      <MiComponente />
      <MiComponente />
    </div>,
    anclaje
  );

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

Ese código lo puedes ejecutar tal cual, copiando y pegando en cualquier fichero con extensión

.html y luego abriéndolo en tu navegador. No necesita de nada más para funcionar, salvo que tengas conexión a Internet para que se puedan cargar los códigos de los scripts cargados desde el CDN de React.

De momento no hay mucho más que agregar, solo que permanezcas atento a futuros artículos donde estaremos explicando otros mecanismos interesantes que nos permiten crear componentes con clases ES6.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en *09/11/2016*
Disponible online en *http://desarrolloweb.com/articulos/componentes-react-createclass.html*

Componentes React mediante clases ES6

Explicación y ejemplos sobre la alternativa de desarrollo de componentes React basada en la utilización de clases ES6.

En el pasado artículo del Manual de React comenzamos a explicar cómo crear componentes. En aquella ocasión lo hicimos con el método createClass() y ahora conoceremos otro mecanismo más actual, basado en clases de las que nos aporta ECMAScript 2015.

No es que por hacer los componentes mediante clases podamos realizar cosas adicionales a las que nos permitía el método de createClass(), sino que nos permitirá organizar el código de nuestras aplicaciones de una manera diferente, más práctica y mantenible. Básicamente podremos colocar cada componente en su módulo independiente e importarlo cuando sea necesario con imports de ES6. Al mismo tiempo, el hecho de ser clases nos permitirá de beneficiarnos de los beneficios de la Programación Orientada a Objetos en general.

En la documentación de React afirman que los componentes con clases ES6 tienen un mejor desempeño que usar createClass(), que se notará en aplicaciones grandes. Como desventaja, tendremos que escribir algún código adicional que React ya te hace de manera automática si usas createClass(), como por ejemplo el bindeo del contexto (this) en los métodos defindos como manejadores de eventos. Hablaremos de ello con detalle llegado el momento.

Quizás pienses en que agregar ES6 te añade la dificultad en la operativa de desarrollo, por la

necesidad de la transpilación. Pero dado que ya estamos en la necesidad de transpilar código JSX, será indiferente tener que transpilar también ES6.

> Nota: Para ir ajustándonos a un proyecto real, en lo sucesivo trabajaremos sobre una instalación de Create React App, que nos facilita tener el entorno ya listo, con transpiladores, linters, live reload, etc. Recuerda que esto ya lo explicamos y lo probamos en el artículo de los primeros pasos con ReactJS, así que esperamos que puedas despejar tus dudas allí.

Cada componente en su archivo

Ahora que vamos a trabajar con ES6, por facilitarnos la vida en lo que respecta a la organización del código del proyecto, vamos a tener el cuidado de crear cada componente en su propio archivo. Cuanto menores sean los archivos, más fáciles serán de mantener.

Cada archivo de código Javascript independiente forma parte de lo que se conoce como "módulo". Así que "cada componente en su módulo" sería más correcto de decir.

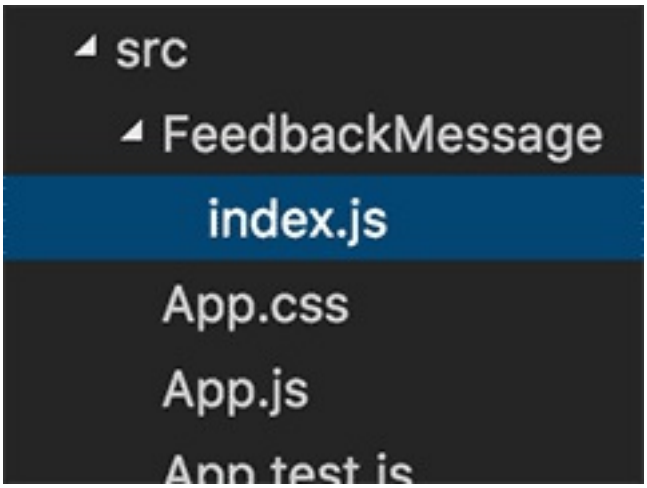
> Nota: Seguimos la regla de hacer componentes pequeños, con una responsabilidad acotada, igual que en la programación orientada a objetos las clases deben tener una responsabilidad limitada. Si te preocupa que sean muchos archivos pequeños, y eso produzca un sitio poco optimizado, cabe decir que más adelante en el proceso de "build" se empaquetarán en un "bundle", que es un archivo compactado, que recoge el código de nuestra aplicación en un solo bloque. Esto es algo que hace Webpack, que es otra de las cosas que nos incluye Create React App.

Los archivos de los componentes los colocarás en la carpeta "src", contenida en un proyecto en blanco de Create React App. Ya luego cómo desees organizar los módulos dentro de esa carpeta puede variar, pero una buena práctica sería crear una carpeta para cada componente o incluso carpetas de grupos de componentes que tengan una responsabilidad asociada. Iremos viendo estructuras más complejas en el futuro y también tus propios gustos y necesidades te irán marcando una pauta. Nosotros de momento nos contentaremos con colocar los archivos del componente en su carpeta independiente.

Así pues, dentro de src podrás crear una carpeta que se llame "MiComponente" o "mi-componente" o y dentro un "index.js" que tendrá el código del componente.

> Nota: Yo personalmente como regla prefiero no usar nombres de archivos o carpetas en el disco duro con mayúsculas y separo las palabras con guiones. Pero en React la costumbre más habitual es colocar mayúsculas en los archivos, de modo que sean igual que el nombre de las clases. En realidad poco importa en la práctica, sólo acuérdate el nombre de tu carpeta para cuando la tengas que importar, en el momento que quieras usar ese componente.

Pienso que casi ni hace falta, pero en la siguiente imagen tenemos una estructura de carpetas y archivo que vamos a crear.



Construir el componente, extendiendo de Component

Ahora dentro de nuestro index.js colocamos el código del componente. Comenzamos con la necesidad de importar React y la clase Component. React es la propia librería y Component es la clase de la que vamos a extender nuestros componentes.

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

A continuación podemos hacer ya la clase ES6 en la que definimos el componente.

```
class FeedbackMessage extends Component {
  render() {
    return (
      <div>
        Bienvenido a FeedbackMessage!
      </div>
    )
  }
}
```

Simplemente fíjate que estamos extendiendo de Component. El resto del código ya lo debes de reconocer, con su método render que devuelve el HTML a renderizar. Siendo que lo que hay en el return no es HTML, sino JSX.

Una vez tenemos la clase, nos tenemos que preocupar de exportarla, para que se conozca desde fuera de este módulo. Esto se consigue con la sentencia "export" de ES6.

```
export default FeedbackMessage
```

Estamos diciendo que exporte la clase "FeedbackMessage", que es la clase con la que hemos implementado el componente. Además con "default" estamos permitiendo que no tengamos que especificar el nombre de la clase cuando queremos importarla, en el siguiente paso.

Solo un detalle, que podríamos definir el export al mismo tiempo que declaramos la clase, con una cabecera combinada como esta:

```
export default class FeedbackMessage extends Component {
```

Usar un componente definido como clase

El único detalle nuevo ahora es que, para usar el componente, debemos importar su código definido en el módulo independiente. Lo conseguimos a partir de un "import" de ES6. Si estás familiarizado con esta práctica no tendrás duda alguna, pero vamos con ello.

```
import FeedbackMessage from './FeedbackMessage'
```

Ese import lo colocarás en cualquier módulo que desee usar tu componente nuevo. En el caso de un proyecto de Create React App tenemos el archivo App.js, que es donde se desarrolla todo el trabajo con la aplicación. Colocaremos allí el import de nuestro componente.

Una vez importado el componente lo podemos usar, ya dentro de un JSX. Usaremos el componente en el marcado que devolvemos mediante el método render del componente App que hay en App.js. Para que quede claro, este es el código que podríamos tener en el archivo App.js

```
import React, { Component } from 'react';
import FeedbackMessage from './FeedbackMessage'

class App extends Component {
  render() {
    return (
      <div>
        <h1>Manual de React</h1>
        <p>Creamos componentes con clases ES6</p>
        <FeedbackMessage />
      </div>
    );
  }
}

export default App;
```

Como puedes comprobar, una vez realizado el import del código del componente FeedbackMessage, podemos usarlo en el componente App. Para ello, en el JSX colocaremos la cantidad de código que sea necesario, junto con la etiqueta del componente que acabamos de crear.

Conclusión

Las clases ES6 nos facilitan muchas cosas en nuestro desarrollo y siempre es una buena idea tratar de aprovechar las ventajas de las versiones de Javascript más recientes. Para mantener una coherencia, en lo sucesivo del manual de React vamos a intentar centrarnos en el desarrollo mediante clases ES6.

Ten muy en cuenta Create React App, que te ofrece una estructura de proyecto con un entorno ya perfectamente configurado. También nuestros ejemplos los crearemos basando su funcionamiento en Create React App, porque así podemos abstraernos de varios aspectos de la configuración del proyecto. Recuerda que ya presentamos Create React App en el artículo de Primeros pasos con React.

