educación IT
CODE YOUR FUTURE

ReactJS Developer

Módulo 3

Ciclo de vida y eventos

Eventos

Un evento es un suceso importante en la interfaz. Con esto nos referimos a que **debe ser atendido y la interfaz debe ofrecer una respuesta.**

La gestión de eventos tiene varias partes:

1. Por un lado, tenemos los **eventos** propiamente dichos, acontecimientos que ocurren **en nuestra interfaz.**
2. Por otro lado, tenemos los **escuchadores,** que son **porciones de código que atienden a un evento y ejecutan una respuesta personalizada al ocurrir dicho evento.**
4. Finalmente, los **disparadores,** son **las condiciones para que un evento ocurra.**

React nos ofrece la posibilidad de utilizar los mismos eventos que usamos en *HTML* y, además, **crear nuestros propios eventos.**

En React, toda la gestión de eventos se hace a través de *JSX* mediante **atributos específicos.** Estos atributos reciben como valor **funciones** que serán ejecutadas **cuando el evento se dispare.**

Veamos algunos de los eventos más utilizados en el desarrollo con React.

Evento *onClick*

Se dispara **al hacer click con el botón izquierdo sobre él.**

```
class App {
    constructor(props) {
        super(props);

        this.state = {
            nombre: "",
        };
    }

    mostrarMensaje()
    {
        alert("Hiciste click ahi")
    }

    render() {
        return (
            <div onInput={this.mostrarMensaje}>
                Clickeame
            </div>
        );
    }
}

export default App;
```

Evento *onInput*

Se dispara **al cambiar el valor de un campo de entrada.**

```
class App {
    constructor(props) {
        super(props);

        this.state = {
            nombre: "",
        };
    }

    cambiarNombre(e)
    {
        this.setState({
            nombre: e.target.value
        })
    }

    render() {
        return (
            <div>
                <input type="text" onInput={this.cambiarNombre} />
                {this.state.nombre}
            </div>
        );
    }
}

export default App;
```

Evento *onChange*

Se dispara **al cambiar el valor de un campo de entrada.** Es válido para elementos que **no sean entradas de texto,** como `Select`.

```
function App()
{

    const cambioPais = function(e)
    {
        alert("Has elegido "+e.target.value);
    }

    return (
        <select onChange={cambioPais}>
            <option value="">Seleccione su país</option>
            <option value="ARG">Argentina</option>
            <option value="BRA">Brazil</option>
            <option value="COL">Colombia</option>
            <option value="DIN">Dinamarca</option>
            <option value="ESP">España</option>
        </select>
    )
}

export default App;

```

Ciclo de vida

El ciclo de vida de un componente es, como bien indica su nombre, el conjunto de pasos secuenciales que transcurren **desde que se inserta un componente en la interfaz hasta que es quitado y limpiada la memoria.**

A grandes rasgos, cuando se trata de gestionar a los componentes de una interfaz, React tiene tres tareas principales:

1. La primera tarea es el **montado.** Montar un componente significa **insertarlo en el *VirtualDOM* de la página.**
2. Una vez que el componente está montado, debe reaccionar a los cambios en los datos a través de la **actualización.** La actualización se dispara **al cambiar el estado o el valor de las props de un componente** y ocasiona que **se vuelva a procesar con los nuevos datos.**
3. Finalmente, cuando el componente debe ser quitado de la interfaz, se procede con el **desmontado.** En esta fase el componente se quita y se liberan los recursos utilizados.

Ciclo de vida en componentes basados en clases

Montado

1. Se ejecuta el **`constructor(props)`** de la clase para crear la instancia.
2. Se llama al método **`componentWillMount.`** Este método tiene acceso al estado, pero si se cambia no ocasiona una actualización. Sirve para **validaciones del estado.**
3. Se llama al método **`render`**. Se dibuja el componente en la interfaz.
4. Se llama al método **`componentDidMount.`** Una vez que ya se creó el componente y se insertó, se llama a este método para poder hacer un re-render con nuevos datos. Es muy útil para llamadas a *API*.
5. Al recibir un cambio en las *props*, se dispara el siguiente método: **`componentWillReceiveProps(nextProps)`** Es útil para validar las nuevas *props.*

Actualización

1. Antes de actualizar el componente se llama al método **`shouldComponentUpdate(nextProps, nextState)`** que sirve para validar si el nuevo componente debe actualizarse con los nuevos datos.
2. Llamamos al método **`componentWillUpdate`** que tiene acceso a la *UI*.
3. A continuación, se llama al método **`render.`**
4. Luego se llama a **`componentDidUpdate`,** que sirve para interactuar con *APIs* y así refrescar la interfaz.

