UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Programador Web Inicial
Front End Developer
React JS Componentes - Parte 2

Universidad Tecnológica Nacional - Facultad Regional Buenos Aires
Medrano 951 (C1179AAQ) C.A.B.A. Tel: (54 11) 4867-7511

UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Bloques temáticos:

- Hooks
- Veamos un poco más acerca de hooks
- Construir nuestros propios hooks
- useReducer
- Algunas consideraciones extra

Hooks

Los Hooks son una nueva característica en React 16.8. Estos te permiten usar el estado y otras características de React sin escribir una clase.

```
import React, { useState } from 'react'
function Example() {
    // Declara una nueva variable de estado, La cual Llamaremos "count"
    const [count, setCount] = usestate(0);
    return (
        <div>
            <p>You clicked {count} times</p>
            <button onClick={() => setCount(count + 1)}>
                Click me
            </button>
        </div >);
}
```

`useState` es un hook en react, y el mismo lo utilizaremos para el manejo de estados.

Veamos un poco más acerca de hooks

Hooks de estados

```
import React, { useState } from 'react'
function Example() {
    // Declara una nueva variable de estado, La cual Llamaremos "count"
    const [count, setCount] = usestate(0);
    return (
        <div>
            <p>You clicked {count} times</p>
            <button onClick={() => setCount(count + 1)}>
                Click me
            </button>
        </div >);
}
```

Este ejemplo renderiza un contador. Cuando haces click en el botón, incrementa el valor

Dentro de un componente funcional llamamos al método `useState` y de esta forma agregamos un estado local al mismo.

React mantendrá este estado entre re-renderizados. `useState` devuelve un par: el valor de estado actual y una función que le permite actualizarlo. Puedes llamar a esta función desde un controlador de eventos o desde otro lugar.

Es similar a `this.setState` en una clase, excepto que no combina el estado antiguo y el nuevo. El único argumento para `useState` es el estado inicial.

En el ejemplo anterior, es 0 porque nuestro contador comienza desde cero. Ten en cuenta que a diferencia de this.state, el estado aquí no tiene que ser un objeto — aunque puede serlo si quisieras. El argumento de estado inicial solo se usa durante el primer renderizado.

Veamos la diferencia entre el uso de hooks y el uso de clases:

Uso de clases

```
class Example extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            count: 0
        };
    }
    render() {
        return (
            <div>
                <p>You clicked {this.state.count} times</p>
                <button onClick={() => this.setState({ count:
this.state.count + 1 })}>
                    Click me
                </button>
            </div >
        );
    }
}
```

Uso con hooks

```
import React, { useState } from 'react'
function Example() {
    // Declara una nueva variable de estado, La cual Llamaremos "count"
    const [count, setCount] = usestate(0);
    return (
        <div>
            <p>You clicked {count} times</p>
            <button onClick={() => setCount(count + 1)}>
                Click me
            </button>
        </div >);
}
```

Como vemos es todo prácticamente igual!. Como vemos con `useState` estamos inicializando la variable de estado dentro del componente. Al hacerlo vemos que el primer elemento del array es la variable de estado y el segundo elemento es la función con la cual podremos modificar dicho estado, en este caso `setCount` (Sería el setState de la declaración por clases).

Ejemplo

```
const [fruit, setFruit] = useState('banana');
```

Esta sintaxis de Javascript se llama “desestructuración de arrays”. Significa que estamos creando dos variables `fruit` y `setFruit`, donde fruit se obtiene del primer valor devuelto por useState y setFruit es el segundo. Es equivalente a este código:

```
var fruitStateVariable = usestate( 'banana'); // Returns a pair
var fruit = fruitStateVariable[0]; // First item in a pair
var setFruit = fruitStateVariable[1]; // Second item in a pair
```

Utilizando múltiples variables de estados

```
const [todos, setTodos] = useState([{ text: 'Learn Hooks'}]);
```

Al contrario que en una clase, actualizar una variable de estado siempre la reemplaza en lugar de combinarla.