En el próximo artículo comenzaremos con una de las utilidades más fundamentales para el desarrollo de componentes React: sus propiedades.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en *24/11/2016*
Disponible online en *http://desarrolloweb.com/articulos/componentes-react-clases-es6.html*

Propiedades en componentes React

Explicaciones sobre qué son las propiedades, en el desarrollo de componentes React. Veremos varios métodos de usar propiedades en componentes.

Uno de los recursos elementales que tenemos al desarrollar componentes React son las propiedades. Nos sirven básicamente para cargar de datos a los componentes, de modo que éstos puedan personalizar su comportamiento, o para transferir datos de unos componentes a otros para producirse la interoperabilidad entre componentes. Existen diversas maneras de pasar propiedades, aunque lo más común es hacerlo de manera declarativa, al usar el componente dentro del código JSX.

Ya hemos visto en el Manual de React que existen diversas maneras de implementar componentes, y todavía veremos otras formas. En este caso es importante porque, dependiendo de cómo hemos definido los componentes en React, las propiedades nos llegarán por parámetros o como atributos de objeto. Analizaremos ambos casos en este artículo. Observarás que es muy sencillo compartir información y aprender a personalizar el comportamiento de los objetos.

Propiedades en componentes definidos mediante clases

Comenzaremos observando cómo usar propiedades en componentes que hemos definido

mediante clases de ES6, ya que es un conocimiento más fresco, que aprendimos justo en el anterior artículo, dedicado a la creación de componentes usando ECMAScript 2015.

Al usar clases, las propiedades enviadas al componente (enseguida mostraremos cómo usar el componente y enviarle las propiedades) las vamos a recibir en forma de propiedades o atributos del objeto. Por tanto accedemos a ellas a través de "this". En concreto usaremos this.props para acceder a un objeto donde tenemos todas las propiedades definidas como atributos.

Imaginemos que a nuestro componente le pasamos dos propiedades, llamadas "nombre" y "app". Entonces podremos usar esas propiedades de la siguiente manera:

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

export default class FeedbackMessage extends Component {
   render() {
      return (
         <div>
            Bienvenido {this.props.nombre} a {this.props.app}
         </div>
      )
   }
}
```

El único detalle nuevo aquí es que estamos usando propiedades: this.props.nombre contendrá el valor pasado en la propiedad "nombre" y this.props.app el valor de la propiedad "app".

También es destacable el hecho de que nuestras propiedades se encuentran encerradas entre llaves. Las llaves son importantes, porque es la manera con la que se escapa un código JSX, permitiendo colocar dentro sentencias Javascript "nativo". Dicho de otra manera, al colocar las llaves estamos abriendo la posibilidad de escribir sentencias Javascript. Aquello que devuelvan esas sentencias se volcará como contenido en la vista.

Cómo pasar propiedades a los componentes

A la hora de usar un componente podemos enviarle propiedades con sus valores. Es una operación que se hace comúnmente de forma declarativa en el código JSX, cuando usamos un componente.

Cada propiedad se indica como si fuera un atributo del HTML, al que le indicamos su valor. Este sería un código donde se usa el componente FeedbackMessage, indicando los valores de sus propiedades. Fíjate como los valores son indicados como atributos del componente "nombre" y "app", que son las propiedades usadas en el ejemplo anterior.

```
import React, { Component } from 'react'
import FeedbackMessage from './FeedbackMessage'

class App extends Component {
  render() {
    return (
      <div className="App">
        <FeedbackMessage nombre="Miguel Angel Alvarez" app="Mi App React" />
      </div>
```

```
    );
  }
}
```

Propiedades en componentes generados a partir de funciones

Otro modo de definir este tipo de componentes sencillos es por medio de simples funciones, lo que dulcifica bastante la sintaxis de implementación de un componente. En este caso las propiedades se recibirán por parámetro en la función que implementa el componente.

> Nota: Este tipo de componentes sencillos normalmente se conoce como "componentes sin estado". En un capítulo posterior veremos una contraposición entre los componentes con y sin estado.

```
import React from 'react'

export default function (propiedades){
  return (
    <div>
      <p>Soy una función!</p>
      <p>Hola {propiedades.nombre}, estás en {propiedades.app}</p>
    </div>
  )
}
```

Observarás que al hacer el import ya no necesitamos traernos la clase Component, sino solamente React.

Luego, al definir la función se prescinde del método render, porque no estamos haciendo una clase. La propia función es el equivalente al método render() que teníamos al crear componentes por medio de una clase ES6. Por lo tanto, devuelve el JSX para representar el componente.

En lo que respecta al uso de propiedades debes observar que te llegan mediante un parámetro.

```
export default function (propiedades){
```

Al parámetro le he llamado "propiedades", pero como es un simple parámetro tú podrás llamarlo como quieras. Algo más común sería usar el nombre "props" que es como React las nombra habitualmente. He usado una palabra en español justamente para remarcar que, al ser un parámetro, yo le puedo poner el nombre que desee.

Luego accederás a esas propiedades con el nombre que hayas puesto a tu parámetro, seguido de un punto y el nombre de la propiedad.

```
Hola {propiedades.nombre}, estás en {propiedades.app}
```

Como antes, para volcar datos que viene de Javascript dentro de un JSX, usamos la notación de las llaves.

> Nota: El uso de un componente definido con una función no difiere del uso de cualquier otro componente, simplemente que debido a que estamos usando una función no podemos almacenar estado en este componente. Volveremos sobre el estado más adelante.

Ya que estamos en ES6, podríamos perfectamente usar las conocidas "arrow functions" para definir este componente. La sintaxis todavía quedará más resumida!

```
export default propiedades => (
  <div>
    <p>Soy una arrow function!</p>
    <p>Hola {propiedades.nombre}, estás en {propiedades.app}</p>
  </div>
)
```

Propiedades de componentes creados con createClass()

Si utilizamos la alternativa tradicional de creación de componentes React con createClass(), el uso de propiedades también se hace por medio de la referencia "this.props".

```
var FeedbackMessage = React.createClass({
  render: function() {
    return (
      <div>
        <p>Soy un componente con createClass()</p>
        <p>También puedo usar propiedades {this.props.nombre} y {this.props.app}</p>
      </div>
    )
  }
})
```

Conclusión a las propiedades en React

Ya conoces lo básico sobre las propiedades. Las hemos podido usar hasta de tres maneras distintas:

- En componentes definidos mediante clases ES6.
- En componentes sencillos, sin estado, definidos mediante una función.
- En componentes definidos con React.createClass.

Al final, las diferencias de trabajo con propiedades son mínimas, pero es importante que conozcas cada alternativa, porque seguramente te las encuentres en tu día a día desarrollando con React.

También vimos el modo de pasar valores a las propiedades de un componente en particular. La manera más habitual es mediante atributos en la etiqueta, de manera declarativa.

En el siguiente artículo comenzaremos a hablar con detalle de algo fundamental para poder entender el desarrollo con React, como es el estado de los componentes.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en *09/12/2016*
Disponible online en *http://desarrolloweb.com/articulos/propiedades-componentes-react.html*

Gestión del estado el los componentes React

El estado es una de las particularidades más importantes y características de la librería React. En los próximos artículos veremos qué es el estado y haremos la distinción entre componentes con y sin estado (statefull y stateless). Entender la gestión del estado es de suma importancia para poder desarrollar correctamente en React.

El estado en los componentes React

Qué son los componentes, con estado y sin estado, en React y cómo gestionar el estado de los componentes adecuadamente.

En este artículo vamos a adentrarnos un poco más en el desarrollo de componentes React, dedicando nuestro estudio al estado de los componentes. Existen dos tipos de componentes, con estado (statefull components) y sin estado (stateless components). Los veremos en contraposición.

Todavía no hemos visto demasiados ejemplos, así que aprovecharemos también para ver algunos otros y así ir familiarizándonos con el desarrollo usando esta librería Javascript.

Recuerda que en pasados artículos hemos introducido el desarrollo de componentes, aunque hasta ahora todos los componentes habían sido stateless. De todas las maneras de crear componentes nos decantaremos en preferencia por las clases ES6, en detrimento del método createClass(). Si quieres saber más sobre este punto, lee primero el artículo Componentes React mediante clases ES6.

Componentes con estado / sin estado

Para comenzar queremos analizar estos dos conceptos en contraposición. Cuando empezamos con React pueden provocar dudas, pero realmente es sencillo. Básicamente, la diferencia entre componentes con o sin estado estriba en que los componentes con estado permiten mantener

datos propios a lo largo del tiempo e implementar comportamientos en sus diferentes métodos del ciclo de vida.

Componentes sin estado

Los componentes sin estado no guardan ninguna información y por ello no necesitan de datos locales. Todos los componentes implementados hasta el momento eran stateless, sin estado. Eso no significa que no puedan personalizar su comportamiento, lo que se consigue con la ayuda de las propiedades de los componentes. Estas propiedades que nos pasan se podrán incluso transformar al producir una salida, de modo que sea acorde con las necesidades, pero no se guardará ningún valor y el componente no tendrá un ciclo de vida.

> Nota: El ciclo de vida de los componentes es algo que todavía no hemos abordado. Más adelante hablaremos de él, porque entenderlo es muy importante en React.

######### Ejemplo de componente sin estado

Para que quede más claro, esto sería una implementación de componente sin estado que recibe un timestamp como propiedad y genera una vista donde ese instante se muestra convertido en una fecha en español.

