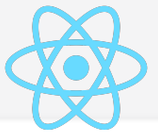

# Tema 3

## Eventos

# Eventos

- El manejo de eventos dentro de React es muy parecido a cómo lo hacemos con los elementos del DOM.

- Sin embargo tienen algunas diferencias:

    - Los eventos de React siguen la nomenclatura *camelCase* (onClick)

    - Al trabajar con JSX, el valor que pasamos al evento es una función en vez de un literal.

```
<button onClick={this.botonPulsado}>PULSA</button>
```

# Eventos

- Otra de las diferencias con la captura de eventos en javascript es que si queremos evitar el comportamiento por defecto del elemento con el cual estamos interactuando, no podemos devolver simplemente **false**.

- Hay que llamar explícitamente al método **preventDefault** dentro del método que captura el evento.

```
<a href="#" onClick={this.botonPulsado}>PULSA</a>


botonPulsado(e){
    e.preventDefault();
    console.log("Se ha pulsado el botón");
}
```

# Eventos

- Cuando trabajamos con HTML, para asignar eventos nuevos sobre los diferentes elementos de nuestra interfaz, tenemos que trabajar con el método **addEventListener**.

- En React podemos evitar la llamada a dicho método y simplemente tenemos que especificar el método que va a capturar el evento que nos interesa (el caso que hemos visto anteriormente con onClick).

# Eventos

- Si creamos nuestros componentes a través de clases ES6, el manejo del evento será un método de la propia clase.

- En este caso, hay que tener en cuenta el ámbito con el que estamos trabajando dentro del método *manejador.*

- Si usamos **this** dentro de este método, no estaremos haciendo referencia a la clase donde nos encontramos, si no que haremos referencia al evento que estamos capturando.

- Se puede modificar este comportamiento, agregando la siguiente modificación dentro del constructor de la clase

```
this.botonPulsado = this.botonPulsado.bind(this);
```

# Eventos

- Con la asignación anterior ya podemos utilizar todas las propiedades de la clase (props y state incluidas) dentro del método que maneja el evento.

- El el siguiente ejemplo vemos cómo acceder a state desde el método que maneja la pulsación de un botón.

```
class Boton extends React.Component {
  constructor(props) {
    super(props);
    this.state = {encendido: true};
    this.botonPulsado = this.botonPulsado.bind(this);
  }

  botonPulsado(e){
    this.setState( (prevState, props) => ({
      encendido: !prevState.encendido
    }));
  }

  render() {
    return (
      <div>
       <button onClick={this.botonPulsado}>
         {this.state.encendido ? 'ON' : 'OFF'}
       </button>
      </div>
    );
  }
}
```

# Eventos

- Existen otras dos opciones para enlazar el valor de this y poder usarlo dentro de nuestro manejadores de eventos.

- La primera opción es la utilización de una *arrow function* en el momento de definir la función manejadora.

```
botonPulsado = () => {
    this.setState( (prevState, props) => ({
        encendido: !prevState.encendido
    }));
}
```

# Eventos

- La segunda opción es la utilización de esta *arrow function* a la hora de definir quién maneja el evento:

```
<button onClick={(e) => this.botonPulsado(e)}>
    {this.state.encendido ? 'ON' : 'OFF'}
</button>
```

# Eventos - SyntheticEvent

- Cuando ejecutamos un evento, React se encarga de pasarle al método un objeto de tipo **SyntheticEvent**.

- Se trata de un objeto especial, el cual envuelve toda la funcionalidad propia del evento, asegurándonos la compatibilidad entre los diferentes navegadores.

- Incluso nos permite, si se diera el caso, poder acceder al propio evento del navegador a través de su propiedad **nativeEvent**.

# Eventos - SyntheticEvent

- Todos los objetos de tipo SyntheticEvent tienen los siguientes atributos:

  - boolean **bubbles**
  - boolean **cancelable**
  - DOMEventTarget **currentTarget**
  - boolean **defaultPrevented**
  - number **eventPhase**
  - boolean **isTrusted**
  - DOMEvent **nativeEvent**
  - void **preventDefault**()
  - boolean **isDefaultPrevented**()
  - void **stopPropagation**()
  - boolean **isPropagationStopped**()
  - DOMEventTarget **target**
  - number **timeStamp**
  - string **type**

# Eventos - SyntheticEvent

- Hemos de tener en cuenta que este tipo de objetos solo se encuentran disponibles en el ámbito de la función que maneja el evento. Una vez termina la ejecución de dicha función no podremos acceder a ninguna de las propiedades del evento.

- Este comportamiento responde a unas serie de implementaciones para la mejora en el rendimiento de nuestras aplicaciones.