Hooks de efectos

El Hook de efecto, `useEffect`, agrega la capacidad de realizar efectos secundarios desde un componente funcional. Tiene el mismo propósito que `componentDidMount`, `componentDidUpdate` y `componentWillUnmount` en las clases React, pero unificadas en una sola API.

Ejemplo con clases

```
class Example extends React.Component
constructor(props) {
    super(props);
    this.state = {
    count: 0
};
}
componentDidMount() {
    document.title = 'You clicked ${this.state.count} times';
}
componentDidUpdate() {
    document.title = 'You clicked ${this.state.count} times';
}
render() {
    return (
        <div>
            <p>You clicked {this.state.count} times</p>
            <button onClick={() => this.setState({ count:
this.state.count + 1 })}>
                Click me
            </button>
        </div >
    );
}
```

Ejemplo con hooks

```
import React, { usestate, useEffect } from 'react';
function Example() {
    const [count, setCount] = useState(0);
    useEffect(() => {
        document.title = 'You clicked $(count} times';
    });
    return (
        <div>
            <p>You clicked {count} times</p>
            <button onClick={() => setCount(count + 1)}>
                Click me
            </button>
        </div>
    );
}
```

Declaramos la variable de estado `count` y le indicamos a React que necesitamos usar un efecto. Le pasamos una función al Hook `useEffect`. Esta función que pasamos es nuestro efecto. Dentro de nuestro efecto actualizamos el título del documento usando la API del navegador `document.title`. Podemos leer el valor más reciente de `count` dentro del efecto porque se encuentra en el ámbito de nuestra función.

Cuando React renderiza nuestro componente, recordará este efecto y lo ejecutará después de actualizar el DOM. Esto sucede en cada renderizado, incluyendo el primero.

Equivalencias entre Hooks de efectos y Ciclos de Vida

componentDidMount

```
useEffect ( () => {
/* ComponentDidMount code * /
}, []);
```

componentDidUpdate

```
useEffect ( () => {
/* componentDidUpdate code * /
}, [var1, var2]);
```

componentWillUnmount

```
useEffect ( () => {
return () => {
/* componentWillUnmount code * /
}
}, []):
```

Construir nuestros propios hooks

Construir tus propios Hooks te permite extraer la lógica del componente en funciones reutilizables.

Ejemplo tenemos un chat con un componente para mostrar el status de un amigo y otro para mostrar la lista de amigos:

FriendStatus

```
import React, { usestate, useffect } from 'react';
function FriendStatus(props) {
    const [isOnline, setIsOnline] = usestate(null);
    useEffect(() => {
        function handleStatusChange(status) {
            setIsOnline(status.isOnline);
        }
        ChatAPI.subscribeToFriendStatus(props.friend.id,
handleStatusChange);
        return () => {
            ChatAPI.unsubscribeFromFriendStatus(props.friend.id,
handleStatusChange);

        };
    });
    if (isOnline === null) {
        return 'Loading...';
    }
    return isOnline ? 'Online' : 'Offline';
}
```

FriendListItem

```
import React, { useState, useEffect } from 'react';
function FriendListItem(props) {
    const [isOnline, setIsOnline] = useState(null);
    useEffect(() => {
        function handleStatusChange(status) {
            setIsOnline(status.isOnline);
        }
        ChatAPI.subscribeToFriendStatus(props.friend.id,
handleStatusChange);
        return () => {
            ChatAPI.unsubscribeFromFriendStatus(props.friend.id,
handleStatusChange);
        };
    })
        ; return (
            <li style={{ color: isOnline ? 'green' : 'black' }}>
                {props.friend.name}
            </li>
        );
}
```

En este caso nuestra aplicación lista los usuarios y muestra con color verde aquellos que están conectados.