```
import React from 'react';

export default function (props) {
    var date = new Date(parseInt(props.timestamp, 10));
    var fecha = date.getDate() + '/' + (date.getMonth() + 1) + '/' + date.getFullYear();
    return (
        <span>{fecha}</span>
    );
}
```

Este componente se podría usar importándolo y luego colocando la etiqueta correspondiente. Me parece interesante mostrar cómo se usaría porque el componente en sí, tal como se ha definido, no tiene nombre. El nombre se lo asignamos a la hora de importarlo.

> Nota: El estilo anterior para la creación del componente se conoce en React como "stateless function component". Es una alternativa a la creación del componente a partir de una clase, que solo podemos usar para los componentes sin estado.

```
import React, { Component } from 'react'
import TimestampToDate from './utils/TimestampToDate'

class App extends Component {
  render() {
    return (
      <TimestampToDate timestamp={1475700297974} />
```

```
    );
  }
}
```

Otro detalle interesante aquí es la forma como le pasamos un valor numérico a un componente. Dentro del componente la propiedad timestamp se esperaría que fuera un entero y para que así sea tenemos que indicarle el valor sin comillas. Lo metemos entre llaves porque si no le colocas comillas JSX te obliga a que sea una expresión Javascript. Esa expresión se evalúa como un número entero.

Con este ejemplo queremos dejar claro que las propiedades de los componentes se pueden manipular para transformarlas en cualquier otro dato. Pero atención, en los componentes stateless las propiedades debemos tratarlas como valores de solo lectura, para evitar posibles situaciones inesperadas.. Si queremos manipular las propiedades y transformarlas en otra cosa lo más normal es guardemos los datos nuevos generados como variables, o estado si fuera necesario. Incluso, si solo se trata de una transformación sencilla para visualizar en la vista, podrías incluirla como una expresión Javascript embebida entre llaves dentro del JSX, aunque por claridad del código es preferible crear variables locales para generar esos cambios, como en el ejemplo anterior. En todo caso, quédate con que las propiedades deben trabajar como solo lectura.

Componentes con estado

Los componentes con estado son aquellos que almacenan datos de manera local al componente. Estos datos pueden variar variar a lo largo del tiempo bajo diversas circunstancias, por ejemplo por la interacción del usuario con el componente. Este tipo de componentes tienen algunas particularidades y posibilidades por encima de los componentes sin estadao que veremos a continuación.

Un ejemplo de componente con estado podría ser un contador. Ese contador puede incrementarse o decrementarse. Incluso podrían pasarnos como propiedad el valor inicial del contador, pero el valor actual de la cuenta lo guardaremos en el estado. Otro ejemplo podría ser un componente que se conecte con un API Rest. A este componente le podemos pasar como propiedad la URL del API y una serie de parámetros para realizar la solicitud al servidor. Una vez que recibamos los datos lo común será almacenarlos como estado del componente, para usarlos como sea necesario.

El componente podrá además reaccionar al cambio de estado, de modo que actualice su vista cuando sea necesario. Eso lo veremos cuando analicemos el ciclo de vida de los componentes.

######### Código necesario para implementar componente con estado

Seguro que estarás deseando ver ya un componente con estado. Enseguida nos ponemos con ello, pero quiero que veamos antes el código de "boilerplate" para crear un componente con estado.

```
import React from 'react'

export default class Contador extends React.Component {
```

```
  constructor(...args) {
    super(...args)
  }

  render() {
    return (
      <div>Esto aun no tiene estado!</div>
    )
  }
}
```

Lo primero decir que este componente todavía no tiene un estado implementado, es solo un código de partida para ver un par de cosas.

Como primer detalle no estamos importando implícitamente la clase Component, para hacer el extends. No es problema porque depende de React. Así que ahora estamos haciendo "extends React.Component", lo que es perfectamente válido. Esto no tiene mucho que ver con el tema que nos ocupa, de los estados, pero así vemos más variantes de codificación.

Más relevante en el código anterior es el constructor. En este boilerplate la verdad es que el constructor no sirve para nada, porque realmente no hemos inicializado nada (como sabes, los constructores resumen las tareas de inicialización de los objetos). En este caso simplemente estamos llamando al constructor de la clase padre, super(), pasándole los mismos argumentos que nos pasaron a nosotros.

> Nota: Eso de "...args" es una desestructuración, algo que nos viene de ES6, que permite en este caso recibir todos los argumentos o parámetros enviados a una función, sin necesidad de indicarlos uno a uno. Puedes obtener más información en el Manual de ES6, artículos del operador Rest y el operador Spread.

Generalmente en los componentes que tienen estado necesitamos inicializarlo, por lo que el sitio más correcto sería el constructor. Si realizamos un constructor tenemos que asegurarnos que se invoque al constructor de la clase padre, que realiza una serie de tareas genéricas para todos los componentes de React. Como esa invocación hay que hacerla explícita al sobreescribir el constructor, nos obliga a escribir la llamada a super(). Enseguida veremos cómo inicializar el estado, pero he querido mostrar ese código, aún sin la inicialización, para comentar este detalle del constructor.

En el boilerplate encuentras también el método render(), que ya sabemos que es el que nos sirve para definir la representación del componente.

######### Ejemplo de componente con estado

Ahora veamos ya una implementación de un componente completo con estado. En nuestro ejemplo vamos a crear el típico del contador. El valor actual del contador será nuestro estado y lo tendremos que inicializar en el constructor.

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

```
export default class Contador extends React.Component {
  constructor(...args) {
    super(...args)
    this.state = {
      contador: 0
    }
  }

  render() {
    return (
      <div>Cuenta actual: {this.state.contador}</div>
    )
  }
}
```

Ahora nuestro constructor ya tiene sentido, porque está realizando la inicialización de la propiedad "state" del componente. Como puedes ver el estado es un objeto, en el que ponemos tantos atributos como sea necesarios guardar como estado.

A la hora de renderizar el componente, por supuesto, podremos usar el estado para mostrar la salida. En este caso puedes ver cómo se vuelca el estado en la vista, con la expresión {this.state.contador}. Algo muy parecido a lo que hacíamos con las propiedades, solo que ahora los datos nos llegan del estado.

Solo nos falta implementar un botón para incrementar ese contador para ello tenemos que entrar en un tema nuevo, que son los eventos. Veamos el siguiente código.

```
import React from 'react'

export default class Contador extends React.Component {
  constructor(...args) {
    super(...args)
    this.state = {
      contador: 0
    }
  }

  incrementar() {
    this.setState({
      contador: this.state.contador + 1
    })
  }

  render() {
    return (
      <div>
        <span>Cuenta actual: {this.state.contador}</span>
        <button onClick={this.incrementar.bind(this)}>+</button>
      </div>
    )
  }
}
```

Ahora tenemos un botón y al hacer clic sobre él se invocará a la función incrementar(). Sin querer entrar en demasiados detalles sobre eventos, pues no es el asunto de este artículo, cabe decir que se pueden definir como si fueran atributos de los elementos o componentes. Como valor le colocamos su manejador de eventos. Además el "bind(this)" lo hacemos para bindear el contexto. Sobre eventos hablaremos más adelante.

Lo que es interesante, para lo que respecta al estado, está en el manejador de evento incrementar(). Este usa el método setState() para modificar el estado. El detalle que no se te puede escapar es que, para manipular el estado no se debe modificar "a pelo" (a mano) la propiedad this.state, sino que tenemos que hacerlo a través de this.setState(). El motivo es que setState(), además de alterar el estado, desencadena toda una serie de acciones implementadas en el core de React, que se encargan de realizar todo el trabajo por debajo para que ese cambio de estado tenga una representación en la vista.

Dicho de otra manera, en el momento que cambiemos el estado con setState(), se pone en ejecución el motor de React para que se actualice el DOM virtual, se compare con el DOM del navegador y por último se actualicen aquellos elementos que sea necesario (porque realmente hayan cambiado). Si no usamos setState() todas esas operativas no se producirían y los componentes empezarían a funcionar de manera no deseada.

Asincronía en la manipulación del estado

Hay un detalle extra que no queremos dejar pasar, sobre la asincronía en el manejo del estado por parte de React. Resulta que React, por motivos de rendimiento, puede llegar a acumular varias llamadas a setState, para procesarlas en un mismo instante.

Lo anterior significa que this.state puede ser actualizado de manera asíncrona. Es decir, this.setState() puede no ejecutarse inmediatamente, sino esperar a que React juzgue oportuno hacer las actualizaciones del estado. Esto podría resultar en una situación complicada de resolver, cuando el cálculo del nuevo valor del estado necesita basarse en el estado que haya anteriormente.

