INTRODUCCIÓN A REACT

React es una librería de Javascript que sirve para construir interfaces de usuario y aplicaciones front-end. Permite crear interfaces de usuario complejas compuestas de pequeñas piezas de código llamadas componentes.

Es una librería de software libre que sirve para desarrollar aplicaciones web de una forma más ordenada y con menos código que si usamos Javascript puro o librerías como JQuery centradas en la manipulación del DOM. Permite que las vistas se asocien con los datos, de manera que si cambian los datos también cambian las vistas.

React introduce el concepto de **Virtual DOM**, que es una de sus principales características. Consiste en una representación del DOM en memoria que se usa para aumentar el rendimiento de los componentes y aplicaciones front-end. Básicamente lo que hace es que, cuando se actualiza una vista, React se encarga de actualizar el DOM Virtual, que es mucho más rápido que actualizar el DOM del navegador (DOM real). Al comparar ambos, React sabe qué partes de la página se deben actualizar y se ahorra la necesidad de actualizar la vista entera.

PRIMEROS PASOS

La mejor manera de empezar a trabajar con React es usando el paquete create-react-app. Permite empezar de manera rápida sin preocuparse de la configuración inicial de un proyecto.

Para crear una aplicación debemos tener instalado el paquete *npm*. Una vez instalado nos colocamos en la carpeta de nuestros proyectos y lanzamos el comando para comenzar una nueva aplicación:

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

Mediante el anterior comando se cargarán los archivos de un proyecto vacío y todas las dependencias de *npm* necesarias para el funcionamiento del proyecto. Una vez terminado el proceso, que puede tardar un poco, podremos entrar en la carpeta de nuestra app:

```
cd mi-app/
```

Y una vez dentro podemos empezar a ejecutarla con el siguiente comando:

```
npm start
```

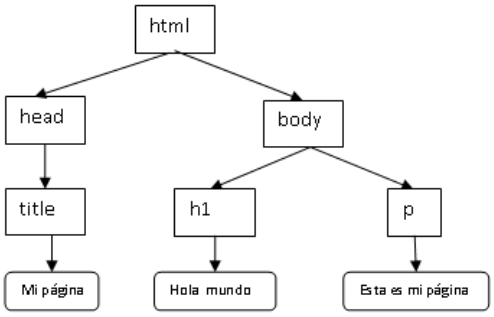
CONCEPTOS PREVIOS

- **El Modelo de Objetos del Documento (DOM)**

La W3C define el DOM o "Modelo de Objetos del Documento" como una interfaz de programación de aplicaciones (API) para documentos HTML y XML.

```
<html>
<head>
  <title>mi página</title>
</head>
<body>
  <h1>Hola mundo</h1>
        <p>Esta es mi página.</p>
</body>
</html>
```

En el DOM se representa como una estructura de arbol de la siguiente manera:



El DOM es la representación de la interfaz gráfica de nuestra aplicación.

- **Programación declarativa**

React utiliza una programación declarativa centrado en los componentes de la interfaz gráfica. A diferencia de la programación imperativa, en la que hay que explicar paso a paso cómo se quieren hacer las cosas, en la programación declarativa simplemente hay que indicar qué es lo que se quiere hacer.

En lenguajes como Javascript o Jquery, al ser imperativos, los desarrolladores tienen completa libertad a la hora de crear el código, existiendo el problema de que con programadores inexpertos se termina escribiendo programas muy complejos (código spaghetti). En cambio, usando React se pueden realizar aplicaciones escribiendo poco código.

En lenguajes como JavaScript y jQuery se manipula el DOM manualmente, paso a paso, de forma completamente granular e imperativa. Por ejemplo, veamos un programa que oculta elementos de una lista usando JQuery.

```
var x;
x=$(document);
x.ready(inicializarEventos);

function inicializarEventos()
{
     var x;
     x=$(“li”);
     x.click(presionItem);
}

function presionItem()
{
     var x;
     x=$(this);
     x.hide();
}
```

React proporciona un marco conceptual novedoso basado en componentes. Los componentes son los elementos que constituyen la interfaz del usuario, como por ejemplo un botón, un reloj, un buscador, etc. En React todo son componentes.

En React no necesitas manipular el DOM porque los componentes reaccionan tanto a los eventos producidos por el usuario como a los del servidor, repintándose a sí mismos cuando ocurre un cambio de estado. De ahí el nombre de la librería.

- **JSX**

Cada vez que un elemento actualiza su estado, los desarrolladores de Javascript o JQuery vuelven a repintar de forma imperativa un determinado trozo de código HTML en el DOM, lo cual implica escribir gran cantidad de código.

