ThePower business school

🇪🇸 Conceptos ReactJS

Introducción

Existen algunos conceptos que debemos tratar, aún de una manera superficial, pero que poco a poco sean permeables en todo el contenido. Por ello haremos referencia a algunos aspectos a tener en cuenta.

Cambio de paradigma

1. Es un cambio de paradigma por lo que no debemos intentar agrupar conceptos de otros lenguajes o frameworks.
2. Los componentes de React se basan en funciones. A diferencia de otras librerías y frameworks que hacen uso de clases. [En su día React también se basaba en clases].
3. Las actualizaciones en React son asíncronas. Es decir si yo cambio un dato del estado de mi componente en la siguiente línea de código no veré ese cambio reflejado.
4. El estado en React es inmutable. Con ello queremos indicar que los datos que manejaremos en nuestra aplicación son inmutables.
    a. Estas restricciones se dan para construir aplicaciones robustas y con un buen rendimiento.
    b. Es un nuevo paradigma y debemos ser pacientes en su aprendizaje.

Aproximación funcional

Con la introducción de los Hooks en la versión 16.8 de React se ha pasado a desarrollar con una aproximación funcional.

1. Los componentes de tipo clase son legacy.

2. Podemos tener estados en componentes funcionales vía Hooks.
3. Olvídate de la herencia.
4. A la hora de reusar funcionalidad se prima la composición.
5. Cambio de paradigma para perfiles que vienen de Java, Angular o .Net.

Disección de un componente de React

En este caso es un pequeño componente que me permite introducir un nombre en un input y lo muestra por pantalla.

```
import React from 'react'

// 1.El componente se define en una funcion
const MyComponent = () => {
  // 2. Memoria de datos
  const [value, setValue] = React.useState('Alberto');
  // 3. Retorno de un jsx element
  return (
    <>
      <h4>{value}</h4>
      <input value={value} onChange={e => setValue(e.target.value)} />
    </>
  )
}

export default MyComponent;
```

1. El componente se define en una función. Esto quiere decir que cada vez que tiene que pintar el componente se ejecuta la función devuelve el contenido del return y la función muere.
2. Si esta función se crea y se destruye en cada pintado, dónde guardo los datos? Para ello usamos los Hooks de React, no te preocupes si el useState te suena a chino porque más adelante lo explicaremos en profundidad.
3. Un componente “siempre” retorna un jsx element. En nuestro caso un grupo de elementos que contiene un h4 y un input.

Inmutabilidad

1. Nunca modificaremos un objeto, en caso de necesitar un cambio crearemos uno nuevo. Esto en React indica que los datos son inmutables.
2. Imagina que tenemos la ficha estudiante con los campos: id, nombre y apellidos. Si quisiese cambiar el campo de nombre lo que haré será crear una ficha de estudiante nueva en la que copiaré todos los campos y sobreescribo la propiedad de nombre.
    a. Un objeto se puede usar en varios puntos de nuestra aplicación, si lo mutamos es como cambiarlo en un punto y no avisar al resto. Imagina que muto un campo “descuento” sobre un item y esto afecta a toda nuestra aplicación.
    b. Si muto un objeto de datos asociado a un componente ¿Cómo sabemos que tenemos que repintar el component? ¿os imagináis ir comparando sus propiedades una a una? En un estado de pedido modifico el dato y pierdo la referencia del estado del mismo.
    c. Si no mutamos nuestros objetos nuestra aplicación es predecible, es decir para saber si hay algún cambio solo tengo que comparar con el puntero de memoria del objeto.
    d. Trabajar con la inmutabilidad no es sencillo y por ello se puede recurrir a algunas librerías como immer o deepfreeze.

Asincronia

Nosotros podemos solicitar el pintado o repintado de contenido pero no de manera directa sino como he indicado “solicitamos” y es el propio motor de React quién realiza la acción. Por lo tanto esto se convierte en un proceso asíncrono. Vamos a ver este concepto con un ejemplo de código.

```
import React from 'react'

const MyComponent = () => {
  const [name, setName] = React.useState("Alberto");
  React.useEffect(() => {
    console.log("Hola " + name);
    setName("Cristian")
    console.log("Hola " + name);
```

```
  }, [third])

  return (
    <>
      <h2>Hello my name is {value}</h2>
    </>
  )
}
```

1. Inicializamos el estado de nuestro componente a “Alberto”.
2. Con el Hook de useEffect, que veremos más adelante en profundidad, pero lo que hace es que al montarse el componente en nuestro árbol del DOM cambia el nombre a “Cristian”.
3. Pero si hacemos un console.log() después del cambio veremos que el name que sigue apareciendo es “Alberto”.
4. Esto nos puede hacer dudar de la complejidad en el desarrollo pero realmente la naturaleza de una aplicación es asíncrona [Datos que recuperamos desde el Server]. Con esta aproximación React no ayuda a gestionar dicha naturaleza.
5. Además nos permite optimizar los repintados ya que puede modificar varios states en el mismo proceso.
6. Todo esto nos obliga a realizar un esfuerzo y pensar siempre en asíncrono.

Flujo unidireccional

React se basa en el flujo unidireccional de los datos, es decir tengo los datos de mi aplicación y estos fluyen del componente padre a los hijos. Dicho estado no se puede modificar desde un componente sino que realizamos una petición al estado para poder modificarlo [setState] y así el motor de React decidirá cuando lanzar esa actualización y notificar a los componentes de dicho cambio para su actualización.

Además tenemos que tener en cuenta que la información que recibe un hijo del padre es de solo lectura, es decir el hijo no puede modificar la información del padre. Pero en caso de querer hacerlo puede lanzar una solicitud al padre mediante un callback.

```
Componente padre =>
const pedido = {
  datos: {
    name:'Alberto',
    address:'C/Moratalaz 55'
  }
  detalle: [
    {id:1, name:'camiseta Rayo', price:55, qty:1},
    {id:14, name:'spiderman zombie hunter', price:350, qty:2}
  ]
}

Componente hijo 1=>
 datos: {
    name:'Alberto',
    address:'C/Moratalaz 55'
  }

Componente hijo 2=>
  [
    {id:1, name:'camiseta Rayo', price:55, qty:1},
    {id:14, name:'spiderman zombie hunter', price:350, qty:2}
  ]

Componente hijo 2.1=>
    {id:1, name:'camiseta Rayo', price:55, qty:1},
Componente hijo 2.2=>
    {id:14, name:'spiderman zombie hunter', price:350, qty:2}
```

Si observamos el funcionamiento o flujo de la información veremos:

1. La información se ha ido fragmentando de Padre a hijo y de hijos a nietos → Es decir el componente 2.1 o 2.2 tan solo tienen un fragmento de información y cualquier actualización sobre pedido deben solicitarla al componente 2 y este a su vez al componente padre.
2. Para solicitar este cambio burbujeamos hasta el componente padre a través de callbacks.
3. Únicamente el componente padre puede modificar porque es el que contendrá el estado de nuestra información → Es decir la visión global del pedido.