```
incrementar() {
  this.setState({
    contador: this.state.contador + 1
  })
}
```

En el código del método incrementar anterior, nos basamos en el estado actual del contador, para incrementarlo en una unidad. Sin embargo, si no sabemos realmente cuándo se va a ejecutar this.setState. Por ello, podría ocurrir que ese incremento se realice a partir de un valor del estado que no era realmente el válido.

Para resolver esta situación en React se puede usar un mecanismo de manipulación del estado basado en una función. Veamos el código para clarificarlo un poco.

```
incrementar() {
  this.setState(prevState => {
    return { counter: prevState.counter + 1};
  });
}
```

Como puedes ver, en lugar de enviarle un objeto a setState() le estamos enviando una función. Esa función es capaz de recibir como parámetro el estado actual. Entonces, el nuevo estado lo calculo en función del estado actual.

Si habías probado ejemplos anteriores en los que no habías tenido problemas con el cálculo del estado es normal. Aunque sea asíncrona la actualización del estado es muy rápida, por lo que es fácil no percibir ningún comportamiento anómalo. Simplemente tenlo en cuenta como una buena práctica en React.

Conclusión

De momento hemos dado bastante información sobre el estado de los componentes, hemos podido distinguir componentes con estado y componentes sin estado, y aunque todavía nos quedan cosas que aprender, hemos podido ver ejemplos para hacernos una idea de cada tipo de componente.

En futuros artículos tenemos que abordar otros asuntos clave, como la inicialización de propiedades y estado ante diversos estilos de codificación, así como el ciclo de vida de los componentes, que es necesario conocer a fondo para resolver muchas de las necesidades de desarrollo en React.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en *14/12/2016*
Disponible online en *http://desarrolloweb.com/articulos/estado-componentes-react.html*

Propiedades y estado predeterminado en componentes React

Cómo definir predeterminados para las propiedades de los componentes React, así como para su estado.

En capítulos anteriores del Manual de React tuvimos la ocasión de aprender a crear, y usar, propiedades en los componentes. También analizamos el estado de los componentes y aprendimos a crear componentes statefull, con sus métodos para definir y alterar el estado, así como representar las vistas dependiendo del estado.

En este capítulo vamos a analizar los modos mediante los cuales podemos definir tanto los valores predeterminados de las propiedades, como del estado. Nos detenemos en este punto porque, dependiendo de cómo se creen los componentes, el modo de trabajo será diferente.

Antes de comenzar cabe simplemente recordar las variantes básicas para la creación de componentes, que ya conoces:

- Crearlos con el método React.createClass()
- Crearlos mediante clases de Javascript ES6

Primer render

Estas operaciones de carga de valores predeterminados en propiedades y estado se realizan guiados por unos métodos específicos de React, que se ejecutan antes de la primera renderización del componente, es decir, antes de representarse en la página.

Como primer render nos referimos a la primera vez que se representa el componente dentro de la página, y se inyecta en el DOM del navegador. Cuando ocurre esa operación de renderizado, el componente debe tener tanto sus propiedades como su estado inicializados.

> Nota: Estos métodos están muy ligados al ciclo de vida de los componentes, puesto que se ejecutan en etapas muy concretas de la vida de un componente. Analizaremos con detalle el ciclo de vida en el próximo artículo.

Métodos para definir propiedades y estado predeterminado

Vamos a comenzar viendo cómo se usan los métodos de renderizado con la variante de creación de componentes a partir clases ES6 y luego veremos la alternativa en ES5 mediante createClass().

Para ambas alternativas debemos tener en cuenta los siguientes procedimientos comunes:

- La inicialización de las propiedades se realiza una vez para todos los elementos de la clase. Si tenemos un componente que se usa 5 veces, los valores de las propiedades predeterminadas se definirán una única vez.
- La inicialización del estado se realiza una vez para cada instancia del componente. Si tenemos un componente que se usa en 5 ocasiones, la inicialización de su estado se realizará 5 veces, una para cada instancia del componente.

Mecanismo de inicialización de propiedades y estado con clases ES6

En el caso de la inicialización de propiedades, en componentes creados con clases de ES6, se debe realizar definiendo una propiedad "defaultProps" en la clase que implementa el componente. En el caso de la inicialización del estado, la operativa se realiza en el constructor. Se verá claramente con un ejemplo enseguida.

Nota: En el caso de la inicialización de propiedades, como se realiza una única vez para todos los componentes de un tipo, no tiene sentido realizarlo en el constructor, puesto que en ese caso se realizaría para cada una de sus instancias. Por tanto, cuando se ejecuta el constructor ya estarán inicializadas las propiedades con sus valores por defecto, o aquellos definidos al usar el componente.

En la alternativa de creación de componentes mediante clases ES6 el código que tendríamos que generar para la inicialización de las propiedades y estado sería el siguiente:

```
import React from 'react'

export default class CicloVida extends React.Component {
  constructor(...args) {
    super(...args)
    console.log('Ejecuto constructor', ...args)
    this.state = {
      estado: 'Inicializado en el constructor'
    }
  }

  render() {
    return (
      <div>
        <p>Componente con propiedades y estado inicializado</p>
        <p>Estado: {this.state.estado}</p>
        <p>Propiedad: {this.props.propiedad}</p>
      </div>
    )
  }
}

CicloVida.defaultProps = {
  propiedad: 'Valor por defecto definido para la propiedad'
}
```

Como estarás observando, la inicialización de propiedades se hace una vez definida la clase, mediante una propiedad que usamos dentro de la propia clase. Al no estar en el código de la clase, se ejecutará una única vez.

Ya para la inicialización de estado, se define en el constructor, con lo que se ejecutará para cada componente instanciado de manera independiente. Esto lo vimos con detalle en el artículo sobre el estado de los componentes.

Inicialización predeterminada de propiedades y estado con createClass (ES5)

Si usamos el mecanismo de createClass, la inicialización de los componentes se realiza de manera ligeramente diferente, por medio de dos métodos específicos que tenemos que crear en el componente, uno para inicializar las propiedades y otro para el estado. React se encargará de invocar una única vez la inicialización de las propiedades, independientemente del número de componentes creados de esa clase, y de invocar una vez para cada componente el método de inicialización del estado.

Nota: Como hemos señalado en varios momentos del manual de React, el método createClass no es el recomendado actualmente para la creación de componentes. Por ello

esta parte quizás te la puedes saltar, si es que todos tus componentes están desarrollados mediante clases ES6, pues será indiferente para ti.

Estos dos métodos los vamos a ver por orden de ejecución.

getDefaultProps()

Este método se ejecuta primero y sirve para generar valores por defecto para las propiedades de un tipo de componente. Con tipo de componente nos referimos a todas las instancias que un componente puede generar. Por ejemplo, si un componente se usa 5 veces, este método se ejecutará una única vez, antes de comenzar a instanciar elementos concretos de ese componente.

Este método debe devolver un objeto, con el conjunto de propiedades cuyos valores predeterminados se quiera definir.

Los valores predeterminados en las propiedades funcionan de la siguiente manera.

Si al usar el componente se indican los valores de las propiedades, se toman esos valores dentro de la instancia del componente. No se indicaron los valores de esas propiedades al usar el componente, se crearán igualmente esas propiedades, cargando sus valores definidos mediante getDefaultProps().

Este método solo es posible usarlo cuando estamos trabajando con la variante de creación de componentes React.createClass().

getInitialState()

Este método se ejecuta después de haberse definido las propiedades por defecto. Sirve para inicializar el estado inicial de un componente y solamente funciona en componentes creados mediante React.createClass(). A diferencia de getDefaultProps(), este método se ejecuta una vez por instancia del componente.

El objeto que devuelva getInitialState() será lo que se inicialice como estado del componente.

Veremos ahora un ejemplo sobre cómo se podrían realizar estas inicializaciones mediante el método de creación de componentes de React.createClass().

```
var ComponenteInicializacion = React.createClass({
  getDefaultProps: function() {
    console.log('Ejecuto getDefaultProps')
    return {
      propiedad: 'Valor por defecto de una propiedad'
    }
  },

  getInitialState: function() {
    console.log('Ejecuto getInitialState');
    return {
      estado: 'inicializado en getInitialState'
```

```
    }
  },

  render: function() {
    console.log('Ejecuto render')
    return (
      <div>
        <p>Componente con propiedades y estado inicializado</p>
        <p>Estado: {this.state.estado}</p>
        <p>Propiedad: {this.props.propiedad}</p>
      </div>
    )
  }

})
```

Si usas este componente varias veces observarás gracias a los console.log que primero se inicializan las propiedades por defecto, una sola vez, y luego se inicializa el estado y se renderiza tantas veces como componentes hayas generado.