JSX es una extensión de Javascript inspirada en XML que facilita el volver a pintar los elementos del DOM. Nos permite generar código Javascript bajo la apariencia de código HTML. Veamos un ejemplo:

```
render() {
     return(
     <h1>Hello World</h1>
     );
}
```

Con JSX se observa con un golpe de vista lo que está pintando nuestro código, una etiqueta h1 con el texto Hello World!

Para ver la diferencia, si quisiéramos hacer lo mismo de forma imperativa con Javascript habría que detallarlo paso a paso, algo así:

```
<script>
     var cabecera
     cabecera=document.getElementById('h1');
     cabecera.value="Hello World!";
</script>
```

- **DOM Virtual**

React no trabaja directamente con el DOM porque es costoso en términos de rendimiento, sino con un objeto Javascript que lo simula: el DOM Virtual.

Actualizar el DOM es una tarea costosa en cuanto a rendimiento se refiere, por lo que cuantos más cambios de estado sean necesarios reflejar en él, más lenta irá nuestra web.

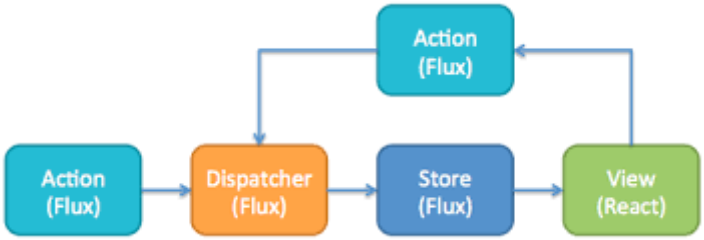
Cada vez que modificamos un elemento dentro del DOM, todos sus hijos tienen que ser pintados de nuevo, hayan o no hayan cambiado. Este proceso es el que provoca los problemas de rendimiento, ya que renderizar elementos en una interfaz gráfica es una tarea costosa.

El Virtual DOM es una representación en memoria del DOM real que actúa de intermediario entre el estado de la aplicación y el DOM de la interfaz gráfica que está viendo el usuario.

Dado que este DOM es virtual, la interfaz gráfica no se actualiza inmediatamente, sino que se compara el DOM real con el virtual con el objetivo de calcular la forma más óptima de realizar los cambios, es decir, se renderizan los menos cambios posibles. De este modo se consigue reducir en términos de rendimiento el coste de actualizar el DOM real. Además, los cambios resultan transparentes al navegador.

- **FLUX**

FLUX es una arquitectura de diseño de aplicaciones React que simplifica el desarrollo front-end, ayudando a construir apps robustas y fáciles de mantener. Sus tres piezas clave son las vistas (componentes de React), el despachador (Dispatcher) y los stores, y utiliza el esquema unidireccional:



En resumen, un componente React o una API del back-end genera una acción que se propaga al despachador central (Dispatcher). A continuación, se actualiza el estado (datos) de la aplicación (store) y los componentes React implicados responden repintándose a sí mismos.

A pesar de que Flux nació para solucionar ciertos problemas de escalabilidad del patrón MVC, se puede observar que si sustituimos el Dispatcher por el controlador y Store por el modelo, recuerda mucho al patrón MVC.

INTRODUCCIÓN A JSX

JSX es una extensión de la sintaxis de Javascript, utilizada por React para describir la composición de la interfaz de usuario. El motivo es que React asume el hecho de que la lógica de renderizado está muy unida a la lógica de la interfaz de usuario: cómo se manejan los eventos, cómo cambia el estado con el tiempo y cómo se preparan los datos para su visualización.

El método *render()* se utiliza para generar la interfaz visual del componente '*App*'. Mediante un *return* devuelve el JSX que debe mostrar el navegador. No es XHTML puro, es un nuevo formato que se compila generando JavaScript puro.

Dentro del bloque JSX se pueden incluir etiquetas HTML. Siempre deben tener etiqueta de comienzo y fin; si no hubiera etiqueta de fin se debe utilizar el carácter '/', en caso contrario se generaría un error al compilar la aplicación.

El método *render()* devolverá un único elemento HTML, aunque puede contener otros elementos en su interior. Si queremos devolver varios elementos un truco es devolver un elemento *div* que envuelva a todos.

```
const element = (
   <div>
      <h1>Hello</h1>
      <h2>Encantado de verte</h2>
    </div>
);
```

- **Expresiones en JSX**

Se pueden declarar variables y luego usarlas dentro de JSX encerrándolas entre llaves. En el siguiente ejemplo declaramos una variable llamada *name* y luego la usamos dentro de JSX:

```
const name = 'José Pérez';
const element = <h1>Hello, {name}</h1>;

ReactDOM.render(
     element,
     document.getElementById('root')
) ;
```

Dentro del bloque JSX se puede acceder a variables o constantes siempre y cuando se encierren entre llaves. Al compilar se mostrará el valor de la expresión correspondiente. Se podría poner cualquier expresión de Javascript dentro de llaves en JSX. Por ejemplo *2+2*, *user.firstname* o *formatName(user)*.

