Xtratego

ACADEMY

UTE EC


UTE
EC
Xtratego
ACADEMY

UTE EC | Xtratego ACADEMY

Modulo 3

xtratego.ec

Unidad 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} />;
    }
  };
}
```

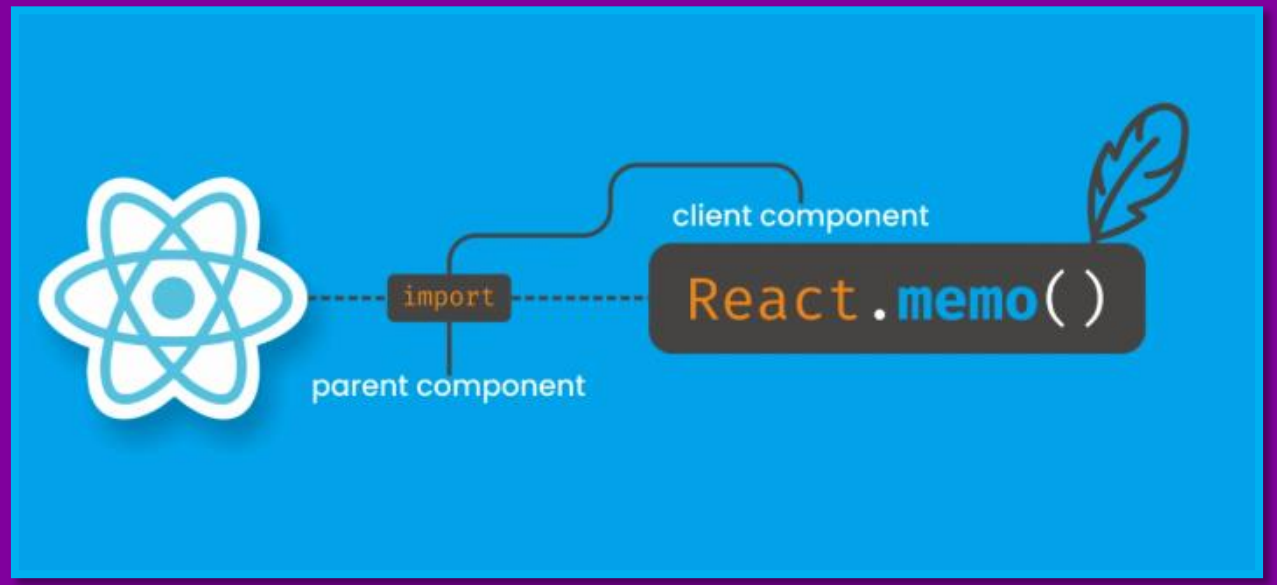
Render Props

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

Unidad 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/

Unidad 3

Optimización y Rendimiento en React

Control de Renderizaciones Innecesarias

En aplicaciones grandes de **React**, uno de los problemas más frecuentes es la **renderización innecesaria** de componentes, lo cual impacta negativamente en el **rendimiento**. Cada vez que cambia un estado o una prop, React renderiza los componentes afectados, aunque muchas veces esto no sea necesario, provocando un uso ineficiente de recursos

Estrategias para evitar renderizaciones innecesarias

Separación de componentes lógicos y presentacionales: Mantener la lógica de negocio separada de la presentación ayuda a que los componentes de UI solo se rendericen cuando reciben nuevas props, sin verse afectados por cambios de estado internos.

Uso adecuado de props: Es fundamental pasar únicamente las props necesarias a los componentes hijos, evitando el envío de objetos o funciones innecesarias que puedan forzar renderizaciones innecesarias.

Ejemplo de renderización innecesaria

```
function ParentComponent({ propA, propB }) {
    return (
      <div>
        <ChildComponentA propA={propA} />
        <ChildComponentB propB={propB} />
      </div>
    );
  }
```

En este ejemplo, si propA cambia, tanto ChildComponentA como ChildComponentB se renderizarán nuevamente, aunque solo ChildComponentA realmente necesita actualizarse, Como solucionamos esto?