```
var anclaje = document.getElementById('anclaje')
ReactDOM.render(
  <div>
    <ComponenteInicializacion />
    <ComponenteInicializacion />
  </div>
  , anclaje)
```

Ahora te dejamos un archivo html completo donde podrás ver el componente en su declaración y funcionamiento, que podrás usar en un entorno vacío, sin necesidad de configurar transpiladores. Recuerda que esta alternativa no es la correcta para usar React en un entorno de producción, como se explicó en el artículo Comenzar con ReactJS desde un entorno de desarrollo vacío.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Inicializar componentes con createClass</title>
</head>
<body>
  <div id="anclaje"></div>

  <script src="https://unpkg.com/react@15.3.2/dist/react.js"></script>
  <script src="https://unpkg.com/react-dom@15.3.2/dist/react-dom.js"></script>
  <script src="https://unpkg.com/babel-core@5.8.38/browser.min.js"></script>

  <script type="text/babel">
    var ComponenteInicializacion = React.createClass({
      getDefaultProps: function() {
        console.log('Ejecuto getDefaultProps')
        return {
          propiedad: 'Valor por defecto de una propiedad'
        }
      },

      getInitialState: function() {
        console.log('Ejecuto getInitialState');
        return {
          estado: 'inicializado en getInitialState'
        }
      },
```

```
    render: function() {
      console.log('Ejecuto render')
      return (
        <div>
          <p>Componente con propiedades y estado inicializado</p>
          <p>Estado: {this.state.estado}</p>
          <p>Propiedad: {this.props.propiedad}</p>
        </div>
      )
    }

  })

  var anclaje = document.getElementById('anclaje')
  ReactDOM.render(
    <div>
      <ComponenteInicializacion />
      <ComponenteInicializacion />
    </div>
    , anclaje)

 </script>

</body>
</html>
```

Sirva entonces el anterior código solo para motivos de pruebas. Se podría ejecutar perfectamente haciendo doble clic sobre el archivo HTML, sin necesidad de un servidor web. Ten en cuenta que nuestro método preferido es el siguiente, en el que realizaremos lo mismo desde un entorno completo con Create React App y clases ES6.

Continuaremos este manual con el ciclo de vida de los componentes, algo cuyas primeras trazas ya hemos empezado a ilustrar con este artículo.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en *29/12/2016*
Disponible online en *http://desarrolloweb.com/articulos/propiedades-estado-predeterminado-componentes-react.html*

Desarrollo de componentes React en detalle

Para aprender React necesitamos conocer más detalles importantes sobre el desarrollo de componentes. Así que vamos a dedicarnos a ver detalles adicionales de herramientas que disponemos en React para definir componentes.

Ciclo de vida de los componentes React

Qué es el ciclo de vida y cómo implementar métodos dentro del ciclo de vida, para qué necesitamos usarlo.

El ciclo de vida no es más que una serie de estados por los cuales pasa todo componente a lo largo de su existencia. Esos estados tienen correspondencia en diversos métodos, que nosotros podemos implementar para realizar acciones cuando se van produciendo.

En React es fundamental el ciclo de vida, porque hay determinadas acciones que debemos necesariamente realizar en el momento correcto de ese ciclo. Ese es el motivo por el que hay que aprenderse muy bien cuáles son las distintas etapas por las que pasa la ejecución de un componente React. No es un tema trivial y tiene diversos matices. Veremos una introducción en este artículo y pondremos ejemplos.

Antes de comenzar cabe decir que esto es algo específico de los componentes con estado, ya que los componentes sin estado tienen apenas un método que se usará para renderizar el componente y React no controlará su ciclo de vida a través de los métodos que veremos a continuación. Montaje, actualización y desmontaje.

Estas son las tres clasificaciones de los estados dentro de un ciclo de vida del componente.

- El montaje se produce la primera vez que un componente va a generarse, incluyéndose en el DOM.
- La actualización se produce cuando el componente ya generado se está actualizando.

- El desmontaje se produce cuando el componente se elimina del DOM.

Además, dependiendo del estado actual de un componente y lo que está ocurriendo con él, se producirán grupos diferentes de etapas del ciclo de vida. En la siguiente imagen puedes ver un resumen de esta diferenciación. Observarás que en el primer renderizado de un componente se pasa por etapas del ciclo de vida diferentes de las que se pasa cuando se produce un cambio en sus propiedades o un cambio en el estado, o su desmontaje.

Primer renderizado	Cambios en las propiedades	Cambio en el estado	Componente se desmonta
getDefaultProps *	componentWillReceiveProps	shouldComponentUpdate	componentWillUnmount
getInitialState *	shouldComponentUpdate	componentWillUpdate	
componentWillMount	componentWillUpdate	render *	
render *	render *	componentDidUpdate	
componentDidMount	componentDidUpdate		

La secuencia de ejecución de las etapas del ciclo de vida se da de arriba hacia abajo. Los elementos marcados con un asterisco naranja no son métodos del ciclo de vida propiamente dicho, pero sí que son etapas de ejecución de componentes y se producen en la secuencia descrita en la imagen.

> Nota: En el Manual de React ya hemos visto todos los métodos marcados con el asterisco, por lo que deberían sonarte. Si no es así, simplemente revisa los artículos anteriores.

Descripción de los métodos del ciclo de vida

Ahora vamos a describir las distintas etapas del ciclo de vida de los componentes y luego podremos ver ejemplos.

> Nota: Puedes entender todos estos métodos como "hooks", ganchos que nos permiten ejecutar código en diferentes instantes. También los puedes ver como "manejadores de eventos", funciones con código a ejecutar cuando se producen diversas situaciones.

componentWillMount()

Este método del ciclo de vida es de tipo montaje. Se ejecuta justo antes del primer renderizado del componente. Si dentro de este método seteas el estado del componente con setState(), el primer renderizado mostrará ya el dato actualizado y sólo se renderizará una vez el componente.

componentDidMount()

Método de montaje, que solo se ejecuta en el lado del cliente. Se produce inmediatamente después del primer renderizado. Una vez se invoca este método ya están disponibles los elementos asociados al componente en el DOM. Si el elemento tiene otros componentes hijo, se tiene certeza que éstos se han inicializado también y han pasado por los correspondientes componentDidMount. Si se tiene que realizar llamadas Ajax, setIntervals, y cosas similares, éste es el sitio adecuado.

componentWillReceiveProps(nextProps)

Método de actualización que se invoca cuando las propiedades se van a actualizar, aunque no en el primer renderizado del componente, así que no se invocará antes de inicializar las propiedades por primera vez. Tiene como particularidad que recibe el valor futuro del objeto de propiedades que se va a tener. El valor anterior es el que está todavía en el componente, pues este método se invocará antes de que esos cambios se hayan producido.

shouldComponentUpdate(nextProps, nextState)

Es un método de actualización y tiene una particularidad especial con respecto a otros métodos del ciclo de vida, que consiste en que debe devolver un valor boleano. Si devuelve verdadero quiere decir que el componente se debe renderizar de nuevo y si recibe falso quiere decir que el componente no se debe de renderizar de nuevo. Se invocará tanto cuando se producen cambios de propiedades o cambios de estado y es una oportunidad de comprobar si esos cambios deberían producir una actualización en la vista del componente. Por defecto (si no se definen) devuelve siempre true, para que los componentes se actualicen ante cualquier cambio de propiedades o estado. El motivo de su existencia es la optimización, puesto que el proceso de renderizado tiene un coste y podemos evitarlo si realmente no es necesario de realizar. Este método recibe dos parámetros con el conjunto de propiedades y estado futuro.

componentWillUpdate(nextProps, nextState)

Este método de actualización se invocará justo antes de que el componente vaya a actualizar su vista. Es indicado para tareas de preparación de esa inminente renderización causada por una actualización de propiedades o estado.

componentDidUpdate(prevProps, prevState)

Método de actualización que se ejecuta justamente después de haberse producido la actualización del componente. En este paso los cambios ya están trasladados al DOM del navegador, así que podríamos operar con el DOM para hacer nuevos cambios. Como parámetros en este caso recibes el valor anterior de las propiedades y el estado.

componentWillUnmount()

Este es el único método de desmontado y se ejecuta en el momento que el componente se va a retirar del DOM. Este método es muy importante, porque es el momento en el que se debe realizar una limpieza de cualquier cosa que tuviese el componente y que no deba seguir

existiendo cuando se retire de la página. Por ejemplo, temporizadores o manejadores de eventos que se hayan generado sobre partes del navegador que no dependen de este componente.

Ejemplo de componente definiendo su ciclo de vida

Vamos a hacer un ejemplo muy básico sobre estos métodos, que simplemente lanza mensajes a la consola cuando ocurren cosas. El objetivo es que podamos apreciar el orden de ejecución de todos estos métodos del ciclo de vida.

