Tema 2

Components y Props

Elementos React

- Los elementos son el bloque más pequeño que podemos encontrar para organizar nuestra aplicación dentro de ReactJS.

- Resumiendo, un elemento es *aquello que vemos en la pantalla*.

- Estos elementos son objetos planos, los cuales son muy sencillos de crear y no cargan en exceso la generación de la página.

- Un error que no debemos cometer es el de confundir los conceptos de **Elemento** y **Componente.**

- Como veremos más adelante, los Elementos son aquellos bloques que, combinándolos nos permiten generar las plantillas de nuestros Componentes.

```
const element = <h1>Hola, Mundo</h1>;
```

Elementos React

- Para renderizar cualquiera de estos elementos dentro de nuestra página, tenemos que *asociarlo* sobre un elemento disponible dentro del DOM.

- Por ejemplo, generamos una etiqueta de tipo **div** a la que vamos a designar como raíz de nuestro proyecto

```
<div id="root"></div>
```

- Las aplicaciones creadas con React suele tener únicamente un elemento raíz a partir del cual cuelgan el resto de elementos o componentes de la página.

- Si estamos integrando React dentro de algún proyecto en el que se usen otro tipo de tecnologías, tendremos que adecuar el elemento raíz al resto de etiquetas de la aplicación.

Elementos React

- Sobre dicho elemento raíz podemos renderizar cualquier elemento usando el método **ReactDOM.render().**

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

- El resultado de esta llamada será la evaluación del código JSX del elemento y su inclusión como HTML dentro del ámbito definido por el div raíz.

- Para asegurarte de que el resultado es el esperado, puedes inspeccionar el código generado en tu página.

Elementos React

- Una cosa importante a tener en cuenta es que los Elementos de React son **inmutables**, es decir, una vez se han enviado para ser renderizados en nuestra página no se pueden modificar.

- La única forma de modificar estos elementos es generando un nuevo objeto y enviándolo a través del método **render.**

```
function tick() {
  const element = (
    <div>
      <h1>Hora actual: {new Date().toLocaleTimeString()}.</h1>
    </div>
  );
  ReactDOM.render(
    element,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```

Elementos React

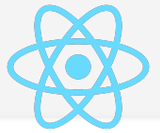
- En la práctica, la mayoría de aplicaciones React solo llaman al método **render** en una ocasión.

- Cuando avancemos en este tema iremos viendo cómo vamos a poder interactuar con los diferentes estados de nuestros componentes para poder visualizar los cambios necesarios en nuestras aplicaciones.

- También podemos prestar atención a qué elementos de todos los que renderizamos son los que React actualiza.

- Si en el ejemplo anterior inspeccionamos el código resultante, podremos apreciar que únicamente se producen cambios en la parte del código que es modificada, es decir, en la hora.

- Aunque, en cada ejecución estemos enviando el elemento completo, únicamente modifica el contenido que cambia.



Componentes

Componentes

- Los Componentes de React nos permiten dividir nuestra interfaz de usuario en piezas totalmente independientes, aisladas unas de otras y reutilizables, tanto en el mismo proyecto como en otros diferentes.

- Podemos identificar nuestros componentes con las funciones Javascript. Aceptan diferentes argumentos (llamados **props**) y devuelven Elementos React, los cuales conforman qué es lo que vamos a visualizar en la pantalla.

Componentes

- Existen dos maneras para poder definir nuestros componentes.

- La primera de ellas sería **generando una función Javascript**. Lo único que debe cumplir es que debe recibir una serie de argumentos (props) y devolver un elemento React válido.

- Podemos ver un ejemplo en el siguiente código:

```
function Saludo(props) {
  return <h1>Hola, {props.nombre}</h1>;
}

ReactDOM.render(
    <Saludo nombre="Pepe" />,
    document.getElementById('root')
);
```

Componentes

- La segunda manera para poder definir nuestros componentes es a través de clases ES6.
- Analizaremos el uso de clases para la definición de componentes más adelante, ya que tienen características extra.
- Un ejemplo equivalente al código anterior, sería el siguiente:

```
class Saludo extends React.Component{
    render(){
        return <h1>Hola, {this.props.nombre}</h1>;
    }
}

ReactDOM.render(
    <Saludo nombre="Pepe" />,
    document.getElementById('root')
);
```

Componentes

- En cualquiera de los dos casos anteriores, cuando React detecta un elemento definido por nosotros mismos, se encarga de pasarle al componente los atributos que correspondan y obtener los elementos que vamos a renderizar.

- Accedemos a dichos atributos, dentro del componente a través de la variable **props**.

- Para diferenciar nuestros componentes del resto de etiquetas del DOM, el estándar de React nos obliga a definirlos siempre con mayúsculas en la primera letra.

- Si React encuentra la etiqueta **<Saludo />**, se encargará de buscar un componente con ese nombre dentro del ámbito donde haya sido definido.

Componentes

- Dentro de nuestras aplicaciones creadas con React, podemos crear cualquier estructura anidando diferentes Componentes.

- De esta manera generamos, desde los elementos más sencillos (botones, etiquetas, desplegables…), hasta los más complejos (dialog, formulario…), a través de combinaciones.

- En el siguiente ejemplo vamos a crear, sobre el código desarrollado anteriormente, un nuevo componente llamado **App**, el cual se va a encargar de mostrar todas las veces que queramos el componente **Saludo**

Componentes

```
function Saludo(props){
    return <h1>Hola, {props.nombre}</h1>;
}

function App(props){
    return (
        <div>
            <Saludo nombre="Pepe" />
            <Saludo nombre="Rosa" />
            <Saludo nombre="Antonio" />
        </div>
    );
}

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

Componentes

- Habitualmente, encontraremos un único componente **App** dentro de nuestras aplicaciones React, el cual actúa como componente raíz y del cual vamos a anidar el resto de componentes.

- Si por el contrario, queremos integrar React dentro de una aplicación ya existente y desarrollada con otra tecnología, lo lógico es trabajar con Componente más sencillos, como por ejemplo podrían ser los que definen los botones con los que vamos a interactuar.

- Los Componentes **deben devolver un elemento único para ser renderizado.** Es por ello que en el ejemplo anterior, hemos anidado las llamadas al componente Saludo dentro de una etiqueta ** **

- Si eliminas la etiqueta div del ejemplo anterior puedes observar que devuelve error en la compilación.

Componentes - State

- Anteriormente hemos comentado que los componentes son inmutables una vez hemos llamado al método **render**.

- Para poder interactuar con nuestros componentes y que reciban automáticamente los cambios en los datos, tenemos que trabajar con el estado de cada uno de ellos.

- Estos estados se manejan de igual manera que las propiedades pero tienen carácter privado y solo son accesibles desde dentro del propio componente.

- **Solo podemos trabajar con los estados si nuestro componente está definido dentro de una clase.**

Componentes - State

- Mediante la asignación de propiedades dentro del objeto **this.state** de cada una de nuestras clases podemos hacer que los datos que se muestren se modifiquen cuando sea necesario.

- Para ello, es necesario que inicialicemos dichos estados. Podemos hacerlo dentro del constructor de la clase, el cual se ejecutará en el momento de renderizar el componente.

- Posteriormente podremos usar los datos almacenados dentro del estado en el punto de nuestra plantilla que necesitemos, accediendo siempre a los datos concretos.

- El siguiente ejemplo indica cómo podríamos transformar el ejemplo del reloj visto anteriormente sin necesidad de repetir la ejecución del componente.

Componentes

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

  render() {
    return (
      <div>
        <h2>Hora actual: {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

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

Componentes - State

- En el ejemplo observamos como en el método **constructor** inicializamos los objetos que queremos contener dentro de **this.state** para poder acceder a ellos posteriormente en la definición de nuestro JSX.

- Aparte, dentro del mismo constructor llamamos al constructor de la clase padre pasándole los atributos recibidos.

- Todos los componentes generados deberían heredar de **React.Component**, que es la clase base para cualquier componente que queramos utilizar dentro de nuestra aplicación.