```
function ParentComponent({ propA, propB }) {
    return (
      <div>
        <ChildComponentA propA={propA} />
        <MemoizedChildComponentB propB={propB} />
      </div>
    );
  }
  const MemoizedChildComponentB = React.memo(ChildComponentB);
```

Usar React.memo asegura que ChildComponentB solo se renderice cuando propB cambie

Técnicas de Partición de Código (Code Splitting)

La partición de código es una técnica que permite dividir la aplicación en pequeños fragmentos que se cargan solo cuando se necesitan, mejorando significativamente el tiempo de carga inicial. Esta técnica es útil en aplicaciones grandes donde el tiempo de carga completo de todos los componentes puede ser elevado

React.lazy y Suspense

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));
  function MyComponent() {
    return (
      <div>
        <h1>Componente principal</h1>
        <React.Suspense fallback={<div>Cargando...</div>}>
          <OtherComponent />
        </React.Suspense>
      </div>
    );
  }
```

Ventajas de la partición de código

Reducción del tiempo de carga inicial: Cargar solo los componentes que se necesitan en el inicio mejora significativamente el rendimiento percibido por el usuario

Mejora de la experiencia del usuario: Al implementar la partición de código, se reduce el tiempo de espera para que los usuarios interactúen con la aplicación.

Optimización del Estado Global

El manejo del estado es uno de los principales factores que afecta el rendimiento en aplicaciones React. Aunque React proporciona mecanismos eficientes como el hook useState, el manejo del estado global en aplicaciones grandes puede ser problemático. A continuación, se describen técnicas y herramientas para optimizar la gestión del estado global

```
import React, { createContext, useState } from 'react';
  export const AuthContext = createContext();
  export const AuthProvider = ({ children }) => {
    const [isAuthenticated, setIsAuthenticated] = useState(false);
    const login = () => setIsAuthenticated(true);
    const logout = () => setIsAuthenticated(false);
    return (
      <AuthContext.Provider value={{ isAuthenticated, login, logout }}>
        {children}
      </AuthContext.Provider>
    );
  };
```

Uso eficiente de Context API

La Context API es útil para compartir estado entre múltiples componentes, pero su uso incorrecto puede generar problemas de rendimiento debido a renderizaciones innecesarias. Para evitar estos problemas, es importante usar Context de manera adecuada, asegurándose de que solo los componentes que realmente necesiten acceder al estado se re-rendericen.

```
import React, { useContext } from 'react';
 import { ThemeContext } from './ThemeContext';
 function ThemedComponent() {
   const { theme, toggleTheme } = useContext(ThemeContext);
   return (
     <div>
       <p>El tema actual es: {theme}</p>
       <button onClick={toggleTheme}>Cambiar tema</button>
     </div>
   );
 }
 export default ThemedComponent;
```

Uso de Redux o Zustand

En aplicaciones grandes, herramientas como Redux o Zustand son mejores alternativas que la API de Contexto para manejar estados globales complejos. Estas librerías permiten separar la lógica del estado en "slices" o porciones manejables, evitando la sobrecarga de la aplicación

```
npm install zustand
```

store.js

```
import create from 'zustand';
  const useStore = create((set) => ({
    isAuthenticated: false,
    login: () => set({ isAuthenticated: true }),
    logout: () => set({ isAuthenticated: false }),
  }));
  export default useStore;
```

React Profiler: Identificación de Cuellos de Botella

React Profiler es una herramienta integrada en React que permite monitorear el rendimiento de los componentes y detectar cuellos de botella. Puedes usarlo para medir el tiempo que tardan los componentes en renderizarse y optimizar aquellos que están afectando la eficiencia de la aplicación.

Pasos

- Abre el panel de desarrollo del navegador y selecciona la pestaña Profiler.
- Realiza acciones en la aplicación para generar renderizaciones.
- Usa las métricas proporcionadas para identificar los componentes que están tardando más tiempo en renderizarse

