3.6 Eventos en React

Contenidos

Introducción

Hasta ahora sólo hemos visto cómo pintar interfaces estáticas o independientes en React. Esta es la sesión divertida: en esta sesión veremos cómo añadir dinamicidad a los componentes con eventos.

Vamos a ver que los eventos nos permiten declarar qué reacciones tendrán nuestros componentes. Sin embargo, en ocasiones necesitamos que una reacción en un componente hijo/a, como puede ser la que causa un evento, provoque una reacción en el elemento padre/madre, o incluso más arriba en la cadena.

React solo nos permite pasar datos unidireccionalmente, de padres/madres a hijos/hijas. Aunque esto puede parecer una limitación, debemos recordar que las funciones en JavaScript se tratan como datos, podemos guardarlas en variables, y conservan las variables del ámbito donde se declararon. Así que, ¿y si declarásemos una función en el padre/madre que provoque una reacción en el propio componente y se la pasamos como `prop` a un hijo/a? Eso es una **práctica habitual en React** que se llama *lifting*, que significa "alzamiento".

¿Para qué sirve lo que vamos a ver en esta sesión?

React tiene un sistema de eventos sintéticos que ejecutan una acción cuando ocurre un acontecimiento. Con los eventos declararemos cómo *react-cionarán* nuestros componentes a la interacción con el usuario. Por ejemplo, cuando haga clic en un botón dentro de un componente que hemos definido.

Eventos sintéticos de React

Ya hemos visto cómo escribimos código parecido a HTML con JSX que se transforma en código HTML de verdad tras pasar por React. De una manera parecida, en React tenemos un sistema de **eventos sintéticos** que usaremos como si fueran normales. Aunque están diseñados para que pasen por eventos regulares, igual que JSX pasa por etiquetas HTML normales, cabe destacar que son una capa que nos proporciona React y que **no son eventos reales del DOM**, y por eso se llaman **sintéticos**.

En el módulo 2 vimos los eventos del DOM y cómo escuchar eventos desde JavaScript con `addEventListener()`. También vimos una manera de hacerlo desde atributos de HTML que **recomendamos no utilizar**, pero la enseñamos por si os la encontrábais en el futuro. Las funciones escuchadoras (*listeners*) de React se declaran de forma parecida a aquellos que no recomendábamos. Esto no es una contradicción: recordemos que en React escribimos interfaces declarativas, y esto es solo una sintaxis comprensible para asignar comportamiento. No debéis declarar funciones escuchadoras así fuera de React.

> Cuando escuchamos un evento, declaramos una función escuchadora (*listener*) que se ejecutará cuando se reciba un evento de cierto tipo. Esto es así tanto para eventos del DOM como para eventos sintéticos de React, sólo cambiaremos cómo asignamos la función al tipo de evento.

Vamos a ver un ejemplo. Queremos escuchar un evento de `click` desde un botón que declaramos con JSX. Escribiremos el botón (`<button>texto</button>`) y en un atributo `onClick` (ojo con la mayúscula) añadiremos la función "escuchadora", que será la reacción. Quedará así:

```
const alertButton =
  <button onClick={ /* aquí va la función */ }>
    Pedir más información
  </button>;
```

Podríamos escribir directamente la función escuchadora como una *arrow function* ahí, pero no quedaría legible. Preferiremos declararla fuera y la pasaremos (sin llamarla) al atributo de JSX:

```
const onClickListener = event => {
  alert('Para más información, acuda a recepción.');
};
const alertButton = (
  <button onClick={onClickListener}>Pedir más información</button>
);
```

Ya está. Cuando hagamos clic en el botón, React se encargará de escuchar el evento y de ejecutar la función.