En el ejemplo siguiente se inserta el resultado de llamar a la función *formatName(user)* dentro de un elemento *<h1>*:

```
function formatName(user) {
     return user.firstName + ‘ ‘ +  user.lastName;
}

const user = {
     firstName: ‘José’
     lastName: ‘Pérez’
};

const element = (
     <h1>
         Hello, {formatName(user)}!
     </h1>
);

ReactDOM.render(
     element,
     document.getElementById(‘root’)
) ;
```

Después de compilarse, las expresiones de JSX se convierten en llamadas a funciones de Javascript. Por tanto se puede usar JSX dentro de sentencias *if* y bucles *for*, asignarlo a variables, aceptarlo como argumento y devolverlo desde dentro de una función.

Se divide el código JSX en varias líneas para facilitar la lectura. Aunque no es necesario, cuando se haga esto es recomendable envolverlo entre paréntesis para evitar errores por la inserción automática del punto y coma.

- **Especificar atributos en JSX**

Se pueden utilizar comillas para especificar strings literales como atributos. También se pueden utilizar llaves para insertar una expresión Javascript en un atributo:

```
const element = <div tabIndex=”0”></div>;
const element = <img src={user.imagen}></img>;
```

Dado que JSX está más cercano a Javascript que a HTML, React DOM usa la convención de nomenclatura *camelCase* en vez de nombres de atributos HTML. Por ejemplo, *class* se vuelve *className* en JSX y *tabindex* se vuelve *tabIndex*.

RENDERIZANDO ELEMENTOS

Los elementos son los bloques más pequeños de las aplicaciones de React. Un elemento describe lo que se quiere ver en pantalla.

```
const element = <h1>Hello, world</h1>;
```

React utiliza un nodo como raíz para renderizar el resultado a través de él. Las aplicaciones construidas sólo con React suelen tener un único nodo raíz en el DOM. Si estamos integrando React dentro de una aplicación existente se podrían tener tantos nodos raíz aislados como se quiera.

Para renderizar un elemento de React dentro de un nodo raíz del DOM simplemente hay que pasar ambos a *ReactDOM.render()*:

```
const element = <h1>Hello, world</h1>;
ReactDOM.render(element,document.getElementById('root'));
```

- **Actualización del elemento renderizado**

Los elementos de React son inmutables, esto quiere decir que una vez que se crea un elemento no se pueden cambiar sus hijos o atributos. Un elemento representa la interfaz de usuario en un momento concreto. La única manera de actualizar la interfaz de usuario es crear un nuevo elemento y pasarlo a *ReactDOM.render()*.

Veamos el siguiente ejemplo que representa un reloj en marcha:

```
function tictac()
{
  const element = (
    <div>
      <h1>Hola mundo!</h1>
      <h2>Son las {new Date().toLocaleTimeString()}.</h2>
    </div>
   );
   ReactDOM.render(element,document.getElementById('root'));
}

setInterval(tictac, 1000);
```

React compara el elemento y sus hijos con el elemento anterior, y solo aplica las actualizaciones del DOM que son necesarias para que se encuentre en el estado deseado.

COMPONENTES Y PROPIEDADES

Los componentes son una característica fundamental de *React*. Permiten separar la interfaz de usuario en piezas independientes, reutilizables y que se diseñan de forma aislada.

Son como funciones de Javascript, aceptan entradas llamadas props y devuelven a React elementos que describen lo que debe aparecer en pantalla.

Una aplicación consta de un componente principal llamado *App* (creado con *create-react-app*). En él se definen objetos de otros componentes, y así sucesivamente.

Todo componente debe heredar de la clase *Component* e implementar el método *render()*.

- **Componentes funcionales y de clase**

La forma más sencilla de definir un componente es escribir una función de Javascript:

```
function Welcome(props) {
   return <h1>Hello {props.name}</h1>;
}
```

Esta función es un componente de React válido porque acepta un solo argumento de objeto *props* con datos y devuelve un elemento de React. Estos componentes se llaman funcionales porque literalmente son funciones Javascript.

También se puede utilizar una clase para definir un componente:

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

Los dos componentes anteriores son idénticos desde el punto de vista de React.

- **Renderizando un componente**

Anteriormente, sólo vimos elementos de React que representan las etiquetas del DOM:

```
const element = <div />;
```

Sin embargo, los elementos también pueden representar componentes definidos por el usuario:

```
const element = <Welcome name=”Sara”  />;
```

Cuando React ve un elemento que representa un componente definido por el usuario, pasa atributos JSX e hijos a este componente como un solo objeto. A este objeto se le llama *props*.