Optimización basada en Profiler

Una vez identificados los componentes que tardan en renderizarse, se pueden aplicar técnicas como partición de código, memoización o la simplificación de los cálculos en los componentes para mejorar su rendimiento

```
import React, { Profiler, useState } from 'react';
  function Counter() {
    const [count, setCount] = useState(0);
    const increment = () => {
      setCount(count + 1);
    };
    return (
      <div>
        <h1>Contador: {count}</h1>
        <button onClick={increment}>Incrementar</button>
      </div>
    );
  }
  function App() {
    // Callback para el Profiler
    const onRenderCallback = (
      id, // ID del componente Profiler
      phase, // 'mount' o 'update'
      actualDuration, // Tiempo que tomó renderizar la actualización
      baseDuration, // Duración estimada para renderizar sin optimización
      startTime, // Timestamp de cuando comenzó el render
      commitTime, // Timestamp de cuando el render se completó
      interactions // Conjunto de interacciones desencadenadas durante el render
    ) => {
      console.log(`Renderización de ${id} en la fase ${phase}`);
      console.log(`Duración real: ${actualDuration}ms`);
      console.log(`Duración base: ${baseDuration}ms`);
    };
    return (
      <Profiler id="CounterComponent" onRender={onRenderCallback}>
        <Counter />
      </Profiler>
    );
  }
  export default App;
```

Optimización de la Carga de Imágenes y Recursos Pesados

Las imágenes y otros recursos multimedia son a menudo la causa principal de tiempos de carga lentos en aplicaciones web. Implementar estrategias de carga eficiente de imágenes puede mejorar significativamente el rendimiento de tu aplicación

Estrategias para optimizar la carga de imágenes

- **Lazy Loading de imágenes:** Similar a la partición de código, el lazy loading de imágenes permite cargar las imágenes solo cuando entran en el viewport del usuario.
- **Compresión de imágenes:** Utiliza imágenes comprimidas para reducir el peso de los recursos sin sacrificar demasiada calidad visual

```
function LazyLoadImage({ src, alt }) {
    return <img loading="lazy" src={src} alt={alt} />;
  }
```

Optimización del Rendimiento en Dispositivos Móviles

El rendimiento en dispositivos móviles es crucial, ya que estos suelen tener menos recursos y conexiones a internet más lentas en comparación con los dispositivos de escritorio. Para optimizar el rendimiento en dispositivos móviles, se pueden aplicar las siguientes técnicas

- Optimización de la carga inicial: Minimiza los recursos que se cargan al inicio para que la aplicación sea interactiva más rápidamente.
- Minimización de scripts: Usa herramientas como Webpack o Parcel para minimizar y empaquetar los archivos JavaScript, reduciendo su tamaño total

Ejemplo de Lazy load para componentes y recursos pesados

```
import React, { Suspense, useState } from 'react';
  // Cargar los componentes de forma diferida
  const HeavyComponent = React.lazy(() => import('./HeavyComponent'));
  function App() {
    const [showHeavyComponent, setShowHeavyComponent] = useState(false);
    return (
      <div>
        <h1>Optimización en Dispositivos Móviles</h1>
        <button onClick={() => setShowHeavyComponent(true)}>
          Cargar Componente Pesado
        </button>
        <Suspense fallback={<div>Cargando...</div>}>
          {showHeavyComponent && <HeavyComponent />}
        </Suspense>
      </div>
    );
  }
  export default App;
```

Medición del Rendimiento con Lighthouse

Lighthouse es una herramienta automatizada para mejorar la calidad de las aplicaciones web. Permite medir el rendimiento de tu aplicación, analizando métricas clave como el First Contentful Paint (FCP) y el Time to Interactive (TTI)

Pasos para usar Lighthouse

- Abre las herramientas de desarrollo del navegador y selecciona la pestaña Lighthouse.
- Ejecuta una auditoría para obtener un informe detallado sobre el rendimiento de tu aplicación.
- Usa las sugerencias proporcionadas para mejorar las áreas identificadas como problemáticas.

UTE EC | Xtratego ACADEMY

Unidad 4

Accesibilidad Web en React, despliegue y hosting

Principios Fundamentales de la Accesibilidad Web

Los principios fundamentales de la accesibilidad web están basados en las Pautas de Accesibilidad para el Contenido Web (WCAG), que organizan los requisitos en cuatro categorías principales: Perceptible, Operable, Comprensible y Robusto (POUR, por sus siglas en inglés). Estos principios establecen la base para diseñar aplicaciones accesibles

Principios Fundamentales de la Accesibilidad Web

- **Perceptible:** El contenido debe presentarse de forma que los usuarios puedan percibirlo. Esto incluye texto alternativo para imágenes y videos con subtítulos.
- **Operable:** La interfaz y los componentes de navegación deben ser operables, lo que significa que todos los usuarios deben poder interactuar con ellos, incluso a través de dispositivos de entrada alternativos.
- **Comprensible:** El contenido debe ser comprensible, es decir, el texto debe ser claro y las interfaces deben ser predecibles.
- **Robusto:** El contenido debe ser compatible con diferentes tecnologías, incluidos los lectores de pantalla y navegadores antiguos.