▸ Evento simple en Codepen

Naturalmente, hay más atributos para escuchar eventos a parte de `onClick`. Los nombres de los atributos tendrán la forma `onEventoEscuchado`, con cada palabra del nombre del evento que se escucha escrita con mayúsculas iniciales. Es decir, escucharemos el evento `focus` rellenando el atributo `onFocus`, el evento `mouseover` rellenando el atributo `onMouseOver`, y así sucesivamente. Podéis consultar el listado completo de atributos soportados, pero a continuación vamos a listar los más usados, como ya hicimos en la sesión de eventos:

- Escuchadores de eventos de ratón:
  - `onClick`: botón izquierdo del ratón
  - `onMouseOver`: pasar el ratón sobre un elemento
  - `onMouseOut`: sacar el ratón del elemento
- Escuchadores de eventos de teclado:
  - `onKeyPress`: pulsar una tecla
- Escuchadores de eventos sobre elementos:
  - `onFocus`: poner el foco (seleccionar) en un elemento, por ejemplo un `<input>`
  - `onBlur`: quitar el foco de un elemento
  - `onChange`: al cambiar el contenido de un `<input>`, `<textarea>` o `<select>` (no es necesario quitar el foco del `input` para que se considere un cambio, al contrario que en el DOM)
- Escuchadores de eventos de formularios:
  - `onSubmit`: pulsar el botón submit del formulario
  - `onReset`: pulsar el botón reset del formulario

React no puede controlar los eventos de la ventana, así que los siguiente eventos sintéticos no existen (sí existen sus correspondientes eventos de DOM):

- ~~Escuchadores de eventos de la ventana~~
  - `onResize` ~~: se ha cambiado el tamaño de la ventana~~
  - `onScroll` ~~: se ha hecho scroll en la ventana o un elemento~~

EJERCICIO 1

Odio la cebolla

Vamos a crear un componente `OnionHater` que consta de un `textarea` . Escucharemos los evento de cambio del valor del `textarea` de forma que, si el texto escrito contiene la palabra 'cebolla' hagamos un `alert` diciendo 'ODIO LA CEBOLLA'.

> PISTA: para acceder al valor del `textarea` lo podemos hacer desde el objeto evento, el parámetro de la función escuchadora, con `event.target.value`
>
> PISTA: para comprobar si una cadena contiene un texto podemos usar el método `includes` de las cadenas

EJERCICIO 2

Elige tu destino

Vamos a crear un componente `Destiny` que contiene un `select` con un listado de ciudades: Buenos Aires, Sydney, Praga, Boston y Tokio. Al cambiar el valor del `select` , haremos aparecer un `alert` que diga 'Tu destino es viajar a XXX', siendo XXX la ciudad seleccionada.

Uso de métodos `handleEvent` para manejar eventos

No solo podemos usar funciones escuchadoras en elementos sueltos de JSX, como hemos hecho en la sección anterior, sino que también podemos declararlas en nuestros componentes de React. Pero antes de hacerlo, vamos a aclarar un término. Las funciones escuchadoras también se llaman *"event handlers"*, que significa "encargadas del evento", porque son **las funciones encargadas** de reaccionar al evento. En esta sección las llamaremos así para entender mejor y recordar los nombres que usaremos para estas funciones.

Es una práctica común declarar **dentro** del componente las *event handlers* que se usan en el componente. Como los componentes son clases, los *event handlers* serán **métodos** de la clase, y su nombre suele empezar por `handle` ("encargarse de", en inglés), seguido del nombre del evento. Por ejemplo, un *event handler* que se ocupe del evento `click` se llamará `handleClick()` .

Vamos a ver un ejemplo. Recordaréis el componente `RandomCat` que creamos en una sesión anterior. Este componente pintaba una foto aleatoria con un gato. Pero algunos días como hoy nos gusta más Bill Murray que los gatos, así que haremos un componente `RandomMurray` que pinte una foto aleatoria de Bill:

```
const getRandomInteger = (max, min) =>
  Math.floor(Math.random() * (max - min + 1)) + min;
const MIN_SIZE = 200;
const MAX_SIZE = 400;

class RandomMurray extends React.Component {
  render() {
    const randomMurray = getRandomInteger(MIN_SIZE, MAX_SIZE);

    return (
      <img
        src={`http://www.fillmurray.com/200/${randomMurray}`}
        alt="Random murray"
      />
    );
  }
}
```

Lo modificaremos para que, cuando hagamos clic en la imagen, se genere otra imagen. Para ello, declararemos un método `handleClick` en el componente:

```
// ...
class RandomMurray extends React.Component {
  handleClick() {
    // method body
  }
  // class body
}
```

