JSX

Tiempo estimado: 20min

Las aplicaciones webs, independientemente del *framework* que estemos usando, tienen HTML como una parte fundamental de su código. En el caso de React, es muy fácil crearlo en código JavaScript mediante el uso de JSX. Aunque JSX no es necesario, se recomienda su uso porque hace el trabajo más fácil y nos hace más productivos.

La lección comienza introduciendo JSX, un preprocesador de JavaScript que permite la añadidura de código XML, lo que ayuda a definir y añadir de manera muy sencilla código HTML desde código JavaScript. A continuación, se explican detalladamente los literales XML. Los elementos con los que añadir HTML a nuestra aplicación React. Después, se presenta el procesamiento especial que hace React de los atributos HTML y de las entidades. Finalmente, se muestra cómo generar código HTML de manera dinámica.

Al finalizar la lección, el estudiante sabrá:

- Qué es JSX.
- Cómo usar JSX.
- Cómo añadir elementos HTML dinámicamente a una aplicación React.

Introducción

JSX (*JavaScript XML*) es un preprocesador de JavaScript. Añade soporte nativo de XML en los archivos de JavaScript, facilitando así su uso. Aunque React se puede utilizar sin JSX, es raro hacerlo porque con JSX el código es más claro. Es más, por convenio y buenas prácticas, se recomienda encarecidamente su utilización.

He aquí un ejemplo para ir abriendo boca:

```
var title = <title>Mi primera aplicación React</title>;
```

Es importante tener claro que el código HTML que aparece en los archivos JSX tiene que encontrarse en formato XML. El *transpiler* los convertirá a HTML5. Así pues, el elemento HTML
 debe aparecer en código JSX como
.

Babel

Babel es el *transpiler* oficial de React y permite convertir código desarrollado con JSX a JavaScript puro para que pueda interpretarse en cualquier motor JavaScript como, por ejemplo, el de los navegadores.

En nuestro caso, usamos Browserify para generar el archivo empaquetado con el código JavaScript de la aplicación React. Recordemos que esta herramienta no entiende más que JavaScript. Por lo que habrá que convertir el código JSX a JavaScript puro. Para ello, usaremos un transformador, recordemos, un componente de Browserify que convierte un archivo en otro antes de su procesamiento por parte de Browserify. Este transformador es babelify que usa el *transpiler* Babel.

A su vez, Babel debe importar los *presets* react y es2015, que deben encontrarse instalados localmente como dependencias de desarrollo del proyecto, o sea, en la propiedad devDependencies del archivo package.json.

He aquí un ejemplo de uso:

```
browserify app/index.jsx --extension=.jsx -o dist/scripts/react-app.js -t [ babelify --presets
[ es2015 react ] ]
```

Para ayudar en el desarrollo, el generador de React de Justo registra una tarea en el catálogo del proyecto que hace esto por nosotros con una llamada tan simple como:

```
justo build
```


Archivos .jsx

Generalmente, los archivos con código JSX se nombran con la extensión .jsx. Aunque también se puede usar .js.

Literales XML

Recordemos que la idea que se esconde detrás de JSX es poder crear fácilmente código HTML y XML desde código JavaScript. Y esto se consigue principalmente mediante literales XML. Recordemos de JavaScript que un valor literal (*literal value*) o simplemente literal es una unidad léxica que denota algo tal cual como, por ejemplo, un número o una cadena de texto. Para poder expresar valores literales de tipo String, JavaScript define su propia sintaxis: expresamos el literal mediante una secuencia de cero, uno o más caracteres delimitados todos ellos por comillas simples (') o dobles ("). También podemos expresar valores numéricos literales como 123 ó 123.456. O valores booleanos como true o false. O literales de tipo *array* como [1, 2, 3] o de tipo objeto como {x: 1, y: 2}.

Veamos pues cómo añadir valores literales de tipo XML al código JavaScript mediante JSX. Es tan simple como añadir un elemento XML, tal cual. Por ejemplo:

```
var tit = <title>Mi primera aplicación React</title>;
```

El ejemplo anterior se traducirá a:

```
var tit = React.createElement("title", null, "Mi primera aplicación React");
```

Es mucho más fácil y claro el uso de JSX.

Plantillas

En el fondo, un literal XML es como una cadena plantilla de JavaScript. Recordemos, aquellas que se delimitan mediante comillas invertidas (`). Con la diferencia de que en vez de definir valores de tipo String, lo hacen de tipo JSXElement.

Dentro de las cadenas plantillas, podemos escribir expresiones JavaScript que serán evaluadas y su valor se insertará en la posición de la plantilla en la que se encuentra. En el caso de los literales XML pasa lo mismo, podemos indicar expresiones JSX que se evaluarán y reemplazarán su contenido. Pero a diferencia de las plantillas literales, donde las expresiones se delimitan por ${ y }, en los literales XML se delimitan por { y }. Veamos un ejemplo ilustrativo:

```
var btn = <input type="button" disabled={!isLoggedIn()} />;
```

Cuando el valor de un atributo de un elemento proceda de la evaluación de una expresión JavaScript, omita las comillas y use una expresión delimitada por { y }, tal como se muestra en el ejemplo anterior.