Uso de Etiquetas Semánticas y ARIA

La semántica correcta en **HTML** es esencial para garantizar que los lectores de pantalla y otros dispositivos de asistencia puedan interpretar correctamente el contenido. Las etiquetas HTML semánticas como <header>, <nav>, <main>, <article>, y <footer> proporcionan información clara sobre la estructura de la página

Uso de Etiquetas Semánticas y ARIA

Las etiquetas ARIA (Accesible Rich Internet Applications) mejoran la accesibilidad de las aplicaciones dinámicas añadiendo atributos que proporcionan contexto adicional a los usuarios de tecnologías de asistencia

```
function AccessibleButton() {
    return (
      <button aria-label="Cerrar ventana">
        <span aria-hidden="true">X</span>
      </button>
    );
  }
```

Accesibilidad en Formularios

Los formularios son una parte crítica de muchas aplicaciones web, y su accesibilidad es clave. Para mejorar la accesibilidad en los formularios, es importante usar correctamente las etiquetas <label> y asociarlas a sus campos correspondientes mediante el atributo for o id. Esto ayuda a los usuarios de lectores de pantalla a identificar claramente los campos de entrada

```
function AccessibleForm() {
  return (
    <form>
      <label htmlFor="name">Nombre:</label>
      <input type="text" id="name" name="name" />

      <label htmlFor="email">Correo electrónico:</label>
      <input type="email" id="email" name="email" />

      <button type="submit">Enviar</button>
    </form>
  );
}
```

Accesibilidad en Formularios

Es importante también proporcionar feedback accesible, como mostrar mensajes de error cuando los campos no se llenan correctamente. En React, puedes gestionar fácilmente los estados de validación y errores para hacer que los formularios sean más accesibles

```
function AccessibleFormWithError() {
  const [name, setName] = useState('');
  const [error, setError] = useState('');
  const handleSubmit = (e) => {
    e.preventDefault();
    if (!name) {
      setError('El campo de nombre es obligatorio');
    } else {
      setError('');
    }
  };
  return (
    <form onSubmit={handleSubmit}>
      <label htmlFor="name">Nombre:</label>
      <input type="text" id="name" value={name} onChange={(e) =>
setName(e.target.value)} />
      {error && <span role="alert" aria-live="assertive">{error}</span>}
      <button type="submit">Enviar</button>
    </form>
  );
}
```

Navegación por Teclado y Foco

La navegación accesible por teclado es un principio clave en la accesibilidad web. Asegúrate de que todos los elementos interactivos sean accesibles a través del teclado y de que el foco se administre correctamente. En React, puedes gestionar el foco utilizando el hook useRef para enfocarte en elementos cuando sea necesario

```
import { useRef } from 'react'
function FocusInput() {
  const inputRef = useRef(null)
  const focusInputField = () => {
    inputRef.current.focus();
  };
  return (
    <div>
      <input ref={inputRef} type="text" placeholder="Ingresa tu nombre" />
      <button onClick={focusInputField}>Focar el input</button>
    </div>
  );
```

Verificación de Accesibilidad en React con Herramientas

- **React Testing Library**: Te permite escribir pruebas que aseguran que los elementos interactivos, como botones y enlaces, son accesibles. También puedes verificar que los atributos ARIA estén implementados correctamente.
- **Axe DevTools**: Es una extensión del navegador que audita la accesibilidad de tu aplicación directamente en el navegador y proporciona recomendaciones detalladas.
- **Lighthouse**: Es una herramienta integrada en Chrome DevTools que audita el rendimiento, la accesibilidad, y las mejores prácticas, proporcionando un informe detallado con sugerencias de mejoras.