La función que pinta nuestro componente, `render()`, es también la función que genera una imagen aleatoria al llamar a `getRandomInteger()`. Si pudiéramos hacer desde nuestra función `handleClick()` que React pintase con `render()` nuestro componente de nuevo, la imagen se generaría de nuevo. Afortunadamente, los componentes de React tienen un método `forceUpdate()` que sirve justo para eso:

```
// ...
class RandomMurray extends React.Component {
  handleClick(event) {
    this.forceUpdate();
  }

  render() {
    const randomMurray = getRandomInteger(MIN_SIZE, MAX_SIZE);

    return (
      <img
        src={`http://www.fillmurray.com/200/${randomMurray}`}
        alt="Random murray"
        onClick={this.handleClick}
      />
    );
  }
}
```

> No debemos usar el método `forceUpdate()` indiscriminadamente, sino delegar el `render` izado de componentes en React. De momento, solo sabemos que nuestros componentes se re- `render` izan cuando cambian sus `props`, pero en la próxima lección aprenderemos cómo hacerlo sin usar `forceUpdate()`.

Como nuestro método `handleClick()` contiene un `this` y el método lo ejecutará React desde otro contexto, tendremos que asegurarnos de que el `this` que usará es el `this` que queremos. Para eso, en el `constructor()` enlazaremos (*bind*, en inglés) el método a nuestro componente para que siempre que lo llamemos, se ejecute bien:

```
// ...
class RandomMurray extends React.Component {
  constructor(props) {
    super(props);

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

```
  }

  handleClick() {
    this.forceUpdate();
  }

  // ...
}
```

Ahora que tenemos nuestro método `handleClick()` declarado y enlazado, podemos registrarlo en el elemento JSX donde queremos escuchar el evento.

```
const getRandomInteger = (max, min) =>
  Math.floor(Math.random() * (max - min + 1)) + min;
const MIN_SIZE = 200;
const MAX_SIZE = 400;

class RandomMurray extends React.Component {
  constructor(props) {
    super(props);

    this.handleClick = this.handleClick.bind(this);
  }

  handleClick(event) {
    this.forceUpdate();
  }

  render() {
    const randomMurray = getRandomInteger(MIN_SIZE, MAX_SIZE);

    return (
      <img
        src={`http://www.fillmurray.com/200/${randomMurray}`}
        alt="Random murray"
        onClick={this.handleClick}
      />
    );
  }
}
```

▶ Métodos `handleEvent()` en Codepen

EJERCICIO 3

Expreso mi odio en rojo

Partiendo del código ejercicio 1, vamos a hacer que nuestro componente ocupe toda la pantalla disponible, y tenga el `textarea` en el centro. Vamos a hacer que al detectar la palabra cebolla en el texto, en vez de mostrar un alert mostrando nuestro odio, pongamos el fondo del componente de color rojo. Al volver a un texto sin cebolla, el fondo vuelve a ser blanco.

1. Guardaremos la información de si estamos odiando o no en un atributo de la clase que al comienzo es falso `this.isHating = false`
2. En la función que maneja el evento `change` del textarea modificaremos el atributo `isHating` y usaremos el método `forceUpdate` para forzar el repintado

> PISTA: recuerda que para que el `this` funcione correctamente en nuestra función de *handle* debemos hacer el `bind` adecuado en el constructor

BONUS: ¿Podrías hacer que nuestro odio aparezca tanto si 'cebolla' tiene mayúsculas o minúsculas?

EJERCICIO 4

Visualiza tu destino

Vamos a partir del ejercicio 2 sobre elegir tu destino. Vamos a crear un nuevo componente `CityImage` que muestra una imagen de una ciudad que recibe por props. Por ejemplo

```
<CityImage city="Praga" />
```

Debe mostrar una imagen de Praga. Para facilitar este comportamiento, este componente debe tener como uno de sus atributos un objeto literal con el formato:

```
{
  'Praga': 'http://path-to-praga-image.jpg',
  'Boston': 'http://path-to-boston-image.jpg',
  ...
}
```

Una vez que tenemos este componente funcionando, vamos a crear uno como hija de nuestro componente `Destiny`, es decir, vamos a hacer que `Destiny` contenga un `CityImage`. Para eso vamos a pintarlo en el JSX de su `render`.

Para terminar, vamos hacer que la magia suceda: en vez de hacer un alert, cuando la usuaria elija una ciudad en el select aparece la imagen de esa ciudad y se muestra el texto 'Tu destino es viajar a XXX'. Para conseguirlo os recomendamos usar un atributo de la clase `this.city` que cambie su valor al cambiar el select. También tendremos que usar `forceUpdate` para se ejecute el método `render` y a) se pasen unas props diferentes al componente `CityImage` y b) se pinte una ciudad diferente en el título.

Lifting (alzamiento) para pasar datos de hijos a padres

Lifting es una técnica que consiste en pasar funciones a los hijos/as y que sean estos quienes se encarguen de ejecutarlas cuando sea necesario, provocando un cambio *hacia arriba*, en los padres. Generalmente se usa para cambiar el *estado* de los padres, que luego provocará un re-`render`izado de los hijos/as con nuevas `props`. Aún no sabemos qué es el estado de un componente ni cómo usarlo, así que de momento utilizaremos `forceUpdate()` para provocar el re-renderizado manualmente.

Vamos a ver un ejemplo, con Murrays de nuevo. Tenemos tres componentes, `MurrayList`, `RandomMurray` y `ReloadButton`, cada uno en su módulo. `MurrayList` renderiza una lista con varios componentes `RandomMurray`, y además un botón `ReloadButton`.

components/MurrayList.js

```
import React from 'react';
import RandomMurray from `./RandomMurray`;
import ReloadButton from './ReloadButton';

class MurrayList extends React.Component {
  constructor(props) {
    super(props);

    // nos aseguramos de que este callback se ejecute siempre sobre el componer
    this.handleClick = this.handleClick.bind(this);
  }

```

```
  handleClick() {
    this.forceUpdate(); // se ejecutará el método `render()` de MurrayList, que
  }

  render() {
    const handleClick = this.handleClick;

    return (
      <section className="section-murrays">
        <h1>All <del>Cats</del> Murrays Are Beautiful</h1>
        <ul className="section-murrays_list">
          <li><RandomMurray /></li>
          <li><RandomMurray /></li>
          <li><RandomMurray /></li>
        </ul>
        {/* pasamos handleClick al hijo como prop */}
        <ReloadButton actionToPerform={ handleClick } label="More murrays"/>
      </section>
    );
  }
}

export default MurrayList;
```

Como vemos en el componente `MurrayList` llamamos a un componente `ReloadButton` al que pasamos por `props` una función que llamamos `actionToPerform` . Esta prop apunta al método `handleClick` que simplemente hace un `forceUpdate` que provoca una llamada al render y por tanto el re-renderizado de todas las hijas, las 3 `RandomMurray` , provocando el pintado de 3 nuevas imágenes aleatorias.

components/ReloadButton.js

```
import React from 'react';

class ReloadButton extends React.Component {
  render() {
    const actionToPerform = this.props.actionToPerform;
    const label = this.props.label || 'More';

    return (
      // Registramos el escuchador que recibimos por props. ¡Lifting hecho!
      <button onClick={actionToPerform}>{label}</button>
```

```
    );
  }
}

export default ReloadButton;
```

En `ReloadButton` es donde sucede el *lifting*: recogemos la función que llega de la madre por props (`actionToPerform`) y se la asignamos al `onClick` del botón. De esta forma, al hacer click en el botón ejecutamos una función de la madre.

components/RandomMurray.js

```
import React from 'react';

const getRandomInteger = (max, min) =>
  Math.floor(Math.random() * (max - min + 1)) + min;
const MIN_SIZE = 200;
const MAX_SIZE = 400;

class RandomMurray extends React.Component {
  render() {
    const randomMurray = getRandomInteger(MIN_SIZE, MAX_SIZE);

    return (
      <img
        src={`http://loremmurray.com/400/200/${randomMurray}`}
        alt="Random murray"
      />
    );
  }
}

export default RandomMurray;
```

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './stylesheets/index.css';
import MurrayList from './components/MurrayList';

ReactDOM.render(<MurrayList />, document.getElementById('react-root'));
```