El código de este componente con los métodos del ciclo de vida implementados lo encuentras a continuación:

```
import React from 'react'

export default class CicloVida extends React.Component {
  constructor(...args) {
    console.log('Ejecuto constructor', ...args)
    super(...args)
    this.state = {
      estado: 'Inicializado en el constructor'
    }
  }

  componentWillMount() {
    console.log('Se ejecuta componentWillMount')
  }

  componentDidMount() {
    console.log('Se ejecuta componentDidMount')
  }

  componentWillReceiveProps(nextProps) {
    console.log('Se ejecuta componentWillReceiveProps con las propiedades futuras', nextProps)
  }

  shouldComponentUpdate(nextProps, nextState) {
    console.log('Ejecutando shouldComponentUpdate. Próximas propiedades y estado: ', nextProps, nextState)
    // debo devolver un boleano
    return true
  }

  componentWillUpdate(nextProps, nextState) {
    console.log('Ejecutando componentWillUpdate. Próximas propiedades y estado: ', nextProps, nextState)
  }

  componentDidUpdate(prevProps, prevState) {
    console.log('Ejecutando componentDidUpdate. Anteriores propiedades y estado: ', prevProps, prevState)
  }

  componentWillUnmount() {
    console.log('Se desmonta el componente...')
  }

  render() {
    return (
      <div>
        <p>Componente del ciclo de vida</p>
        <p>Estado: {this.state.estado}</p>
        <p>Propiedad: {this.props.propiedad}</p>
      </div>
    )
  }
}
```

```
CicloVida.defaultProps = {
  propiedad: 'Valor por defecto definido para la propiedad'
}
```

Solo con que uses este componente tal cual podrás ver ya cómo se ejecutan los métodos del ciclo de vida relacionados con la parte del "primer renderizado" (mira la imagen anterior de este artículo).

Si quisieras poner en marcha otros métodos del ciclo de vida necesitarías hacer cosas como cambiar el estado del componente o sus propiedades. Como referencia te dejamos este otro código, en el que usamos el componente de CicloVida y cambiamos de manera dinámica una de sus propiedades.

```
import React from 'react'
import CicloVida from './index'

export default class UsarCicloVida extends React.Component {
  constructor(...args) {
    super(...args)
    this.state = {
      valorPropiedad: 'Test de valor de propiedad'
    }
  }

  cambiarPropiedad() {
    this.setState({
      valorPropiedad: 'Otro valor'
    })
  }

  render() {
    return (
      <div>
        <button onClick={this.cambiarPropiedad.bind(this)}>Cambiar propiedad</button>
        <CicloVida propiedad={this.state.valorPropiedad} />
      </div>
    )
  }
}
```

Ese código hace uso del componente anterior. Lo podemos analizar parcialmente, puesto que todavía hay cosas aquí que no hemos llegado a tocar. Pero lo básico en que fijarse es en la propiedad bindeada. El estado del componente "UsarCicloVida" tiene un dato que se ha bindeado como valor de la propiedad del componente "CicloVida". Por otra parte tenemos un botón que cambia el estado, y por consiguiente el cambio de estado se traduce en una actualización de la propiedad del componente CicloVida, lo que desencadenará en la ejecución de los métodos del lifecycle relacionados con el cambio del valor de una propiedad.

Seguramente que, si tienes alguna experiencia con el desarrollo de librerías o frameworks que usen binding, podrás entender el ejemplo. Pero, como decíamos, son cosas que tenemos que ver con calma en futuros artículos de este manual.

De momento lo dejamos por aquí. Tienes bastante material para experimentar. Haremos otros ejemplos del ciclo de vida más interesantes en futuros artículos.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en *17/01/2017*
Disponible online en *http://desarrolloweb.com/articulos/ciclo-vida-componentes-react.html*

Eventos en React

Cómo definir eventos de manera declarativa en React, cómo implementar los manejadores de eventos como métodos del componente y bindear el contexto.

Con React se construyen interfaces de usuario para la web y por supuesto, una de las partes fundamentales de éstas es la interacción con el visitante. Para poder responder a acciones realizadas sobre los componentes por parte de nuestros usuarios usamos los eventos.

El tratamiento de los eventos en Javascript en general es muy parecido a cómo se definen en React en particular, aunque hay un par de detalles importantes que debes saber y que vamos a tratar en el presente artículo del Manual de React.

Cómo definir un evento de manera declarativa en React

Los eventos en React se definen generalmente de manera declarativa, en el código de la vista o template, producido con JSX en el método render().

Para definirlo tenemos que indicar dos cosas: Primero el tipo de evento que queremos implementar y luego el método que hará las veces de manejador de evento.

Nota: un manejador de evento no es más que la función que se encarga de definir la funcionalidad que tiene que ejecutarse cuando se dispara un evento.

```
render() {
  return (
    <p>
      <button onClick={this.toggleSlider}>Mostrar / Ocultar Slider</button>
    </p>
  );
}
```

Como puedes apreciar en el código anterior, el método render devuelve el tempate, escrito con el lenguaje con JSX. Dentro del template tenemos un botón, el cuál tiene definido un evento "click".

> Nota: Es importante darse cuenta que el método asignado a onClick no tiene los paréntesis al final. No estamos queriendo ejecutar una función, sino indicar qué función se debe asociar como manejador de eventos. Por eso no se le colocan los paréntesis.

Las declaraciones de eventos en React tienen siempre esa forma: el prefijo "on", seguido del tipo de evento que queremos capturar (onClick, onInput...). Como valor, colocamos una referencia al método del componente encargado de procesar el evento correspondiente. Por tanto, al hacer clic sobre el botón, se ejecutará el método toggleSlider().

Luego tendremos que definir el manejador, como cualquier otro método del componente.

```
toggleSlider() {
    alert('Has hecho clic');
}
```

Bindear el contexto para acceder a this

Al implementar un manejador de evento es habitual que queramos acceder a las propiedades o métodos del propio componente. Por ejemplo, en el método toggleSlider() queremos cambiar el estado del componente y por tanto necesitamos manipularlo con this.setState().

Sin embargo aquí nos encontramos con una dificultad que viene del propio lenguaje Javascript no de React específicamente. En las funciones no es posible acceder a this como referencia al objeto sobre el que se invoca el método. Es por ello que necesitamos bindear el contexto.

Este sería el código real del método toggleSlider(), que simplemente tiene que negar una propiedad boleana del estado del componente.

```
toggleSlider() {
  this.setState(prevState => {
    return { showSlider: !prevState.showSlider };
  });
}
```

Como puedes ver, nos basamos en el estado anterior para calcular el nuevo estado. Por todas partes estamos usando "this" con la intención de acceder al propio componente que ha recibido el mensaje toggleSlider(). El problema aquí, como decíamos, es que "this" puede no contener esa referencia.

> Nota: puedes ver más sobre el tratamiento del estado y this.setState en el artículo del estado de componentes en react.

Para solucionarlo existen varios mecanismos en React. El más recomendable es binderar el contexto directamente en el constructor del componente.

Lo consigo simplemente así:

```
constructor(props) {
  super(props);
  this.state = {
    showSlider: false
  }
  this.toggleSlider = this.toggleSlider.bind(this);
}
```

En este constructor estamos haciendo diversas tareas, la primera es la inicialización del estado, pero lo que nos interesa a nosotros es la parte donde se hace `this.toggleSlider = this.toggleSlider.bind(this);`

> Nota: Gracias a la llamada a .bind(this) sobre una función o método, le estamos indicando a Javascript qué es lo que queremos que valga "this" dentro del método. Esto es algo que seguramente, si tienes experiencia en Javascript, ya habrás hecho en muchas otras ocasiones.

Otras alternativas de bindear this

También podemos hacer este bindeo de this (como contexto para en handler) a la hora de declarar el evento, en el template. El código nos quedaría más o menos así.

```
<button onClick={this.toggleSlider.bind(this)}>Mostrar / Ocultar Slider</button>
```

También podríamos obtener el mismo resultado usando una arrow function, que es capaz de mantener el contexto de this, con un código como este:

```
<button onClick={(e) => this.toggleSlider(e)}>Mostrar / Ocultar Slider</button>
```

Aunque estas opciones nos ensucien algo el template, tienen la ventaja de ahorrarnos algo de ceremonia en el constructor del componente. Sin embargo, en términos de rendimiento resultan una solución peor. El problema de estas alternativas es que, cada vez que se renderiza el componente, se le pasa una nueva función como manejador de evento, donde se tiene que bindear el contexto nuevamente. Incluso, si la función se está pasando hacia componentes hijos, forzaría un redibujado adicional del componente hijo, lo que posiblemente nos lleve a una disminución del rendimiento que podrá notarse en aplicaciones grandes.

Pasar parámetros a manejadores de eventos

En los métodos de React asociados como manejadores de eventos podemos pasar parámetros. Puede ser útil cuando quieres personalizar el comportamiento del manejador, mediante el paso de datos que tienes desde el template.

