Tras dos capítulos aprendiendo sobre Componentes web, ECMAScript y Babel, ya es el momento de escribir nuestro primer componente en React. Hasta ahora todos los ejemplos los hemos ejecutado en la REPL de Babel. Ya no es suficiente, pues necesitamos ejecutar nuestro código y ver el HTML generado. Por ello, a partir de ahora tendréis todos los ejemplos en la colección Curso React en Codepen.

Sin más, aquí tenéis el código de nuestros primeros componentes de React:

```
// Nuestro primer componente
const firstComponent = <h1>React rocks!</h1>;

// También podemos definir los componentes como una función
const secondComponent = () => <h2>React rocks!</h2>;

// O como una clase
class ThirdComponent extends React.Component {
  // Renderizamos un título
  render() {
    return <h3>React Rocks!</h3>;
  }
}

// "Montamos" todos nuestros componentes en el DOM
ReactDOM.render(
  <div>
    { firstComponent }
    { secondComponent() }
    <ThirdComponent />
  </div>,
  document.getElementById('app')
);
```

Código en Codepen.

En React, un componente es toda clase, método o variable que devuelva *React Elements*. Un *React Element* es una función que devuelve un objeto propio de React. Estos objetos nos permiten renderizar nuestra vista. Entraremos más en detalle en el siguiente apartado.

Otra forma de crear un componente es mediante el método `React.createClass(/* ... */)`. Este no requiere de ES2015 al invocarse como una función. No obstante, la sintaxis es más compleja y la mayoría de documentación existente utiliza clases. Por ello, en este curso no se va a tratar más el método `React.createClass`.

Quedan aún dos partes del código que no acaban de encajar. ¿Por qué hay etiquetas HTML en código ECMAScript? ¿Qué es `ReactDOM`?

JSX

Analizemos la siguiente línea del ejemplo anterior:

```
const firstComponent = <h1>React rocks!</h1>;
```

Como hemos visto, los componentes el React devuelven *React Elements*. Esta sintaxis que mezcla “etiquetas HTML” y código JavaScript es JSX. Cuando compilamos este código con Babel (configurado con su preset de React) obtenemos lo siguiente:

```
"use strict";

var firstComponent = React.createElement(
  "h1",
  null,
  "React rocks!"
);
```

Código en Babel REPL

JSX es una sintaxis para definir la interfaz de usuario de nuestra aplicación en JavaScript. Tras compilar el código, esta sintaxis se transforma en llamadas al método `React.createElement` que devuelven un objeto.

Expresiones JavaScript

JSX nos permite introducir expresiones de JavaScript en su sintaxis. De esta manera podemos realizar llamadas a funciones o interpolar variables.

```
let car = {
  name: 'coche',
  attribute: 'rojo'
}

const buildString = obj => `Mi ${obj.name} es ${obj.attribute}`;

const element = <p>
  Object: { car.name } => { buildString(car) }.
</p>;
```

Código en Codepen;

Atributos

Las etiquetas de HTML tienen atributos, como `src` o `href`. Estos atributos también se pueden definir en JSX:

```
let text = <p id="myText">Este párrafo tiene un ID.</p>;
```

También podemos utilizar variables y funciones como valor de los atributos utilizando `{}`.

```
const twitterUrl = (user) => `https://twitter.com/${user}`;
let twitterLink = <p>Mi twitter es <a href={ twitterUrl('laux_es') }>@laux_es</a></p>;
```

Las clases de CSS son un atributo especial en JSX. Al ser `class` una palabra reservada en JavaScript, no podemos utilizarla como atributo en JSX. Para definir las clases de CSS utilizamos el atributo `className`:

```
const customClass = () => 'redText';

let redText = <p className={ customClass() }>Este texto es rojo</p>;
```

style también es otro caso especial. Los estilos se definen en JSX como objetos planos de JavaScript utilizando el formato *camelCase* para los nombres de los atributos.

```
const customStyle = {
  fontSize: '4em'
};

let bigText = <p style={ customStyle }>Este texto es grande</p>;
```

En la documentación de React tenéis todos los atributos HTML disponibles.

Código en Codepen.

Nodos hijos en JSX

Normalmente, los componentes de nuestras aplicaciones renderizarán más de un nodo HTML. Por ejemplo, para crear un componente `text` necesitaremos una cabecera y un párrafo. **En JSX podemos incluir más de un nodo hijo siempre que todos pertenezcan al mismo padre**. Es decir, cada componente renderiza un nodo padre

con todos los nodos hijos que necesitemos. Si intentamos devolver varios nodos sin un padre común el compilador devolverá un error de sintaxis.

```
let invalidText = <h1>Cabecera</h1><p>Párrafo</p>;
```

Para que este componente sea válido, englobamos ambos elementos con un nodo padre:

```
let text = <div>
  <h1>Cabecera</h1>
  <p>Párrafo</p>
</div>;
```

Código en Codepen.

Iteración de elementos

Un problema muy común a la hora de desarrollar una interfaz es la posibilidad de iterar sobre un array de elementos y renderizarlos. En JSX, solventamos este caso incluyendo una expresión de JavaScript que itera sobre el array y retorna los nuevos *React Elements*.

```
const TransportList = () => {
  // Estos son los elementos que vamos a dibujar en la interfaz
  let elements = ['car', 'plane', 'train'];

  return <ul>
    { elements.map(element => <li key={ element }>{ element }</li>) }
  </ul>
}
```

Código con Codepen.

Cuando iteramos sobre un array, **es necesario que todos los elementos que rendericemos incluyan un atributo único llamado** `key` . React utiliza este parámetro para controlar los distintos componentes que está renderizando y poder actualizarlos o borrarlos posteriormente.

ReactDOM

Todos los ejemplos que hemos visto incluyen al final unas líneas de código que referencian a una librería llamada ReactDOM.

```
// "Montamos" todos nuestros componentes en el DOM
```

```
// Montamos todos nuestros componentes en el DOM
ReactDOM.render(
  /* Nuestro componente */,
  document.getElementById('app')
);
```

Esta librería es la que se encarga de conectar nuestro componentes de React con la interfaz DOM del navegador*. Cómo vimos al comienzo del curso, una de las grandes ventajas de React es que separa la lógica de procesamiento y definición de componentes de la de renderización. Es por ello, que para visualizar nuestros componentes necesitamos una librería que los represente en la plataforma que estemos utilizando.

Existen numerosas librerías que representan nuestros componentes en otras plataformas como ReactNative en móviles o ReactCanvas en elementos `<canvas>` de HTML.