Evaluar la Accesibilidad con Lighthouse

Lighthouse es una herramienta esencial para auditar la accesibilidad. Proporciona un informe detallado con puntuaciones y recomendaciones para mejorar la accesibilidad de tu aplicación web. Puedes ejecutar Lighthouse desde las herramientas de desarrollo de Chrome y obtener sugerencias específicas para cumplir con los estándares WCAG

Preparación para el Despliegue

Empaquetar la Aplicación: Ejecuta el comando de compilación para empaquetar tu aplicación para producción

```
npm run build
```

Configurar el Servidor NGINX

Configuración del Servidor: Asegúrate de que el servidor que utilices esté configurado para servir archivos estáticos correctamente. En muchos casos, esto implica configurar el servidor para que maneje las solicitudes de rutas dinámicas utilizando un archivo de redirección o un archivo .htaccess en el caso de Apache.

Hosting de la aplicación en React

Opciones de Hosting Tradicional vs. Moderno

- **Hosting Tradicional:** Usar servidores web como Apache o Nginx para servir los archivos estáticos de la aplicación React.

- **Hosting Moderno:** Plataformas como Netlify, Vercel, y Firebase Hosting están diseñadas específicamente para aplicaciones web modernas, ofreciendo despliegue continuo y optimización automática de los recursos.

Servicios de Hosting para Aplicaciones React: Netlify y Vercel

Netlify y Vercel son dos de las plataformas de hosting moderno más populares para aplicaciones React. Ambas están diseñadas para facilitar el despliegue continuo y ofrecen una serie de características útiles para desarrolladores.

Vercel: Despliegue para Aplicaciones Modernas

Vercel es otra plataforma que simplifica el despliegue de aplicaciones React y está optimizada para proyectos basados en JavaScript, como Next.js. Al igual que Netlify, permite conectar tu repositorio y desplegar automáticamente con cada actualización de código.

Practica Final de Modulo

Integrar el proyecto del modulo 2 al proyecto-react-2
https://github.com/jairojumbo/proyecto-react-2.git

Usar la Aplicación y modificar Page1 o Page2 para ahí subir el proyecto del modulo 2

Instalar surge.sh
npm install –global surge

Hacer la integración Continua (CI/CD) con Git Actions + Surge

Rubrica de Calificacion

Criterio	Excelente (5)	Bueno (4)	Aceptable (3)	Insuficiente (1-2)
1. Integración del proyecto del Módulo 2	Proyecto del Módulo 2 completamente integrado en Page1 o Page2, funcional y adaptado al estilo.	Proyecto integrado pero con detalles de estilo o estructura no adaptados completamente.	Proyecto visible pero con errores funcionales o integración parcial.	No se integró el proyecto o no es visible.
2. Uso de React y estructura del proyecto	Uso correcto y organizado de componentes, hooks, rutas y props. Código limpio y modular.	Uso mayormente correcto con algunos detalles de estructura o estilo de código.	Uso básico o limitado de React, con estructuras poco organizadas.	Código desorganizado, no se sigue la estructura recomendada de React.
3. Configuración de CI/CD con GitHub Actions	CI/CD completamente funcional: lint/build/test/deploy automáticos al hacer push/pull request.	CI/CD configurado y funcionando, pero con pasos mínimos (ej. solo deploy).	Configuración incompleta o falla en algún paso del flujo.	No se configuró ningún tipo de integración continua.
4. Despliegue exitoso en Surge.sh	App desplegada correctamente en Surge, accesible y actualizada con cada push a main.	App desplegada correctamente, pero sin automatización con GitHub Actions.	App desplegada manualmente o con errores ocasionales.	No hay despliegue funcional en Surge.sh.
5. Uso de Git y buenas prácticas de versionado	Commits frecuentes, claros y descriptivos. Buen manejo de ramas y pull requests.	Commits regulares con mensajes comprensibles, uso mínimo de ramas.	Uso limitado de Git: pocos commits, sin mensajes claros o ramas.	No hay evidencia de uso de Git de forma adecuada.
6. Documentación del proyecto (README.md)	Documentación completa: descripción, instalación, uso, CI/CD, despliegue y enlaces.	Documentación clara pero incompleta en uno o dos aspectos.	Documentación mínima (solo título y descripción básica).	No hay README o es irrelevante.
7. Calidad visual y experiencia de usuario (UI/UX)	Interfaz pulida, coherente, con buena usabilidad y adaptada a diferentes tamaños de pantalla.	Interfaz visualmente buena, pero con pequeños problemas de UX o diseño no responsivo.	Interfaz funcional pero poco estética o sin cuidado de la experiencia del usuario.	Interfaz básica o difícil de usar, sin atención al diseño o accesibilidad.

GRACIAS