Xtratego

ACADEMY

UTE EC


UTE
EC
Xtratego
ACADEMY

UTE EC | Xtratego ACADEMY

Modulo 3

xtratego.ec

Unidad 3.1

Técnicas Avanzadas de React.js

Optimización de rendimiento con React.lazy y Suspense

La función **React.lazy** permite la carga diferida de componentes, lo que mejora el rendimiento de la aplicación al dividir el código. Suspense se utiliza para mostrar contenido mientras se carga el componente. Esto es especialmente útil en aplicaciones grandes, donde cargar todos los componentes al inicio afectaría negativamente la experiencia del usuario.

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));
function MyComponent() {
   return (
      <Suspense fallback={<div>Loading...</div>}>
         <OtherComponent />
      </Suspense>
   );
}
```

Error boundaries en React

Los **error boundaries** son componentes que capturan errores de sus hijos y permiten mostrar mensajes o realizar acciones cuando ocurren errores en la UI

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }
  static getDerivedStateFromError(error) {
    return { hasError: true };
  }
  render() {
    if (this.state.hasError) {
      return <h1>Algo salió mal.</h1>;
    }
    return this.props.children;
  }
}
```

Higher-Order Components (HOC)

Un **Higher-Order Component (HOC)** es una función que toma un componente como argumento y retorna un nuevo componente con lógica adicional. Este patrón es útil para compartir comportamiento entre componentes sin duplicar código.

```
function withLogging(WrappedComponent) {
  return class extends React.Component {
    componentDidMount() {
      console.log(`Component ${WrappedComponent.name} montado`);
    }
    render() {
      return <WrappedComponent {...this.props} />;
    }
  };
}
```

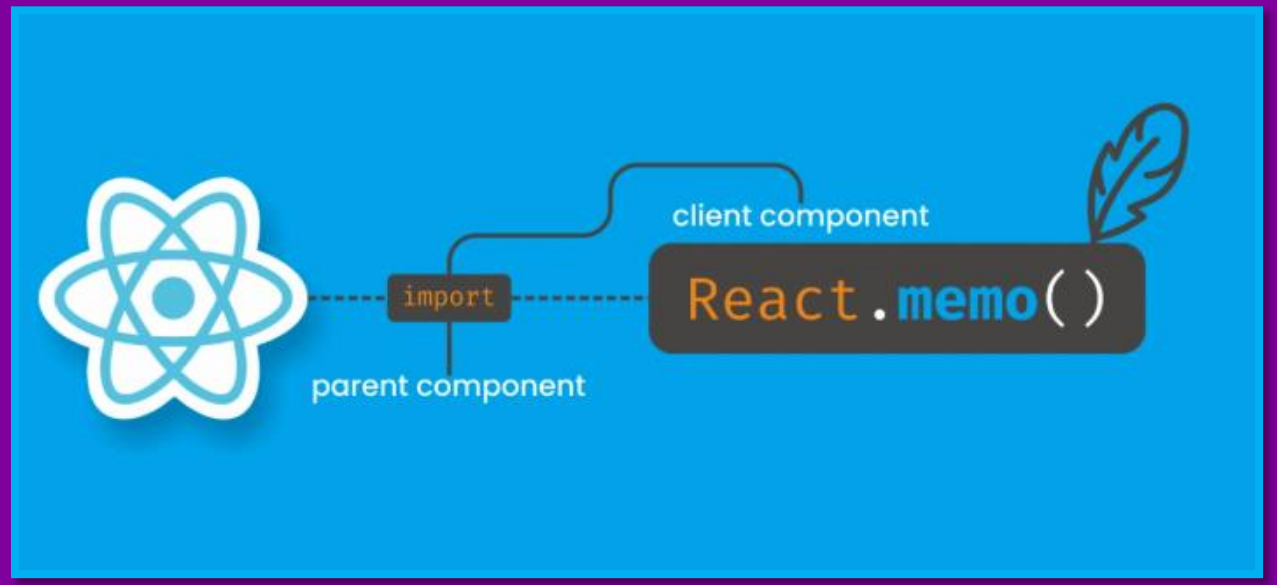
Render Props

La técnica de **Render Props** permite que un componente tome una función que retorna JSX como argumento. Esto es útil para compartir lógica entre componentes y manejar renderizados condicionales o dinámicos

```
const First = (props) => {
    return (
        <div>
            <h1>First Component</h1>
            {props.render()}
        </div>
    );
};
```

Memoization y React.memo

React.memo es una función de orden superior que memoriza el resultado de un componente funcional, evitando que se vuelva a renderizar si sus props no han cambiado. Esto es crucial en componentes que manejan grandes cantidades de datos o que se renderizan frecuentemente



Contextual Rendering

El **renderizado contextual** es una técnica avanzada que permite optimizar cómo y cuándo los componentes se vuelven a renderizar. En lugar de que todos los componentes se rendericen cada vez que cambia el estado global, se puede limitar el renderizado a los componentes que realmente lo necesitan.

Patrones de Componentes Dinámicos

En **React**, los **componentes dinámicos** permiten generar componentes basados en datos, lo que es útil cuando necesitas crear interfaces flexibles y reutilizables. Esta técnica puede aplicarse mediante la combinación de dynamic imports y la renderización condicional.

