React

Índice

Elementos

Sintaxis de un elemento

Los elementos de React se escriben igual que los de HTML, hay algunas diferencias pero dentro de React se puede escribir HTML puro.

Una diferencia con HTML, por ejemplo, es que no se pueden escribir elementos que no se cierran, como imagenes o saltos de línea.

Por ejemplo, en HTML una imagen podría usarse así

```
<img src="/image" alt="imagen" />
```

Pero en React debe cerrarse ese tag, de esta manera

```
<img src="/image" alt="imagen" />
```

Atributos de un elemento

Los atributos en React no pueden contener caracteres especiales, por lo que algunos serán diferentes a los de HTML.

Un cambio importante en React es el del atributo clase, el cual cambia de `class` a `className`.

HTML:

```
<div class="clase"></div>
```

React:

```
<div className="clase"></div>
```

Componentes

Los componentes de React pueden ser escritos como clases o como funciones.

Un componente funcional de React se expresa de la siguiente manera

```
function App() {
    return <div>Hello world!</div>;
}
```

Los componentes de React deben empezar con una letra mayúscula obligatoriamente y devolver código JSX (HTML de React)

Fragmentos de React

Los componentes de React deben devolver un único elemento. Por ejemplo, esto no es válido.

```
function App() {
  return (
     <div>Hello world!</div>
     <div>Hello world!</div>
  );
}
```

Si se quiere devolver más de un elemento sin encerrarlo en otro, como un div, se debe usar un Fragmento, el cual puede expresarse de 2 maneras.

```
function App() {
    return (
        <>
            <div>Hello world!</div>
            <div>Hello world!</div>
        </>
    );
}
```

```
function App() {
    return (
        <React.Fragment>
            <div>Hello world!</div>
            <div>Hello world!</div>
        </React.Fragment>
    );
}
```

Propiedades

Al crear componentes en React, podemos añadirle propiedades con el parametro props, el cual recibe todos los atributos que se le pasan al crearlo en otro componente.

Por ejemplo, en el componente App puedo crear el componente User y pasar la propiedad nombre para que se muestre ese texto en el otro componente.

```
function App() {
    return <User nombre="John Doe" />;
}

function User(props) {
    return <h1>Hola, {props.nombre}</h1>;
}
```

Esto resultará en

```
<h1>Hola, John Doe</h1>
```

Hay algunas propiedades que se pasan por defecto, como la propiedad children, la cual contiene los elementos que pusiste adentro del componente.

Por ejemplo

```
function App() {
    return (
        <User>
            <h1>Hola, John Doe</h1>
        </User>
    );
}

function User(props) {
    return <div>{props.children}</div>;
}
```

Esto resultará en

```
<div>
    <h1>Hola, John Doe</h1>
</div>
```

Condicionales

Los condicionales en React se pueden hacer de 2 maneras, con operadores ternarios o con if.

Operadores ternarios

Los operadores ternarios se pueden usar para mostrar un componente si se cumple una condición.

Por ejemplo, si quiero mostrar un componente si el usuario está logueado, puedo usar un operador ternario de la siguiente manera

```
function App() {
    const isLogged = true;
    return <div>{isLogged ? <User /> : <Login />}</div>;
}
```

If

Los condicionales if se pueden usar para mostrar un componente si se cumple una condición.

Por ejemplo, si quiero mostrar un componente si el usuario está logueado, puedo usar un if de la siguiente manera

```
function App() {
    const isLogged = true;

    if (isLogged) {
        return <User />;
    } else {
        return <Login />;
    }
}
```

Listas

Para crear una lista con un array, se debe usar el método map, el cual recibe una función que se ejecutará por cada elemento del array.

Por ejemplo, si quiero mostrar una lista de usuarios, puedo usar un array de la siguiente manera

```
function App() {
    const users = ["John Doe", "Jane Doe", "Jack Doe"];

    return (
        <div>
            {users.map((user) => (
                <User nombre={user} />
            ))}
        </div>
    );
}
```

Hooks

Los hooks son funciones que nos permiten usar características de React sin tener que crear un componente. Acá hay una lista de los hooks mas conocidos.

useState

El hook useState nos permite crear variables de estado, las cuales se pueden modificar y React se encargará de actualizar el componente.

Por ejemplo, si quiero crear una variable de estado que se llame count y que empiece en 0, puedo usar el hook useState de la siguiente manera

```
function App() {
    const [count, setCount] = useState(0);
    return (
        <div>
            <p>Count: {count}</p>
            <button onClick={() => setCount(count + 1)}>Incrementar</button>
        </div>
    );
}
```

useEffect

El hook useEffect nos permite ejecutar código cuando se renderiza el componente o cuando se actualiza una variable de estado.

Por ejemplo, si quiero ejecutar un código cuando se renderiza el componente, puedo usar el hook useEffect de la siguiente manera

```
function App() {
    useEffect(() => {
        console.log("Se renderizó el componente");
    }, []);
    return (
        <div>
            <p>Count: {count}</p>
            <button onClick={() => setCount(count + 1)}>Incrementar</button>
        </div>
    );
}
```

Si quiero ejecutar un código cuando se actualiza una variable de estado, puedo usar el hook useEffect de la siguiente manera

```
function App() {
    const [count, setCount] = useState(0);
    useEffect(() => {
        console.log("Se actualizó la variable count");
    }, [count]);
    return (
```

```
        <div>
            <p>Count: {count}</p>
            <button onClick={() => setCount(count + 1)}>Incrementar</button>
        </div>
    );
}
```

useContext

El hook useContext nos permite usar el contexto de un componente padre en un componente hijo.

Por ejemplo, si quiero usar el contexto de un componente padre en un componente hijo, puedo usar el hook useContext de la siguiente manera

```
const UserContext = React.createContext();

function App() {
    return (
        <UserContext.Provider value="John Doe">
            <User />
        </UserContext.Provider>
    );
}

function User() {
    const user = useContext(UserContext);
    return <h1>Hola, {user}</h1>;
}
```

useRef

El hook useRef nos permite crear una referencia a un elemento del DOM.

Por ejemplo, si quiero crear una referencia a un elemento del DOM, puedo usar el hook useRef de la siguiente manera

```
function App() {
    const inputRef = useRef(null);
    return (
        <div>
            <input ref={inputRef} />
            <button onClick={() => inputRef.current.focus()}>Focus</button>
        </div>
    );
}
```

useMemo

El hook useMemo nos permite crear una variable que se actualiza solo cuando una variable de estado cambia.

Por ejemplo, si quiero crear una variable que se actualiza solo cuando el contador cambia, puedo usar el hook useMemo de la siguiente manera

```
function App() {
    const [count, setCount] = useState(0);
    const countDoble = useMemo(() => {
        return count * 2;
    }, [count]);
    return (
        <div>
            <p>Count: {count}</p>
            <p>Count doble: {countDoble}</p>
            <button onClick={() => setCount(count + 1)}>Incrementar</button>
        </div>
    );
}
```

useCallback

El hook useCallback nos permite crear una función que se actualiza solo cuando una variable de estado cambia.

Por ejemplo, si quiero crear una función que se actualiza solo cuando el contador cambia, puedo usar el hook useCallback de la siguiente manera

```
function App() {
    const [count, setCount] = useState(0);
    const incrementar = useCallback(() => {
        setCount(count + 1);
    }, [count]);
    return (
        <div>
            <p>Count: {count}</p>
            <button onClick={incrementar}>Incrementar</button>
        </div>
    );
}
```