Vemos que el código del hook `useEffect` en ambos casos es igual entonces podríamos reutilizar dicha lógica creando nuestro propio hook.

```
import { useState, useEffect } from 'react'
function useFriendStatus(friendID) {
    const [isOnline, setIsOnline] = usestate(null);
    useEffect(() => {
        function handleStatusChange(status) {
            setIsOnline(status.isOnline);
        }
        ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
        return () => {
            ChatAPI.unsubscribeFromFriendStatus(friendID,
handlestatusChange);
        };
    });
```

```
    return isOnline;
}
```

Este hook mediante la función `useFriendStatus` recibe un id y realiza la lógica para determinar si el mismo está logueado o no.

Veamos cómo se modifican los códigos de los componentes antes mencionados.

FriendStatus

```
function FriendStatus(props) {
    const isOnline = useFriendStatus(props.friend.id);
    if (isOnline === null) {
        return 'Loading...'
    }

    return isOnline ? 'Online' : 'Offline';
}
```

FriendListItem

```
function FriendListItem(props) {
    const isOnline = useFriendStatus(props.friend.id);
    return (
        <li style={{ color: isOnline ? 'green' : 'black' }}>
            {props.friend.name}
        </li>
    );
}
```

useReducer

Acepta un reducer de tipo `(state, action) => newState` y devuelve el estado actual emparejado con un método dispatch

`useReducer` a menudo es preferible a `useState` cuando se tiene una lógica compleja que involucra múltiples subvalores o cuando el próximo estado depende del anterior. `useReducer` además te permite optimizar el rendimiento para componentes que activan actualizaciones profundas, porque puedes pasar hacia abajo dispatch en lugar de callbacks.

```
const initialState = { count: 0 };
function reducer(state, action) {
    switch (action.type) {
        case 'increment':
            return { count: state.count + 1 };
        case 'decrement':
            return { count: state.count - 1 };
        default:
            throw new Error();
    }
}
function Counter() {
    const [state, dispatch] = useReducer(reducer, initialState);
    return (
        <>
            Count: {state.count}
            <button onClick={() => dispatch({ type: 'decrement' })}>-
</button>
            <button onCLick={() => dispatch({ type: 'increment'
})}>+</button>
        </>
    );
}
```

En el ejemplo vemos un componente llamado `Counter`: El mismo hace uso de `useReducer`, en el primer parámetro se pasa la función reducer y en el segundo parámetro el estado inicial (count=0) useReducer nos retorna un state y una función dispatch

Vemos que cada vez que hacemos click en el botón “incrementar” o “decrementar” se realiza el dispacth y se envía como parámetro el tipo de operación (es el action de la función reducer y se puede enviar cualquier objeto que queramos como parámetro)

En la función reducer tenemos la lógica para determinar qué acción tomar con el estado en base al action que tomamos como parámetro. Esta parte es totalmente programable.

Algunas consideraciones extra

¿Tengo que nombrar mis Hooks personalizados comenzando con ”use“?

Por favor, hazlo. Esta convención es muy importante. Sin esta, no podríamos comprobar automáticamente violaciones de las reglas de los Hooks porque no podríamos decir si una cierta función contiene llamados a Hooks dentro de la misma.

¿Dos componentes usando el mismo Hook comparten estado?

No. Los Hooks personalizados son un mecanismo para reutilizar lógica de estado (como configurar una suscripción y recordar el valor actual), pero cada vez que usas un Hook personalizado, todo estado y efecto dentro de este son aislados completamente.

¿Cómo un Hook personalizado obtiene un estado aislado?

Cada llamada al Hook obtiene un estado aislado. Debido a que llamamos useFriendStatus directamente, desde el punto de vista de React nuestro componente llama a useState y useEffect.

Bibliografía utilizada y sugerida

Fedosejev, A. (2015). React.js Essentials (1 ed.). EEUU, Packt.

Amler, . (2016). ReactJS by Example (1 ed.). EEUU, Packt.

Stein, J. (2016). ReactJS Cookbook (1 ed.). EEUU, Packt.

https://es.reactjs.org/docs/hooks-overview.html https://es.reactjs.org/docs/hooks-state.html https://es.reactjs.org/docs/hooks-effect.html

https://es.reactjs.org/docs/hooks-custom.html