React intermedio

Abstracción de funcionalidades

React Hooks

ÍNDICE

- Componentes funcionales y estado
- Hooks básicos
- Ciclo de vida con Hooks

Componentes funcionales y estado

```
class Component extends React.Component {
    constructor(props) {
        super(props);
    }
    render() {
        return (
            <h1>¡Hola { this.props.name }!</h1>
        )
    }
}
```

Componente de clase

```
function Component(props) {
    return (
        <h1>¡Hola { props.name }!</h1>
    )
}
```

Componente funcional

Componentes funcionales y estado

```
class Component extends React.Component {
    constructor(props) {
      super(props);
      this.state = {
        counter: 0
      }
      this.increaseCounter = this.increaseCounter.bind(this);
    }
    increaseCounter(e) {
      e.preventDefault();
      this.setState(state => {
        return { counter: state.counter + 1}
      })
    }
    render() {
        return (
          <div>
            <h1>¡Hola { this.props.name }!</h1>
            <h2>Has pulsado { this.state.counter } veces</h2>
            <button onclick={this.increaseCounter}>Click aquí</button>
          </div>
        )
    }
}
```

- Antes de la versión 16.8 **los componentes de clase eran los únicos que tenían estado**
- Los componentes de clase **van en contra de la filosofía React de nunca mutar** debido a la existencia del **this**
- El ciclo de vida almacena lógicas combinadas y **difíciles de separar**

Componentes funcionales y estado

```
function Component(props) {
  const [counter, setCounter] = useState(0);
  return (
    <div>
      <h1>¡Hola { props.name }!</h1>
      <h2>Has pulsado { counter } veces</h2>
      <button onclick={() => setCounter(counter + 1)}>
        Click aquí
      </button>
    </div>
  )
}
```

- Los hooks introducen funcionalidades de **estado** y **ciclo de vida**
- Su funcionalidad es similar a la del **componente de clase**

Hooks básicos

```
function CountButton(props) {
    const [counter, setCounter] = useState(0);
    return (
        <div>
          <h2>Has pulsado { counter } veces</h2>
          <button onclick={() => setCounter(counter + 1)}>Click aquí</button>
        </div>
      )
}
```

Hook de estado

El hook de estado **useState** nos permite conservar un estado **memorizado y modificable** mediante una función setter devuelta

Hooks básicos

```
const [counter, setCounter] = useState(0);
```

Hook de estado

La variable puede guardar **cualquier valor**, y **podemos crear cuantas nuevas variables de estado necesitemos**, simplemente volviendo a llamar a este hook

Hooks básicos

```
setCounter(counter + 1);

setCounter(prevCounter => prevCounter + 1);
```

Hook de estado

Podemos **introducir el valor directamente** en la función setter o una función que **recibe de parámetro de entrada el valor actual** de la variable de estado dada

Hooks básicos

```
function CountButton(props) {
    const [counter, setCounter] = useState(0);
    useEffect(
        () => { document.title = `Has pulsado ${counter} veces`; },
        [counter]
    );
    return (
        <div>
          <h1>¡Hola { props.name }!</h1>
          <h2>Has pulsado { counter } veces</h2>
          <button onclick={() => setCounter(counter + 1)}>Click aquí</button>
        </div>
      )
}
```

Hook de efecto

El hook de efecto se encarga de tratar con **los efectos secundarios,** esto es, se encarga de realizar **cambios adicionales relacionados después de realizarse una renderización**

Hooks básicos

```
useEffect(
    () => { document.title = `Has pulsado ${counter} veces`; },
    [counter]
);
```

Hook de efecto

El hook de efecto ejecuta la función dada cuando se cumple una condición en un array de condiciones, esto es, **cuando una de las variables de estado o props en este array cambia**

Hooks básicos

```
useEffect(
    () => {
      document.title = `Has pulsado ${counter} veces`;
      return () => {
        document.title = `Título original`;
      }
    },
    [counter]
);
```

Hook de efecto

Si devolvemos una función dentro del callback a ejecutar esta funcionará como **función de saneamiento,** es decir, se ejecutará al desmontar el componente para sanitizar nuestros efectos

Hooks básicos

```
const CounterContext = React.createContext(0);

function App() {
  return (
    <CounterContext.Provider value={10}>
      <ContenedorDelBoton />
    </CounterContext.Provider>
  );
}
```

```
import counterContext from '../app.jsx';

function CountButton(props) {
    /* <-- Hooks previamente vistos -->*/
    const ctx = useContext(counterContext);
    return (
        <div>
          <h2>Has pulsado { counter + ctx } veces</h2>
          <button onclick={() => setCounter(counter + 1)}>
              Click aquí
          </button>
        </div>
    )
}
```

Hook de contexto

El hook de contexto nos ofrece acceso a los contextos de React como si se definiera un consumer y **permite acceso de lectura al valor actual de dicho contexto**

Ciclo de vida con Hooks

```
class CountButton extends React.Component {
    /* <-- inicializacion componente -->*/
    componentDidMount() {
      document.title = `Aún no has pulsado el botón`;
    }
    componentDidUpdate() {
      document.title = `Has pulsado ${this.state.counter} veces`;
    }
    componentWillUnmount() {
      document.title = `Titulo original`;
    }
    /* <-- renderizacion componente -->*/
}
```

- Los componentes funcionales **pueden sustituir también el ciclo de vida de los de clase**
- Las funciones del ciclo de vida, simplemente lidian con **los efectos secundarios**

Ciclo de vida con Hooks

```
function CountButton(props) {
    useEffect(
      () => {
        document.title = `Aún no has pulsado el botón`;
        return () => {
          document.title = `Titulo original`;
        }
      },
      []
    );
    useEffect(
      () => {
        document.title = `Has pulsado ${counter} veces`;
      },
      [counter]
    );
}
```

- Al usar el array vacío en las dependencias del hook indicamos **que se ejecute solo una vez**
- La función devuelta por el primer hook realiza la función del **componentWillUnmount**
- La función del update puede ser sustituida **por cuantos hooks de efecto necesitemos**

PARA RESUMIR

- Los Hooks de React permiten a los componentes funcionales **adquirir la funcionalidad ausente anteriormente de los componentes de clase**
- Las funcionalidades incluyen **estado**, **gestión de efectos secundarios** y **acceso a los contextos existentes de React**
- Usando los **Hooks de efecto** también es posible replicar de manera completa **el ciclo de vida disponible en los componentes de clase**