Por ejemplo, este código muestra “Hello, Sara” en la página:

```
function Welcome(props) {
   return <h1>Hello, {props.name}</h1>;
}

const element = <Welcome name=”Sara”  />;
ReactDOM.render(
   element,
   document.getElementById(‘root’)
) ;
```

Lo que hace el código es llamar al método *render()* con el elemento *<Welcome name=”Sara” />*, es decir, React llama al componente *Welcome* con *{name: ‘Sara’}* como *props*.
El componente *Welcome* devuelve un elemento *<h1>Hello, Sara</h1>* como resultado.
React actualiza eficientemente el DOM para que coincida con ese resultado devuelto.

- **Composición de componentes**

Los componentes pueden referirse a otros componentes en su salida; esto permite usar la misma abstracción para cualquier nivel de detalle. Un botón, un cuadro de diálogo, un formulario o una pantalla, en aplicaciones React, todos son expresados comúnmente como componentes.

Como ejemplo, podemos crear un componente *App* que renderiza *Welcome* muchas veces:

```
function Welcome(props) {
   return <h1>Hello, {props.name}</h1>;
}

function App() {
   return  (
      <div>
         <Welcome name=”Sara”  />
         <Welcome name=”Sara”  />
         <Welcome name=”Sara”  />
      </div>
   );
}

ReactDOM.render(
   <App/>,
   document.getElementById(‘root’)
) ;
```

- **Extracción de componentes**

En ocasiones es útil dividir componentes en otros más pequeños. Veamos el siguiente ejemplo con el componente *Comment*:

```
function Comment(props) {
   return  (
      <div className=”Comment”>

         <div className=”UserInfo”>
            <img className=”Image”
               src={props.author.image}
               alt={props.author.name}
            />
            <div className=”UserInfo-name”>
               {props.author.name}
            </div>
         </div>

         <div className=”Comment-text”>
            {props.text}
         </div>

         <div className=”Comment-date”>
            {formatDate(props.date)}
         </div>
      </div>
   );
}
```

Este componente acepta un objeto (*author*), un texto (*text*) y una fecha (*date*) como props, y describe un comentario en una web.

Este componente resulta difícil de modificar debido a todo el anidamiento, y también es complicado reutilizar partes de él. Para hacerlo más sencillo, se podrían extraer algunos componentes de él.

Vamos a extraer el componente *Image*:

```
function Image(props) {
   return  (
      <img className=”Image”
         src={props.user.image}
         alt={props.user.name}
      />
   );
}
```

Este componente no tiene por qué estar siendo renderizado dentro de un componente *Comment*. Por eso se le ha cambiado el nombre de su propiedad por otro más genérico, *user*.

Es recomendable nombrar las *props* desde el punto de vista del componente, en lugar de la del contexto en el que se va a utilizar.

Utilizando este nuevo componente creado se podría simplificar el componente *Comment*, que quedaría de la siguiente manera:

```
function Comment(props) {
   return  (
      <div className=”Comment”>

         <div className=”UserInfo”>
            <Image user={props.author} />
            <div className=”UserInfo-name”>
               {props.author.name}
            </div>
         </div>

         <div className=”Comment-text”>
            {props.text}
         </div>

         <div className=”Comment-date”>
            {formatDate(props.date)}
         </div>
      </div>
   );
}
```

A continuación, vamos a extraer un componente *UserInfo*, que renderiza una *Image* al lado del nombre de usuario:

```
function UserInfo(props) {
   return  (
      <div className=”UserInfo”>
         <Image user={props.user} />
         <div className=”UserInfo-name”>
            {props.user.name}
         </div>
      </div>
   );
}
```

Esto permite simplificar *Comment* aún más:

```
function Comment(props) {
   return  (
      <div className=”Comment”>

         <UserInfo user={props.author} />

         <div className=”Comment-text”>
            {props.text}
         </div>

         <div className=”Comment-date”>
            {formatDate(props.date)}
         </div>

      </div>
   );
}
```

Puede resultar una tarea pesada extraer los componentes, pero es una buena técnica tener una batería amplia de componentes para reutilizar en aplicaciones más complejas.

Nota: al declarar un componente, ya sea como una clase o como una función, nunca debe modificar sus *props*.