▶ *Lifting* de eventos en Codepen

EJERCICIO 5

Ciudades

Para terminar de entender bien cómo funciona el lifting vamos a hacer un ejercicio muy sencillo. Partimos de un select con nombre de ciudades que encapsulamos en un componente `CitySelector`. Vamos a hacer que, al seleccionar una ciudad del select, aparezca una foto de la misma al lado. La diferencia con ejercicios anteriores es que ahora el select está en su propio componente. Para llevarlo a cabo debemos:

- guardar en un atributo de la clase la ciudad seleccionada inicial, por ejemplo, `this.selectedCity = 'Madrid'` y usarlo para pintar la imagen en el `render`
- crear un método `handleClick` que actualice el valor de `selectedCity` y llame a `forceUpdate` para forzar el repintado de la imagen
- y usar lifting para pasarlo al componente hijo que se ejecute al cambiar el select

> NOTA: en la próxima sesión veremos el estado de React que nos facilitará este flujo, pero de momento hacemos el repintado "a mano" con `forceUpdate`

EJERCICIO 6

Traductor MIMIMI

¿No os ha pasado nunca que habéis dicho algo y se han burlado de vosotras con un MIMIMI?

- "Al final de esa línea te falta un punto y coma."
- "Il finil di isi linii ti filti in pinti y cimi."

Pues es hora de contraatacar y crear nuestro propio traductor MIMIMI con React.

Vamos a partir de un formulario simple con un textarea donde escribimos una frase. Según vamos escribiendo, obtendremos en un párrafo el resultado de la traducción a MIMIMI. Es importante que tanto formulario como el párrafo resultado estén cada uno en su propio componente independiente. El componente del formulario, por ejemplo `TextInput`, simplemente se encarga de recoger los cambios de la usuaria y enviarlo al componente madre `App`, que los guarda en un atributo y fuerza el repintado. El componente `MIMIMITranslator` recoge el texto que le pasan por props, lo traduce y muestra en un párrafo.

> PISTA: para realizar la traducción basta con buscar una expresión regular (RegExp) y el método `replace` de las cadenas. Si buscas "javascript regex replace vowels" en Google va a ser fácil de encontrar.

EJERCICIO 7

Filtrando artículos

Vamos a partir del ejercicio 1 de la sesión 3.5 (los items de la lista de la compra). Si recuerdas bien, teníamos un componente `ItemList` que mostraba un listado de `Item` s. Vamos a crear un nuevo componente `CategoryButton` que es un botón con el nombre de una categoría de productos y que recibe por `props` el nombre de la categoría.

```
<CategoryButton category="Bebida" />
```

En nuestro `ItemList` vamos a pintar, además del `ul` con el listado de items, un botón con una categoría, por ejemplo, 'Bebida'.

> Recuerda que un componente de React sólo debe tener un elemento raíz en su método `render` y si queremos meter más cosas debemos agruparlas, por ejemplo, en un `div` .

Ahora viene lo bueno: vamos a escuchar eventos `click` sobre el botón de la categoría, de forma que se invoca un método del componente madre (`ItemList`) que hemos recibido por `props` (*lifting*). A este método de la madre le pasamos como parámetro el nombre de la categoría, para que sepa qué botón se ha clicado.

En `ItemList` definimos ese método para que si se pulsa el botón de la categoría bebidas sólo aparezcan los items de esa categoría en pantalla. Esto lo hacemos filtrando los items del array para dejar solo los de la categoría seleccionada.

Al final tendremos que realizar un cambio en un atributo y forzar la ejecución del método `render` de `ItemList` que a su vez fuerza el de sus hijas.

BONUS: cuando tenemos todo funcionando para una categoría, podemos añadir botones para cada de las que tengamos productos. Incluso un botón especial 'Todos' para mostrar de nuevo los productos de todas las categorías.

Recursos externos

Egghead

Serie de clases en vídeo que introduce y explora los fundamentos básicos de React.

- Eventos sintéticos de React

Documentación oficial de facebook sobre lifting

- Lifting State Up
- JavaScript Regular Expressions