Si el valor de un atributo consiste en un valor literal de tipo objeto, no hay que olvidar que éstos también se delimitan por { y }. Por lo que tendremos que usar un doble {{ y }}. El primero delimita la expresión JavaScript en el literal XML y el segundo el literal objeto. He aquí un ejemplo ilustrativo:

```
<div style={{color: "white", backgroundImage="url(/images/background.png)"}}>
  Hello world!
</div>
```

Atributos

Cada vez que React se encuentra con un literal XML cuyo nombre es un elemento de la especificación HTML como, por ejemplo, table, td, tr, lo, ul, h1, etc., React lo procesa de manera particular. Los atributos de estos elementos requieren una especial atención por nuestra parte.

Atributos HTML

En React, cuando se define un elemento HTML mediante un literal XML de JSX, los atributos tienen los mismos nombres, pero deben expresarse en notación *camelCase*. Así pues, el atributo background-image debe definirse en JSX como backgroundImage. El atributo background-color como backgroundColor. El controlador de eventos onchange como onChange. Observe que se omite el guión separador de palabras (-) y se concatenan todas ellas, indicándose la primera en mayúscula, salvo la primera palabra que siempre será minúscula.

Para conocer los elementos HTML soportados por React y los nombres de los atributos que debemos

usar, se puede consultar la página Tags and Attributes, facebook.github.io/react/docs/tags-and-attributes.html, del sitio oficial de React.

A continuación, se enumera algunos atributos que suelen llevar a error cuando se usan en JSX porque llevan consigo un cambio de nombre a *camelCase* sin llevar un guión (-) en su nombre:

Atributo HTML	Atributo JSX
autocomplete	autoComplete
autofocus	autoFocus
autoplay	autoPlay
cellpadding	cellPadding
cellspacing	cellSpacing
charset	charSet
colspan	colSpan
datetime	dateTime
enctype	encType
formaction	formAction
formenctype	formEncType
formmethod	formMethod
formnovalidate	formNoValidate
formtarget	formTarget
hreflang	hrefLang
maxlength	maxLength
minlength	minLength
novalidate	noValidate
radiogroup	radioGroup
readonly	readOnly
rowspan	rowSpan
tabindex	tabIndex
usemap	useMap

Atributos con nombres reservados

Algunos atributos HTML son palabras reservadas de JavaScript y, por ende, de JSX. Por lo que hay que utilizar nombres sustitutos que no lo sean:

Atributo HTML	Atributo JSX	Descripción
class	className	Define la clase del elemento.
for	htmlFor	Indica el elemento al cual asociar un elemento <label>.

Así, por ejemplo, es la manera correcta de escribir en JSX el elemento HTML . Es muy común olvidarlo. Pero a medida que lo olvide y React se queje, se irá acostumbrando.

Atributo style

Por buenas prácticas, se recomienda aplicar estilo mediante el atributo className, recordemos el equivalente al class de HTML, en vez de hacerlo mediante el atributo style que define un estilo en línea para el elemento. Pero si decidimos saltarnos este convenio, hay que recordar que en React este atributo espera un objeto JavaScript, no una cadena. Así pues, si en HTML haríamos algo como:


```
<div style="color:white;background-color:black">
```
En JSX, tendremos que hacer:
```
<div style={{color: "white", backgroundColor: "black"}}>
```
O bien:
```
var estilo = {color: "white", backgroundColor: "black"};
<div style={estilo}>
```

Propagación de atributos

La propagación de atributos (*attribute spread*) es una operación mediante la cual las propiedades de un objeto JavaScript se añaden como atributos a un elemento XML de JSX. Se utiliza el operador ... de ES6:
```
...objeto
```
A continuación, se muestra un ejemplo que añade al elemento input todas las propiedades del objeto props como atributos:
```
var props = {id: "save", disabled: true};
var save = <input type="button" {...props} />;
```
Lo anterior es lo mismo que:
```
var save = <input type="button" id="save" disabled="true" />;
```
Si a continuación del operador de propagación se define un atributo propagado, su valor tendrá prioridad. Es decir, en caso de que el mismo atributo se defina varias veces, la última definición sobrescribirá las anteriores.

Atributos personalizados

En React, si se define un atributo no definido en la especificación HTML para un elemento HTML, simplemente no se representará en el DOM del documento. Se omitirá. Por ejemplo, el atributo abc no existe. Si lo definimos, por ejemplo, en un elemento como sigue:
```
<div className="container" abc="valor">
```
React simplemente lo omitirá y representará el elemento como:
```
<div class="container">
```
Cuando es necesario que este tipo de atributos se represente, es decir, formen parte del elemento HTML final que se añadirá al documento, debemos preceder su nombre con data-. Ejemplo:
```
<div className="container" data-abc="valor">
```

Entidades

En HTML, una entidad (*entity*) es una secuencia de caracteres especial que representa uno o más caracteres. Por ejemplo, & representa al carácter &; <, <; etc.