Desmontado

Esta fase consta únicamente del método **`componentWillUnmount`** que permite ejecutar acciones necesarias que están por fuera de las que hace React por defecto tales como **operaciones de cierre** y **liberación de recursos.**

Algunos consejos

- Si quieres que tu componente inicie con datos de fuentes externas, **no cargues el render inicial.** En su lugar, haz que el primer render se haga sin los datos de *API* y luego, en **`componentDidMount`** realiza los cambios de estado necesarios para que el componente se actualice con los datos de esa fuente externa.
- Puede que tu componente tenga ciertas validaciones para **`props`** o **`state`**. Imagina un componente `List` que recibe una *prop* llamada "`data`" que debe ser sí o sí un `array`. Para asegurar que React actualice este componente solo cuando reciba un nuevo `array` (y no otro tipo de dato) puedes usar **`shouldComponentUpdate`.**
- A veces usamos librerías de terceros que necesitan una limpieza especial de recursos y ahí es donde sirve el desmontado.

```
3   import React from "react";
4   class App extends React.Component
5   {
6       constructor(props)
7       {
8           super(props)
9
10          this.state = {
11              usuario: {
12                  nombre: "",
13                  apellido:""
14              }
15          }
16      }
17
18      componentDidMount()
19      {
20          fetch(`http://api.misitio.com/users/${this.props.id}`)
21          .then(r => r.json())
22          .then(usar => this.setState({ usuario: user}))
23      }
```

```
24
25      render()
26      {
27          return(
28              <div>
29                  <span> {this.usuario.nombre}</span>
30                  <span> {this.usuario.apellido}</span>
31              </div>
32          );
33      }
34  }
35
36  export default App;
37
```

Ciclo de vida en componentes funcionales

Montado

- El montado de un componente funcional es más sencillo. Simplemente **se llama a la función y el retorno de la función es el componente que se inserta en el *DOM*.**
- Es importante recordar que React lee la función **una sola vez al inicio.** El montado, en este caso, será **el único lugar donde se procese un código diferente al que está retornado** (el cuerpo de la función).
- También es el lugar para **inicializar todos los hooks necesarios.**

Actualización

- La actualización es ocasionada por **un cambio de estado o de props.**
- Consiste en volver a ejecutar **el elemento *JSX* retornado por la función** (y no toda la misma).

Desmontado

El desmontado es, también, más sencillo. Consiste simplemente en **la liberación de forma automática de todos los recursos consumidos por la función.**

Algunos consejos

- La gestión del ciclo de vida con componentes funcionales es **mucho más simple y fácil.**
- No siempre simple significa mejor. El costo de esa simplicidad se cobra haciendo que la personalización de determinadas partes del ciclo de vida sea **mucho más costosa,** por no decir imposible.
- El ciclo de vida de un componente funcional es **una reducción del ciclo de vida de un componente basado en clases.**
- Para ocasionar una actualización, usa el estado mediante **`useState`.**
- Para añadir comportamientos asincrónicos usa el hook **`useEffect`** que sirve justamente para eso. Este hook permite añadir comportamientos asincrónicos **protegiendo la creación y el montado.** Cuando se termine de montar en el *DOM*, se llaman todos los `callbacks` registrados con `useEffect`, por lo que tiene un comportamiento parecido a **`componentDidMount`.**

```jsx
import React, { useEffect, useState } from "react";

function App(props) {
    const [user, setUser] = useState({});

    useEffect(function () {
        fetch(`http://api.miistio.com/users/${this.props.id}`)
            .then((r) => r.json())
            .then((user) => setUser(user));
    });

    return (
        <div>
            <span>{user.nombre}</span>
            <span>{user.apellido}</span>
        </div>
    );
}

export default App;

```

educación IT
CODE YOUR FUTURE

¡Sigamos trabajando!