- Implementamos el método **render**, mediante el cual vamos a definir los diferentes elementos React que vamos a renderizar en nuestra página una vez utilicemos nuestro componente.

Componentes - State

- Con este ejemplo hemos logrado definir la hora en la que el componente es renderizado en la página. (podría ser cualquier otro tipo de valor).

- El siguiente paso es generar, de manera interna al componente, la funcionalidad que nos permita modificar dicha hora cada segundo, para lograr que el componente funcione como en el ejemplo anterior pero sin necesidad de métodos de terceros.

Componentes - Ciclo de Vida

- Para conseguir que nuestro componente funcione como realmente necesitamos, podemos interactuar con los diferentes métodos del **ciclo de vida**.

- Se trata de una serie de métodos que se ejecutan automáticamente para cada uno de nuestros componentes según van alcanzando ciertas etapas.

- En dichos métodos tenemos que preocuparnos de liberar espacio cuando el Componente vaya a ser destruido o inicializar sus elementos cuando el Componente se vaya a mostrar dentro de nuestra aplicación.

- Existen numerosos métodos dentro del ciclo de vida pero, siguiendo con nuestro ejemplo anterior, de momento nos vamos a centrar en **componentDidMount** y **componenteWillUnmount**

Componentes - Ciclo de Vida

- El método **componentDidMount** se ejecuta después de que el contenido del componente se haya renderizado.

- Cuando se ejecuta todo el código incluido dentro de este método nos aseguramos que el componente ya está dentro de la interfaz de usuario y por tanto, podemos interactuar con cualquiera de sus elementos.

- Para nuestro ejemplo, es el sitio idóneo para iniciar el *timer.*

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

Componentes - Ciclo de Vida

- Aparte de poder usar **props** y **state**, en nuestras clases, podemos asignar cualquier tipo de atributo sobre **this.**

- En este ejemplo, hemos almacenado el identificador del timbre directamente sobre **this** para poder acceder a su valor en los diferentes métodos de la clase.

- Tenemos que asegurarnos que el Componente queda totalmente eliminado cuando dejemos de visualizarlo en nuestra página.

- Para ello podemos llevar a cabo las funciones precisas, dentro del método **componentWillUnmount**

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

Componentes - Ciclo de Vida

- Por último, vamos a definir el método **tick,** donde vamos a actualizar el valor de la propiedad definida dentro de **state**.

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

- El método **setState** es el que le indica al componente que sus datos se deben modificar y por lo tanto procederá a cambiar el contenido renderizado en cada una de las llamadas.

Componentes - Ciclo de Vida

- Cuando usamos la propiedad **state** de nuestros componentes debemos tener una serie de normas en cuenta:

- Para modificar alguno de los valores contenidos dentro de **state nunca debemos acceder directamente. Siempre se debe acceder a través de los métodos SET.**

```
// Incorrecto
this.state.saludo = "Hola";

// Correcto
this.setState({saludo: "Hola"});
```

Componentes - Ciclo de Vida

- Dentro de los diferentes métodos del ciclo de vida de un componente nos podemos encontrar múltiples llamadas al método **setState**. Estas llamadas, para no interferir en el correcto funcionamiento de la aplicación deberían ser asíncronas.

- Es por ello que el método **setState** dispone de una sintaxis en la que pasamos por parámetro una función que recibe el atributo props y el estado previo.

```
this.setState((prevState, props) => ({
  date: new Date()
}));
```

Componentes - Estilos

- Podemos definir los diferentes estilos de los elementos renderizados por un componente a través del atributo **className.**

- Posteriormente, implementaremos las clases CSS dentro de las hojas de estilo asociadas a nuestra aplicación, como si estuviéramos trabajando con CSS y HTML.

- Aparte, contamos con el atributo **style**, mediante el cual podemos definir nuestros estilos a través de Javascript

```
const divStyle = {
  color: 'blue',
  backgroundImage: 'url(' + imgUrl + ')',
};

<div style={divStyle}>Mensaje</div>
```