Sería algo tan sencillo como bindear los otros datos que deseas enviar al método que hace de manejador de eventos. De la siguiente manera:

```
<pre class="language-markup"><code>&lt;button onClick={(e) =&gt; this.toggleSlider(e)}&gt;Mostrar / Ocultar Slider&lt;/button&gt;</code></pre>
```

Ahora, dentro de nuestro método showText() podemos recibir el dato de esta forma.

```
showText(item, e) {
    console.log(e, item);
}
```

Esto nos mostrará dos datos en la consola, el "item" es aquella variable extra que estamos pasando al manejador. Por su parte, el parámetro "e" es el objeto evento de Javascript.

Prevenir el comportamiento por defecto de un evento

Algo también muy típico que querrás realizar en un componente al trabajar con eventos es prevenir el comportamiento predeterminado. Por ejemplo, ante un clic en un enlace, podemos evitar que el navegador se vaya a la URL definida en el href del enlace.

Para ello en Javascript nativo tendríamos dos opciones. La primera sería devolver "false" (return false) en el código del manejador. Esta alternativa no es válida para React. La segunda opción, que será la que realmente funcione, es invocar al método nativo de Javascript preventDefault(), que pertenece al objeto evento.

Por ejemplo, podríamos tener un enlace como este:

```
<a onClick={this.clickRealizado} href="http://escuela.it">Haz clic</a>
```

Cuyo manejador de evento sería el siguiente:

```
clickRealizado(e) {
    e.preventDefault();
    console.log('clickEnP')
}
```

Gracias a e.preventDefault() al hacer clic sobre el enlace no provocará que el navegador se vaya de la página, hacia la web de EscuelaIT.

Conclusión sobre los eventos en React

Hemos conocido lo básico sobre los eventos en React, junto con algunas cosas no tan básicas,

pero de necesario conocimiento para poder trabajar con esta librería.

Como has podido comprobar, trabajar con eventos no es nada complejo, pero requiere saber las particularidades de la librería React para evitar problemas o situaciones poco deseadas, que a veces resulta difícil de debuguear.

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en *28/02/2019*
Disponible online en *http://desarrolloweb.com/articulos/eventos-en-react.html*

Condicionales en templates JSX de React

Cómo definir expresiones condicionales en JSX, para que nuestros componentes React puedan cambiar su presentación dependiendo de sus propiedades o estado.

En este artículo del Manual de React vamos a explicar una de las necesidades habituales a la hora de escribir los templates de los componentes React, con JSX. Se trata de realizar expresiones condicionales, con las que podemos conseguir que varíe condicionalmente el aspecto de un componente, en función de una de sus propiedades o estado.

Para crear un condicional en un template de React podemos usar una expresión con el mismo código Javascript que usaríamos en la programación tradicional, aplicando el operador ternario.

Como sabemos ya, las expresiones en JSX se escriben entre llaves. Ahí podemos colocar código Javascript y aquel valor que resulte de la evaluación de la expresión es lo que aparecerá como contenido en el template.

La expresión condicional nos quedará de manera similar a esta:

```
{ this.state.showTitle ? <h1>Título</h1> : '' }
```

Para el procesamiento del condicional se evalúa un atributo del estado "showTitle". En caso positivo se muestra un heading H1 con un texto. En caso contrario, simplemente mostramos la cadena vacía, o sea, nada.

En este caso hemos usado poco código JSX para cada parte del condicional, pero podríamos tener unas cuantas etiquetas o componentes que mostrar para cada caso. No es un problema, puesto que podemos seguir usando la misma expresión con el operador ternario. Sólo entonces convendría organizar el código de una manera que resultase fácil para la lectura. Por ejemplo tal como se puede ver a continuación.

```
{ this.state.showTitle
  ? <div>
      <h1>Título</h1>
      <span> Más contenido...</span>
    </div>
  : <div className="sin-titulo">
      <b>Desconocido...</b>
      <FeedbackMessage message="No hemos recibido título" />
    </div>
}
```

Obviamente, como en nuestro ejemplo de código anterior estamos usando el estado para crear la expresión condicional, tendremos que haberlo inicializado, generalmente en el constructor.

```
constructor(props) {
  super(props);
  this.state = {
    showTitle: true
  }
}
```

Ejemplo de componente con un condicional

En este artículo no vamos a agregar mucho más conocimiento adicional sobre React, pero sí queremos aprovechar la ocasión para practicar un poco. La idea a continuación es realizar un componente completo que incorpore un condicional.

Vamos a practicar con los componentes funcionales sin estado, que ya vimos por primera vez en el artículo sobre El estado en React. Nuestro ejemplo en este caso es un sencillo "badge" (placa, mensaje pequeñito) que nos dice si algo está completado o no.

Para ello recibirá una propiedad boleana, donde indiquemos desde fuera si tiene que mostrarse como completado o incompleto. Nuestro componente lo podremos usar de dos maneras:

En el primer caso mostrará un badge con el texto "completado":

```
<CompletedBadge completed />
```

Si no le pasamos la propiedad completed al componente, entonces se entiende que debe mostrar el badge con el texto "incompleto".

```
<CompletedBadge />
```

El componente en sí tendrá este código:

```
import React from 'react';
import './CompletedBadge.css';

const CompletedBadge = (props) => {
  return props.completed
    ? <span className="Badge Badge-completed">Completado</span>
    : <span className="Badge Badge-incompleted">Incompleto</span>
}
export default CompletedBadge;
```

El código es bastante claro y, salvo el condicional que se acaba de explicar, no tenemos prácticamente nada nuevo que no se haya explicado antes en el Manual de React. Solo hay un detallito que quizás ha llamado tu atención.

Importar CSS

Queremos detenernos en el hecho de importar CSS, algo que podremos hacer o no dependiendo de la configuración de nuestro proyecto. El caso es que los componentes pueden importar su propio CSS. No hay problema en ello, siempre que tengamos un entorno de desarrollo que sepa lidiar con este tipo de imports. Si hemos iniciado nuestro proyecto con "create react app", tendremos la habilidad de importar CSS. En caso contrario deberíamos saber configurar Webpack para resolver este tipo de imports.

El importado de CSS lo realizamos en el componente en la línea:

```
import './CompletedBadge.css';
```

El código CSS que estamos importando en el archivo CompletedBadge.css es el siguiente:

```
.Badge {
  border: 1px solid #ddd;
  border-radius: 8px;
  font-size: 11px;
  padding: 4px;
}

.Badge-completed {
  color: green;
}

.Badge-incompleted {
  color: red;
}
```

Bindear un valor boleano a un componente o elemento HTML

Ese ejercicio nos lleva a otra necesidad común, que consiste en bindear un valor boleano a un elemento.

Por ejemplo, en el caso de nuestro componente con el template condicional, CompletedBadge,

el valor de la property que tenemos que pasar ("completed"), es un valor boleano. En HTML en general, los valores boleanos funcionan de tal modo que, si es verdadero se indica el valor y si es falso no se indica nada.

Para bindear un valor boleano al componente, simplemente tenemos que indicar una expresión que se evalúe a boleano.

```
<CompletedBadge completed={this.state.isCompleted} />
```

Si la expresión es positiva, entonces estaremos pasado un true. Si la expresión es negativa, React es lo suficientemente inteligente para no pasar ningún valor y producir que dentro del componente el valor de la property se evalúe como negativo.

Conclusión

Hemos resuelto una necesidad común en los templates de React, escritos en JSX, que permite componer ciertas partes de manera condicional, para colocar unos contenidos u otros dependiendo del valor de properties o estado.

Además hemos aprendido a cómo enviar datos con valores boleanos, desde los hijos a los padres, para setear propiedades que vamos a usar para las expresiones condicionales.

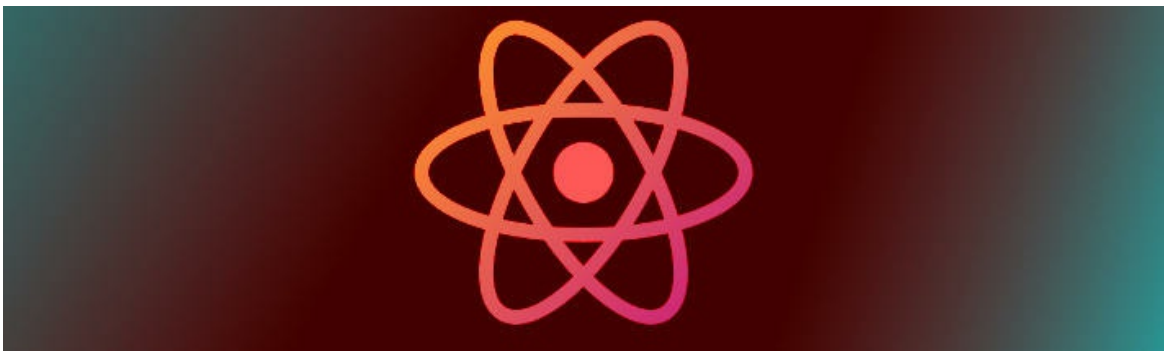
Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en *21/03/2019*
Disponible online en *http://desarrolloweb.com/articulos/condicionales-templates-jsx-react.html*

Crear repeticiones en templates JSX con React

En este artículo vamos a mostrar cómo se deben crear repeticiones, para mostrar listados de ítems, en templates creados con JSX para componentes React.