ESTADO Y CICLO DE VIDA

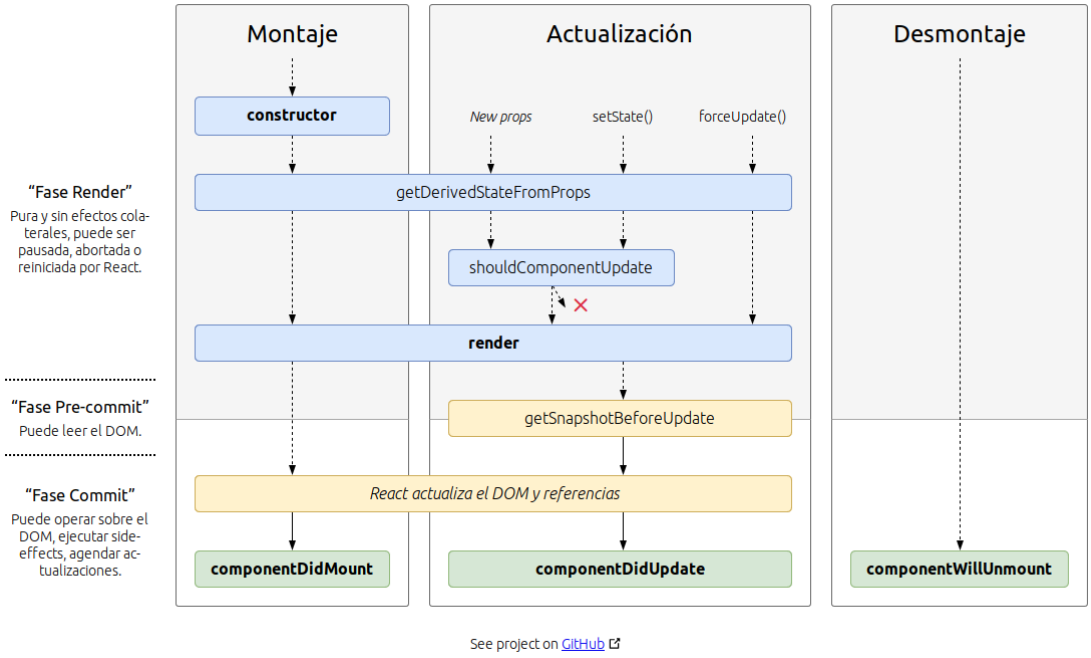
El ciclo de vida consiste en una serie de estados por los cuales pasa todo componente a la largo de su existencia. Estos estados tienen correspondencia en diversos métodos, que se podrán implementar para realizar acciones cuando se van produciendo.

En React es fundamental el ciclo de vida, porque hay determinadas acciones que deben realizarse necesariamente en el momento correcto de ese ciclo. Este es el motivo por el que hay que conocer muy bien las distintas etapas por las que pasa la ejecución de un componente React.

Esto es específico de los componentes con estado, ya que los componentes sin estado unicamente tienen un método para renderizar el componente y React no controlará su ciclo de vida a través de los métodos que veremos a continuación:

- El montaje se produce la primera vez que un componente se genera y será incluido 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.

Cada componente tiene varios “métodos de ciclo de vida” que se pueden sobreescribir para ejecutar código en momentos particulares del proceso.



En el diagrama se pueden observar las funciones del ciclo de vida que se utilizan comúnmente. Veamos las principales:

- **render()**

El método *render()* es el único obligatorio en un componente de clase.

Esta función debe ser pura, lo que quiere decir que no modifica el estado del componente, devuelve el mismo resultado cada vez que se invoca y no interactúa directamente con el navegador; si hiciera falta interactuar se haría en el método *componentDidMount()*.

Cuando se llama, debe examinar a *this.props* y *this.state* y devolver uno de los siguientes tipos:

- **Elementos de React**, normalmente creados a través de JSX, por ejemplo o <MyComponent />.
- **Arrays y fragmentos**, permiten devolver múltiples elementos.
- **Portales**, permiten renderizar hijos en otro subárbol del DOM, fuera de la jerarquía del nodo padre.
- **Strings y números**, son renderizados como nodos de texto en el DOM.
- **Booleanos o nulos**, no renderizan nada.

Nota: *render()* no será invocado si *shouldComponentUpdate()* devuelve falso.

- **constructor()**

Será necesario si tenemos que inicializar el estado del componente.

El constructor de un componente es llamado antes de ser montado. Al implementar el constructor para una subclase *React.Componet*, habría que llamar a *super(props)* antes de cualquier otra instrucción; de otra forma, *this.props* no estará definido en el constructor y esto podría ocasionar errores.

Normalmente, los constructores en React se utilizan por dos motivos:

- Para inicializar un estado local asignando un objeto al *this.state*.
- Para enlazar manejadores de eventos a una instancia.

No se debe llamar a *setState()* en el constructor. Si el componente necesitara usar el estado local, se asigna directamente el estado inicial al *this.state* en el constructor. Por ejemplo:

```
constructor(props) {
   super(props);
   // No llamar a this.setState() aquí
   this.state = {counter: 0}
   this.handleClick = this.handleClick.bind(this);
}
```

El constructor es el único lugar donde se debe asignar *this.state* directamente. En todos los demás métodos se debe usar *this.setState()* en su lugar.

- **componentDidMount**

Se invoca inmediatamente después de que un componente se monte (se inserte en el árbol). La inicialización que requiere nodos DOM debería ir aquí.