En JSX, cuando necesitamos indicar una entidad podemos hacerlo sin problemas, pero tenemos que tener clara una cosa. Cuando utilizamos una expresión JSX en un literal XML, recordemos aquello que delimitamos por { y }, si su valor pasa a formar parte del contenido de un elemento, cualquier carácter que tenga asociada una entidad, será reemplazado por su entidad. Por ejemplo, si la cadena devuelta por la expresión contiene el carácter <, entonces el motor convertirá automáticamente < a <. En cambio, si no lo hacemos mediante una expresión, sino directamente en el literal, hay que indicar la entidad. Veámoslo mediante unos ejemplos:
```
var title = <title>La entidad &lt;</title>;
var title = <title>{'La entidad <'}</title>;
```

Atributo dangerouslySetInnerHTML

En algunas ocasiones, el contenido de un elemento HTML procede del valor de una variable. Tal como acabamos de ver, si la variable contiene Esto es un ejemplo , React lo representará como <p>Esto es un ejemplo</p>. ¿Esto significa que no es posible añadir código HTML desde una variable? No, sí es posible, pero no se recomienda, aunque generalmente todos lo hacemos.

Para añadir el contenido de una variable como código HTML, hay que hacer un pequeño trabajo extra. En primera instancia, debemos tener claro que hay que hacerlo mediante un atributo especial, dangerouslySetInnerHTML. Veamos un ejemplo, antes de continuar:


```
<p dangerouslySetInnerHTML={{__html: marked(props.question)}} />
```
marked es un paquete, disponible en NPM, que permite generar código HTML a partir de un fragmento de texto en formato Markdown. El ejemplo anterior muestra cómo añadir el código HTML generado por la función marked() como contenido de un párrafo. Observe que no se hace lo siguiente:

```
<p>{marked(props.question)}</p>
```
Lo anterior escaparía el contenido. Nosotros no queremos que lo escape, deseamos que lo añada como contenido de <p> como HTML puro y duro. En React, esto se consigue mediante el atributo dangerouslySetInnerHTML. Observe que este atributo espera un objeto JavaScript donde su propiedad __html es el código HTML que debe añadirse como contenido del elemento.

Generación dinámica de un elemento

A la hora de definir un elemento, podemos definirlo estática o dinámicamente. Cuando conocemos de antemano su estructura exacta, lo normal es hacerlo estáticamente. Consiste en definirlo como un literal XML tal cual. Ejemplo:

```
<table>
  <thead>
    <tr>
      <th>Ciudad</th>
      <th>Población (año)</th>
    </tr>
  </thead>

  <tbody>
    <tr>
      <td>Valencia</td>
      <td>786.189 (2015)</td>
    </tr>

    <tr>
      <td>Roma</td>
      <td>2.874.038 (2014)</td>
    </tr>

    <tr>
      <td>Londres</td>
      <td>8.630.000 (2015)</td>
    </tr>
  </tbody>
</table>
```
Pero por suerte o desgracia, no siempre es posible conocer exactamente cómo será el elemento. En estos casos, podemos añadir parte de su contenido dinámicamente, por ejemplo, mediante expresiones delimitadas por { y }. Ejemplo:

```
<table>
  <thead>
    <tr>
      <th>Ciudad</th>
      <th>Población (año)</th>
    </tr>
  </thead>

  <tbody>
    {
      ciudades.map(function(c) {
        return (
          <tr key={c.nombre}>
            <td>{c.nombre}</td>
            <td>{c.población.habitantes} ({c.población.año})</td>
          </tr>
        );
      })
    }
  </tbody>
</table>
```


El método map() de los *arrays* es muy útil y utilizado cuando se va a generar contenido a partir de los elementos de un *array*. Recordemos que este método ejecuta una función, conocida como función de transformación (*transform function*) y pasada como argumento, con cada elemento del *array*. Al finalizar el recorrido, map() devuelve *otro array* con los valores devueltos por la función de transformación para cada elemento.

Cuando React recibe un *array* como valor de una expresión, lo que hace es añadir cada uno de sus elementos como si se hubieran definido uno detrás de otro.

Otra posibilidad de hacer lo anterior sería como sigue, para gustos colores:

```
const ciudades = […];
const filas = ciudades.map(function(c) {
  return(
    <tr key={c.nombre}>...</tr>
  );
});

<table>
  <thead>
    <tr>
      <th>Ciudad</th>
      <th>Población (año)</th>
    </tr>
  </thead>

  <tbody>
    {filas}
  </tbody>
</table>
```

Atributo key

React es un poquito quisquilloso o puntilloso. Cuando una expresión devuelve un *array* de elementos XML, como en el ejemplo anterior, cada uno de ellos debe tener un atributo key con un identificador único para cada uno de ellos. En el ejemplo anterior, hemos considerado que ninguna ciudad puede tener el mismo nombre, aunque esto sí es posible, y lo hemos usado como valor clave.

Si no añadimos este atributo, React mostrará un mensaje de error en la consola JavaScript del navegador. Aunque la aplicación funcionará inicialmente sin problemas.