En este artículo del Manual de React, después de haber abordado las estructuras condicionales en JSX, nos vamos a dedicar a otro importante tipo de estructura que podemos usar a la hora de construir templates: la creación de repeticiones. Este es un recurso que usarás cada vez que tengas que mostrar listados de ítems en una vista, a partir de datos alojados en un array.

La tarea es bastante sencilla y se realiza mayormente usando una herramienta del propio lenguaje Javascript, el método map() disponible en los Arrays. Este método sirve para producir un nuevo array a partir de una repetición por cada uno de los elementos del array inicial.



Un uso de map() lo podemos ver a continuación.

```
let miArray = [1, 2, 4, 8];
let transformado = miArray.map(item => item * 2);
console.log(transformado);
```

Este código produce una salida en la consola del array [2, 4, 8, 16].

Supongo que a estas alturas ya estarás familiarizado con las arrow functions, pero por si acaso, este código sería equivalente (lo remarco porque vamos a usar bastantes arrow functions en este artículo).

```
let transformado = miArray.map(function(item) {
  return item * 2;
});
```

> Nota: si tienes dudas con esto, por favor, lee el artículo sobre las arrow functions de ES6.

Sabiendo usar el método map del array tienes bastante hecho por delante para aprender a producir tus primeras repeticiones en JSX. Sin embargo, hay un detalle que lo complica un poquito más y nos hará alargarnos algo, que es el uso de llaves. Pero vamos poco a poco comenzando por una repetición simple.

Crear una repetición básica en JSX

Comencemos por crear una repetición sencilla, sin preocuparnos todavía de las llaves. Para ello vamos a necesitar un componente que tenga una propiedad o estado con valor de array. En el método render() realizaremos la repetición en ese array, creando un elemento de lista para cada ítem del array.

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

class RepeatComponent extends Component {
  constructor(props) {
    super(props);
    this.state = {
      lenguajes: ['Javascript', 'JSX', 'Typescript', 'NodeJS']
```

```
    }
  }
  render() {
    return (
      <ul>
        {this.state.lenguajes.map(item => <li>{item}</li>)}
      </ul>
    );
  }
}

export default RepeatComponent;
```

El código de la repetición queda bastante compacto aunque, si no estás acostumbrado a ver este tipo de construcciones, quizás puede chocar algo al principio.

Para entenderlo piensa que el método map está devolviendo un array de "sub-templates". Para cada ítem genera un nuevo sub-template y lo almacena en un array. Luego, el template principal simplemente hace que se renderice ese array de sub-templates.

Este sería un código equivalente, sin la arrow function, por si sirve de ayuda.

```
render() {
  return (
    <ul>
      {this.state.lenguajes.map(function(item) {
        return <li>{item}</li>
      })}
    </ul>
  );
}
```

Incluso hay personas que prefieren componer unos templates a partir de llamadas a otras funciones dentro del componente.

```
languageItems(items) {
  return items.map((item) => {
    return <li>{item}</li>
  });
}

render() {
  return (
    <ul>
      {this.languageItems(this.state.lenguajes)}
    </ul>
  );
}
```

> Nota: Como sabes, a la hora de organizar el código, lo ideal es que lo hagas de la manera que te parezca más claro, para facilitarte el mantenimiento del código más adelante. Personalmente yo prefiero ver la repetición en el mismo código JSX, antes de generar un método extra que me obliga a entender el template en varios pasos. Sin embargo, si lo que vamos a repetir es muy complejo y consta de mucha cantidad de marcado quizás pueda tener sentido usar un método o función extra. No obstante, en este caso quizás te interesa "componetizar" y crear un nuevo componente que encapsule y te abstraiga de todo lo complejo que sea el marcado a repetir. Luego veremos un ejemplo con esto.

Usar llaves (keys) para los ítem de la repetición

El código del componente anterior realiza la repetición en el template correctamente. Al ejecutarse obtendremos un listado de cada uno de sus ítem en la vista, sin embargo, podremos apreciar que aparece un warning en la consola: *(Warning: Each child in a list should have a unique "key" prop)*.

Este mensaje te sale porque en React se deben crear una especie de llaves primarias para cada uno de los ítem de la repetición. Como en el código anterior no los hemos expresado, se nos advierte desde la consola para corregir el problema. Aunque, en realidad no se trata de un problema, en principio, porque React genera unas keys predeterminadas, basadas en el número del índice de cada elemento del array y las usa en caso que no se hayan indicado manualmente.

Sin embargo, las llaves generadas a partir de los índices no son muy seguras. Para comenzar, si ordenamos el array dinámicamente de distintos modos los índices cambiarán y el componente podría comenzar a funcionar de manera errática. Incluso, aún en el caso que no se necesite ordenar el array dinámicamente, podemos encontrarnos ante varias desventajas, como una caída del rendimiento.

Puedes encontrar más información sobre cómo afecta el uso de llaves índice a las repeticiones en la documentación de React: Lists and Keys.

Sea como fuere, lo ideal es siempre indicar nuestros propios índices. Para ello podemos seguir varias estratégias:

Usar los índices de los ítem de la base de datos

Lo más normal es que los datos a listar te lleguen desde una base de datos. En este caso, podrías usar las propias claves primarias de los elementos como índices en el listado.

Por ejemplo, para un array de esta forma:

```
[
  {
    id: 1,
    title: 'Primer mensaje',
    content: 'Este es el contenido del primer mensaje'
  },
  {
    id: 2,
    title: 'Segundo mensaje',
    content: 'Otro mensaje para listarlo con React'
  },
]
```

Podríamos hacer una repetición con este código JSX:

```
<div>
  {this.state.msgs.map((item) =>
    <article key={item.id}>
      <h3>{item.title}</h3>
      <p>{item.content}</p>
    </article>
  )}
</div>
```

Como puedes ver, en el tag ARTICLE estamos especificando la key, asignando el valor de cada identificador de mensaje. Es tan sencillo como esto.

Qué hacer si no tienes un identificador único

Si no tienes llaves primarias o identificadores en cada uno de tus ítem, podrías usar el propio index del array, que también te entregan como parámetro en la función que envías al método map(). Sería de esta forma.

```
{this.msgs.map((item, index) =>
  <article key={index}>
    <h3>{item.title}</h3>
    <p>{item.content}</p>
  </article>
);
```

Sin embargo, aunque esto te ahorre ver el warning en la consola, estaremos ante el mismo problema que no poner nada, pues este índice es el que React crea automáticamente para ti, que tiene los mencionados problemas de rendimiento o hasta funcionamiento.

Componetizar el JSX de cada repetición

"Componetizar" es uno de nuestros verbos preferidos a la hora de desarrollar con arquitectura de componentes. Básicamente nos referimos a que, cuando tenemos algo suficientemente complejo, es mejor separarlo en un componente, para abstraernos de esa complejidad y de paso poder reutilizar el componente más adelante si fuera necesario.

En el caso de la repetición sobre el array de comentarios, el JSX que se va a usar para cada comentario es suficientemente complejo como para que creemos un componente "tonto" (sin estado) que nos aporte una mayor claridad en la repetición.

Este es el código que podríamos usar para crear este componente sin estado:

```
import React from 'react';

const CommentComponent = (props) => {
  return (
    <article>
      <h3>{props.comment.title}</h3>
      <p>{props.comment.content}</p>
    </article>
  );
}
```

```
export default CommentComponent;
```

> Nota: en el anterior código estamos usando lo que se conoce como "stateless function component", que ya explicamos en el artículo sobre el estado de los componentes.

Ahora, para hacer la repetición y mostrar todos los comentarios que tenemos en un array, usamos este mismo componente, lo que nos simplificará bastante el código del componente principal.

```
render() {
  return (
    <div>
      {this.state.msgs.map((item) => <CommentComponent key={item.id} comment={item} />)}
    </div>
  );
}
```

Conclusión sobre las repeticiones para listados en JSX

Bien, hemos aprendido algo tan importante en el desarrollo de componentes, como la realización de un listado a partir de los elementos de un array. Para producir esta repetición usamos el método map(), con el que conseguimos generar un array con cada uno de los templates a representar para cada ítem.

La cosa sería tan simple como lo que acabamos de ver, si no fuera porque en React necesitamos generar unas llaves únicas para cada elemento de la repetición. Esas llaves las podemos recibir desde el backend, con lo que usarlas sería algo directo. Sin embargo, si no las tenemos, podemos usar cualquier otro recurso a nuestro alcance.

Las llaves no tienen por qué ser numéricas, simplemente tienen que ser únicas para que los mecanismos internos de React funcionen correctamente,

Este artículo es obra de *Miguel Angel Alvarez*
Fue publicado / actualizado en *04/04/2019*
Disponible online en *http://desarrolloweb.com/articulos/repeticiones-templates-jsx-react.html*