Este método se suele utilizar para establecer cualquier suscripción. Si se hace, no hay que olvidar darle de baja en *componentWillUnmount()*.

Se puede llamar a setState() inmediatamente desde aquí, aunque activaría un renderizado extra, pero sucederá antes de que el navegador actualice la pantalla.

- **componentDidUpdate**

Se invoca inmediatamente después de que la actualización ocurra. Este método no es llamado en el renderizado inicial.

Se usa para operar en el DOM cuando el componete se haya actualizado.

- **componentWillUnmount**

Se invoca inmediatamente antes de desmontar y destruir un componente. Este método se usa para realizar las tareas de limpieza necesarias.

Aparte de los métodos de ciclo de vida anteriores, que son llamados por React, existen dos más que podemos llamar desde nuestros componentes:

- **setState**

Este método hace cambios en el estado del componente y le dice a React que este componente y sus elementos secundarios deben volver a procesarse con el estado actualizado. Este es el método principal que se utiliza para actualizar la interfaz de usuario en respuesta a los manejadores de eventos y las respuestas del servidor.

React no garantiza que el componente se actualice inmediatamente. Por motivos de rendimiento puede retrasarlo y actualizar varios posteriormente de una sola pasada. Por tanto, si necesitamos leer *this.state* sería conveniente hacerlo en *componentDidUpdate()*.

- **forceUpdate**

Por defecto, cuando un componente cambia se renderizará. Si el método *render()* depende de algunos otros datos, se puede usar este método para indicarle a React que el componente necesita ser re-renderizado.

Anteriormente se vio como una manera de renderizar elementos para actualizar la interfaz de usuario. A continuación veremos cómo crear un componente; para ello partiremos de la función que actualizaba la hora.

```
function tictac()
   {
   const element = (
     <div>
        <h1>Hola mundo!</h1>
        <h2>Son las {new Date().toLocaleTimeString()}.</h2>
     </div>
   );
   ReactDOM.render(element,document.getElementById('root'));
   }

   setInterval(tictac, 1000);
```

Vamos a crear un componente Reloj verdaderamente reutilizable y encapsulado. Se comportará como un temporizador que se actualiza cada segundo.

Para empezar veremos cómo se encapsula el Reloj:

```
function Reloj(props) {
   return (
    <div>
      <h1>Hola mundo!</h1>
      <h2>Son las {new Date().toLocaleTimeString()}.</h2>
    </div>
   );
   ReactDOM.render(element,document.getElementById('root'));
}

function tictac() {
   ReactDOM.render (
      <Reloj date={new Date()} />,
      document.getElementById('root')
   ) ;
}

   setInterval(tictac, 1000);
```

En este caso se pierde el requisito de que la actualización de la interfaz de usuario cada segundo debe estar implementado en el Reloj. Para conseguirlo tendríamos que agregar un estado al componente. El estado es similar a las *props* pero es privado y está completamente controlado por el componente.

Para conseguir lo anterior habría que **convertir la función en una clase**. Lo haremos en cinco pasos:

1. Crear una clase con el mismo nombre que herede de *React.Component*.
2. Agregar un único método vacío *render()*.
3. Mover el cuerpo de la función al método *render()*.
4. Reemplazar *props* con *this.props* en el cuerpo de *render()*.
5. Borrar el resto de la declaración de función ya vacía.

```
class Reloj extends React.Component {
  render() {
  return (
    <div>
      <h1>Hola mundo!</h1>
      <h2>Son las {this.props.date.toLocaleTimeString()}.</h2>
    </div>
  );
  }
}
```

Con esto, *Reloj* se define como una clase en lugar de una función. El método *render()* se invocará cada vez que ocurre una actualización, pero siempre que se renderice en el mismo nodo del DOM, se usará una única instancia de la clase *Reloj*. Esto nos permite utilizar características adicionales como el estado local y los métodos del ciclo de vida.

- **Agregar estado local a una clase**

Moveremos *date* de las *props* hacia el estado en tres pasos:

1. Reemplazar *this.props.date* con *this.state.date* en el método *render()*:

```
class Reloj extends React.Component {
  render() {
  return (
    <div>
      <h1>Hola mundo!</h1>
      <h2>Son las {this.state.date.toLocaleTimeString()}.</h2>
    </div>
  );
  }
}
```

2. Añadir un constructor de clase que asigne el *this.state* inicial:

```
class Reloj extends React.Component {
   constructor(props) {
```

```
        super(props);
        this.state = {date: new Date()};
    }
    render() {
    return (
     <div>
       <h1>Hola mundo!</h1>
       <h2>Son las {this.state.date.toLocaleTimeString()}.</h2>
     </div>
    );
    }
}
```

Notar cómo se pasa *props* al constructor base. Los componentes de clase siempre deben invocar al constructor base con *props*.