Docker lab play

https://labs.play-with-docker.com/

https://github.com/jairojumbo/docker-lab

UTE EC | Xtratego ACADEMY

Unidad 3.2

Hooks y Contexto

useReducer: Manejo de Estados Complejos

El **hook useReducer** es una alternativa avanzada a useState, útil cuando el estado de un componente es más complejo y depende de varias acciones. La principal diferencia es que useReducer permite manejar un estado que cambia mediante un patrón similar al de los "reducers" en Redux, con acciones que determinan cómo se debe modificar el estado.

useReducer: Manejo de Estados Complejos

Debes usar ***useReducer*** cuando:

- Tienes un estado que se ve afectado por varias acciones.
- El manejo del estado es complicado y puede beneficiarse de una lógica más estructurada (Johnson, 2021).

```
// Componente principal
function App() {
  const [state, dispatch] = useReducer(reducer, initialState);
```

useRef: Referencias en Componentes Funcionales

El **hook useRef** permite crear una referencia mutable que persiste durante el ciclo de vida de un componente sin causar re-renderizados. Es especialmente útil cuando se desea acceder directamente a un elemento del DOM o cuando necesitas conservar un valor entre renderizados sin que su cambio dispare una nueva renderización.

useRef: Referencias en Componentes Funcionales

En este ejemplo, el input puede ser enfocado directamente con una referencia a través de *useRef* sin provocar un re-renderizado del componente

```
import { useRef } from 'react';

function App() {
  const nameInputRef = useRef(null);
  const emailInputRef = useRef(null);
```

useMemo y useCallback: Optimización de Cálculos y Funciones

Los **hooks useMemo y useCallback** son esenciales para optimizar el rendimiento de las aplicaciones React, evitando cálculos y funciones innecesarias durante los re-renderizados. Ambos hooks se utilizan para memorizar un valor o una función respectivamente, de modo que solo se recalculen cuando sus dependencias cambien

useMemo y useCallback: Optimización de Cálculos y Funciones

useMemo: Memorizar Valores

useMemo memoriza el valor retornado de una función costosa para evitar su recalculación en cada renderizado, mejorando el rendimiento en componentes que realizan cálculos intensivos

useCallback: Memorizar Funciones

useCallback es similar a useMemo, pero está diseñado para memorizar funciones. Se utiliza cuando pasas funciones como props a componentes hijos que dependen de esas funciones

La API de Contexto: Gestión de Estado Global

La **API de Contexto** es una herramienta poderosa para gestionar el estado global en una aplicación React sin tener que pasar props manualmente a través de múltiples niveles de componentes, lo que se conoce como prop drilling. Esta técnica es útil cuando varios componentes en diferentes niveles necesitan acceder a los mismos datos

La API de Contexto: Gestión de Estado Global

En este ejemplo, el botón accede al valor del tema a través del contexto, eliminando la necesidad de pasar props a través de varios niveles de componentes

Ventajas de la API de Contexto

- Reduce la necesidad de prop drilling en aplicaciones grandes.
- Permite compartir datos de manera eficiente entre componentes sin pasarlos explícitamente como props

useEffect: Manejo de Efectos Secundarios

El **hook useEffect** es fundamental en React para manejar efectos secundarios como la carga de datos, suscripción a eventos o manipulación del DOM. useEffect se puede configurar para ejecutarse en diferentes momentos, dependiendo de las dependencias que le pases

```
useEffect(() => {
    document.title = `Has hecho clic ${count} veces`;
  }, [count]);
```

useEffect solo se ejecuta cuando la variable count cambia, lo que optimiza el rendimiento al evitar ejecuciones innecesarias

Manejo avanzado de efectos con limpieza

```
useEffect(() => {
    const handleResize = () => {
      console.log("Ventana redimensionada");
    };
    window.addEventListener("resize", handleResize);
    return () => {
      window.removeEventListener("resize", handleResize);
    };
  }, []);
```

Este ejemplo utiliza la función de limpieza dentro de useEffect para eliminar un event listener cuando el componente se desmonta, evitando fugas de memoria

Custom Hooks: Encapsulando Lógica Reutilizable

Los **hooks personalizados (custom hooks)** permiten encapsular lógica que puede ser reutilizada entre varios componentes. Un custom hook es una función de JavaScript que usa los hooks de React para implementar funcionalidad que puede ser compartida fácilmente

```
function useFetch(url) {
    const [data, setData] = useState(null);
    const [loading, setLoading] = useState(true);

    useEffect(() => {
      const fetchData = async () => {
        const response = await fetch(url);
        const result = await response.json();
        setData(result);
        setLoading(false);
      };

      fetchData();
    }, [url]);
    return { data, loading };
  }
```

Ejemplo: https://github.com/jairojumbo/proyecto-react

useContext con useReducer: Manejo Global del Estado

Una técnica poderosa es combinar useContext con useReducer para gestionar el estado global de una aplicación React sin la necesidad de usar Redux. Esto permite manejar el estado de una manera estructurada y reutilizable sin la complejidad añadida de librerías externas

```
const initialState = { count: 0 };
  function reducer(state, action) {
    switch (action.type) {
      case 'increment':
        return { count: state.count + 1 };
      default:
        throw new Error();
    }
  }

  const CountContext = React.createContext();
  function App() {
    const [state, dispatch] = useReducer(reducer, initialState);
    return (
      <CountContext.Provider value={{ state, dispatch }}>
        <Counter />
      </CountContext.Provider>
    );
  }
  function Counter() {
    const { state, dispatch } = useContext(CountContext);
    return (
      <div>
        Count: {state.count}
        <button    onClick={()    =>    dispatch({    type:    'increment'
})}>Increment</button>
      </div>
    );
  }
```

Combinación de Context API y Redux

Aunque la API de Contexto es poderosa por sí misma, puede complementarse con Redux para manejar estados más complejos. Usar Context para compartir el store de Redux entre componentes es una práctica común en aplicaciones grandes, ya que permite usar lo mejor de ambos mundos

Ejemplo Frontend: https://github.com/jairojumbo/proyecto-react

Ejemplo Backend: https://github.com/jairojumbo/api-express-node

Docker lab play

https://labs.play-with-docker.com/

Surge.sh

https://surge.sh/

GRACIAS