- Por esto, tampoco podremos acceder a dichos objetos de manera asíncrona. Si lo hacemos obtendremos error o indefiniciones.

```
botonPulsado(event){
    console.log(event.type); // click
    const tipo = event.type; // click
    setTimeout(function(){
        console.log(event.type); // null
        console.log(tipo); // click
    });
}
```

# Eventos - SyntheticEvent

- Si queremos acceder a las propiedades del evento de manera asíncrona, tendremos que hacer la llamada al método **persist** sobre el objeto.

```
botonPulsado(event){
    console.log(event.type); // click
    const tipo = event.type; // click
    event.persist();
    setTimeout(function(){
        console.log(event.type); // click
        console.log(tipo); // click
    });
}
```

# Eventos

- Al igual que pasa al trabajar con HTML, existen numerosos eventos que podemos capturar para lograr una mayor interacción del usuario con nuestra página.

- Vamos a analizar alguno de los más interesantes y así poder ver qué usos les podemos dar dentro de nuestras aplicaciones.

# Eventos - Clipboard

- Son todos aquellos eventos que detectan cambios dentro del *portapapeles* de nuestra máquina.

- Es decir, nos permite controlar si el usuario ha copiado, cortado o pegado texto perteneciente a alguno de nuestros componentes.

- Los eventos que podemos capturar son: **onCopy**, **onCut**, **onPaste**

```
<div>
    <input type="text" onPaste={this.manejarPegoTexto} />
</div>

---------

manejarPegoTexto = (event) => {
    console.log(event.clipboardData.getData('Text'));
}
```

# Eventos - Keyboard

- Se trata de un tipo de eventos muy útiles a la hora de manejar la interacción con el usuario, ya que nos permiten conocer qué teclas son las que se han pulsado.

- Pueden ser de gran utilidad a la hora de validar campos de un formulario mientras se van rellenando.

- Los 3 eventos que podemos capturar son: **onKeyDown, onKeyPress, onKeyUp**

```
<div>
    <input type="text" onKeyPress={this.manejarEventoTeclado}
/>
</div>

----------

manejarEventoTeclado = (event) => {
    if(event.charCode === 13){
        alert("Se ha pulsado la tecla Enter");
    }
}
```

# Eventos - Focus

- Este tipo de eventos detectan cuándo cualquiera de los elementos de nuestra página recibe el foco, es decir, cuando pasa a estar activo o deja de estarlo.

- Son de gran utilidad para conocer qué elementos está modificando el usuario y mostrar información relacionada.

- Los eventos que podemos capturar son: **onFocus**, **onBlur**

```
// Los dos eventos apuntan a este método
manejarFoco = (event) => {
    if (event.type === 'focus'){
        console.log("El elemento recibe el foco");
    }else if (event.type === 'blur'){
        console.log("El elemento pierde el foco");
    }
}
```

# Eventos - Form

- Existen una serie de eventos que nos permiten gestionar la interacción del usuario con nuestros formularios.

- El uso de formularios es el método más recomendable para tratar la entrada de datos en nuestras aplicaciones, por lo que es muy importante controlar cómo se usan y validar qué datos recibimos.

- En el próximo tema dedicaremos un apartado para entrar en profundidad en su implementación.

# Eventos - Mouse

- Otra de las herramientas fundamentales cuando trabajamos con cualquier aplicación web es el ratón.

- Como veremos en siguientes eventos, esta funcionalidad cambia mucho dependiendo de si accedemos a la aplicación a través de un ordenador o a través de un dispositivo móvil, la interacción es diferente.

- Mediante este tipo de eventos podemos detectar qué movimientos o pulsaciones está realizando el usuario con su ratón sobre los componentes que estemos desarrollando.

- Los eventos más significativos que podemos capturar son: **onClick onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit, onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave, onMouseMove onMouseOut onMouseOver onMouseUp**

# Eventos - Mouse

```
manejaEntraRaton = (event) => {
    console.log("Entra el ratón");
};

manejaSaleRaton = (event) => {
    console.log("Sale el ratón");
};
```

```
manejaClickRaton = (event) => {
    console.log("El usuario hace click en X: " + event.clientX + " Y: " +
event.clientY);
};

_________

<div className="contenedor" onMouseEnter={this.manejaEntraRaton}
onMouseLeave={this.manejaSaleRaton} onClick={this.manejaClickRaton}>
    Contenido
</div>
```

# Eventos - Touch

- Cuando visualizamos nuestras aplicaciones sobre un dispositivo móvil, no disponemos de un ratón para poder accionar los diferentes elementos.

- Muchas de las acciones que realizamos con un ratón, no las podemos replicar con el dedo. Por eso existen una serie de eventos específicos para este tipo de interacciones.

- Los eventos disponibles son: **onTouchCancel, onTouchEnd, onTouchMove, onTouchStart**

# Eventos - Image

- Otra serie de eventos muy interesantes son los que manejan la carga correcta de las imágenes incluidas en nuestros componentes.

- Nos permiten tener un control sobre dichos elementos y actuar en consecuencia en caso de encontrar un error.

- Podríamos, por ejemplo, mostrar una imagen genérica, en caso de que encontremos error en la carga de alguna de nuestras imágenes principales.

- Los eventos que podemos capturar son: **onLoad** y **onError**

```
cargaImagen = (event) => {
    console.log("La imagen se ha cargado correctamente");
};

errorImagen = (event) => {
    console.log("Se ha producido un error en la carga de la
imagen");
}
```