3. Eliminar la *prop date* del elemento < *Reloj* />:

```
ReactDOM.render()(
   < Reloj />,
   document.getElementById('root')
);
```

Posteriormente habría que incluir el código del temporizador al propio componente. El resultado es el siguiente:

```
class Reloj extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
  return (
    <div>
      <h1>Hola mundo!</h1>-
      <h2>Son las {this.state.date.toLocaleTimeString()}.</h2>
    </div>
  );
  }
}
```

Habría que renderizarlo desde el componente padre:

```
ReactDOM.render()(
   < Reloj />,
   document.getElementById('root')
);
```

A continuación haremos que Reloj configure su propio temporizador y se actualice cada segundo.

- **Agregar métodos de ciclo de vida a una clase**

En aplicaciones con muchos componentes es muy importante liberar los recursos utilizados por los componentes cuando éstos se destruyen.

En este caso queremos configurar un temporizador cada vez que *Reloj* se renderice en el DOM por primera vez. Es lo que se llama “montaje” en React.

También querremos borrar ese temporizador cada que el DOM producido por *Reloj* se elimine. Es lo que se llama “desmontaje” en React.

Podemos declarar métodos especiales en la clase del componente para ejecutar algún código cuando un componente se monta y se desmonta:

- componentDidMount
- componentWillUnmount

Estos métodos son llamados “métodos de ciclo de vida”.

El método *componentDidMount()* se ejecuta después de que la salida del componente ha sido renderizada en el DOM. Sería el lugar correcto para configurar el temporizador.

```
componentDidMount() {
   this.timerID = setInterval(
       () = > this.tictac(),
       1000
   );
}
```

Notar como se guarda el ID del temporizador en *this*. Se pueden añadir manualmente campos adicionales a la clase si se necesita almacenar algo que no participa en el flujo de datos.

Eliminaremos el temporizador en el método de ciclo de vida *componentWillUnmount()*:

```
componentWillUnmount() {
   clearInterval(this.timerID);
}
```

Finalmente habría que implementar un método *tictac()* que el componente *Reloj* ejecutará cada segundo.

Utilizará *this.setState()* para programar actualizaciones al estado local del componente.

```
class Reloj extends React.Component {
   constructor(props) {
      super(props);
      this.state = {date: new Date()};
   }

componentDidMount() {
   this.timerID = setInterval(
      () = > this.tictac(),
      1000
   );
}

componentWillUnmount() {
    clearInterval(this.timerID)
}

tictac() {
   this.setState( {
      date: new Date()
   } );
}

render() {
  return (
    <div>
      <h1>Hola mundo!</h1>
      <h2>Son las {this.state.date.toLocaleTimeString()}.</h2>
    </div>
  );
  }
}
```

Y se renderizaría desde el componente App:

```
ReactDOM.render()(
   < Reloj />,
   document.getElementById('root')
);
```

MANEJANDO EVENTOS

Manejar eventos en elementos de React es muy similar a hacerlo con elementos del DOM. Hay algunas diferencias de sintaxis:

- Los eventos de React se nombran utilizando *camelCase* en vez de minúsculas.
- Con JSX se pasa una función como manejador del evento en vez de un string.

Por ejemplo en HTML sería:

```
<button onclick=”realizaAccion”>
     Realiza acción
</button>
```

Sin embargo en React es algo distinto:

```
<button onClick={realizaAccion}>
     Realiza acción
</button>
```

Cuando en React se quiere asignar un manejador de eventos a un objeto del DOM, se suele hacer cuando el elemento se renderiza inicialmente, en lugar de utilizar *addEventListener* para agregar el escuchador de eventos una vez que el elemento ya está creado.

Al definir un componente como una clase, los manejadores de eventos suelen ser un método de dicha clase. Por ejemplo, el siguiente componente *Toggle* renderiza un botón que permite al usuario cambiar el estado entre “Encendido” y “Apagado”.

```
class Toggle extends React.Component {
   constructor(props) {
      super(props);
      this.state = {isToggleOn: true};

      // Este enlace es necesario para que ‘this’ funcione en
      // el callback
      this.handleClick = this.handleClick.bind(this);
}

   handleClick()  {
      this.setState(state => ({
         isToggleOn: !state.isToggleOn
      }));
}

   render()  {
      return (
        <button onClick={this.handleClick}>
            {this.state.isToggleOn ? ‘ON’ : ‘OFF’}
         </button>
```

```
        );
    }
}
```

Y se renderizaría desde el componente App:

```
ReactDOM.render(
    <Toggle />,
    document.getElementById('root')
);
```

Hay que tener mucho cuidado en cuanto al significado de *this* en los *callbacks* de JSX, ya que en Javascript, los métodos de clase no están enlazados por defecto. Si se nos olvidara enlazar *this.handleClick* cuando lo pasamos a *onClick*, *this* no estaría definido cuando se llamara a la función.

- **Pasar argumentos a escuchadores de eventos**

Dentro de un bucle es muy común querer pasar un parámetro extra a un manejador de eventos. Por ejemplo, si *id* es el ID de una fila, lo podríamos hacer de cualquiera de las dos siguientes formas:

```
<button onClick={(e) => this.deleteRow(id, e)}>
    Borrar Fila
</button>

<button onClick={this.deleteRow.bind(this, id)}>
    Borrar Fila
</button>
```

Los dos códigos anteriores son equivalentes. En ambos casos, el argumento *e* que representa el evento de React va a ser pasado como segundo argumento después del ID. Con una función flecha hay que pasarlo explícitamente, pero con *bind* cualquier argumento adicional es pasado automáticamente.

RENDERIZADO CONDICIONAL

En React se pueden crear distintos componentes que encapsulan el comportamiento que se necesite. Dependiendo del estado de la aplicación, se pueden renderizar solamente algunos de ellos.

El renderizado condicional funciona de la misma forma que las condiciones de Javascript, usando el operador condicional.

Consideremos los siguientes componentes:

```
function UserGreeting(props) {
   return <h1>Bienvenido de nuevo!</h1>;
}

function GuestGreeting(props) {
   return <h1>Por favor, regístrese.</h1>;
}
```

Vamos a crear un componente *Greeting* que muestra cualquiera de estos componentes dependiendo de si el usuario ha iniciado sesión:

```
function Greeting(props) {
   const isLoggedIn = props.isLoggedIn;
   if (isLoggedIn) {
      return <UserGreeting />;
   }
   return <GuestGreeting />;
}

ReactDOM.render(
   // Intentar cambiando isLoggedIn={true}:
   <Greeting isLoggedIn={false} />,
   document.getElementById('root')
);
```

- **Evitar que el componente se renderice**

En ciertos casos, es posible que queramos que un componente se oculte a sí mismo aunque haya sido renderizado por otro componente. Para hacer esto, habría que devolver *null* en lugar del resultado renderizado.

El devolver *null* desde el método *render* de un componente no influye en la activación de los métodos del ciclo de vida del componente. Por ejemplo, *componentDidUpdate* seguirá siendo llamado.

FORMULARIOS

Los elementos de formulario en HTML funcionan un poco diferente a otros elementos del DOM en React, debido a que conservan naturalmente algún estado interno. Por ejemplo, este formulario escrito con HTML, acepta un solo nombre.

```
<form>
 <label>
  Nombre:
  <input type=”text” name=”nombre” />
 </label>
 <input type=”submit” value=”Enviar” />
</form>
```

Este formulario tiene el comportamiento predeterminado en HTML, se dirigirá a una nueva página al enviar el formulario. React funciona así directamente, pero en la mayoría de los casos querremos tener una función en Javascript que se encargue del envío del formulario y tenga acceso a los datos introducidos en el formulario. La forma predeterminada para conseguir esto es una técnica llamada “componentes controlados”.

- **Componentes controlados**

En HTML los elementos de formulario mantienen sus propios estados y se actualizan mediante la interacción del usuario. En React el estado se establece con la propiedad estado de los componentes, y solamente se actualiza con *setState()*.

Podemos combinar ambos haciendo que los componentes de React que rendericen un formulario también controlen las entradas del usuario. A esto a lo que se le llama “componente controlado”.

Por ejemplo, si en el formulario anterior quisiéramos que mostrara el nombre que se introduzca, se puede hacer con un componente controlado.

```
class NameForm extends React.Component {

   constructor(props) {
      super(props);
      this.state = {value: ‘’};

      this.handleChange = this.handleChange.bind(this);
      this.handleSubmit = this.handleSubmit.bind(this);
}

   handleChange(event)  {
      this.stetState({value: event.target.value});
}
```

```
        handleSubmit(event)  {
           alert('Se va a enviar un nombre: '+this.state.value);
           event.preventDefault();
    }

        render()  {
           return (
             <form>
               <label>
               Nombre:
               <input type="text" name="nombre" />
               </label>
               <input type="submit" value="Enviar" />
             </form>
           );
        }
    }
```

Al agregar el atributo *value* al elemento del formulario, el valor mostrado será el de *this.state.value*. Como el método handleChange se ejecutará cada vez que se pulsa una tecla para actualizar el estado de React, el valor mostrado se actualizará mientras que el usuario escribe.

Con un componente controlado, el valor del *input* siempre estará dirigido por el estado de React. Vemos que es necesario añadir más código, pero a cambio se consigue poder pasar el valor del campo a otros elementos de la interfaz de usuario, o incluso reiniciarlo desde otros manejadores de eventos.