Parte 2 FrontEnd

1. Instalar emotion/react y emotion/styled

¿Qué es Styled Components? Styled Components es una librería que te permite escribir estilos CSS directamente en tus componentes de React usando JavaScript. En este caso, usaremos @emotion/styled, que es una alternativa popular a la librería styled-components original, y @emotion/react para soportar características avanzadas como temas.

Paso 1: Instalar las dependencias Asegúrate de estar en el directorio de tu proyecto (my-react-app). En tu terminal, ejecuta:

```
npm install @emotion/react @emotion/styled
```

Esto instalará las dos librerías necesarias:

- @emotion/react: Proporciona utilidades como el soporte para temas y el componente css.
- @emotion/styled: Permite crear componentes estilizados con una sintaxis similar a CSS.

Paso 2: Verificar la instalación Asegúrate de que las dependencias se hayan agregado a tu package.json. Deberías ver algo como:

```
"dependencies": {
  "@emotion/react": "^11.11.1",
  "@emotion/styled": "^11.11.0",
  // otras dependencias
}
```

---

2. Creación Styled Título

¿Qué es un Styled Component? Un styled component es un componente de React que tiene estilos asociados directamente. En lugar de usar archivos CSS separados, defines los estilos junto con el componente.

Ejemplo Práctico Vamos a estilizar el título de nuestro componente Header usando @emotion/styled.

Modifica src/components/Header.jsx:

```
import styled from '@emotion/styled';

const StyledTitle = styled.h1`
  font-size: 2.5rem;
  color: #2c3e50;
  text-align: center;
  margin-bottom: 0.5rem;
  text-transform: uppercase;
  letter-spacing: 2px;
```

```
`;

function Header() {
  return (
    <header>
      <StyledTitle>Gestor de Tareas</StyledTitle>
      <p>Bienvenido a tu aplicación de tareas en React 2025</p>
    </header>
  );
}

export default Header;
```

Explicación:

- Importamos styled desde @emotion/styled.
- Creamos StyledTitle usando styled.h1. Esto genera un componente <h1> con los estilos definidos dentro de las comillas de template literals (`).
- Los estilos se escriben como CSS normal, pero están encapsulados: solo afectan a este componente.
- Usamos StyledTitle como si fuera un componente React normal (<StyledTitle>).

---

3. Creación de Div Círculo con Texto Dentro Usando Styled

Creando un Círculo Estilizado Vamos a crear un componente que muestre un círculo con un número dentro, para representar la cantidad de tareas completadas en TaskCounter.

Modifica src/components/TaskCounter.jsx:

```
import { useState, useEffect } from 'react';
import styled from '@emotion/styled';

const Circle = styled.div`
  width: 60px;
  height: 60px;
  background-color: #3498db;
  border-radius: 50%;
  display: flex;
  justify-content: center;
  align-items: center;
  color: white;
  font-size: 1.5rem;
  font-weight: bold;
```

```
  margin: 0 auto 1rem;
`;

function TaskCounter() {
  const [completedTasks, setCompletedTasks] = useState(0);

  useEffect(() => {
    console.log('Componente montado');
    setTimeout(() => {
      setCompletedTasks(5);
    }, 1000);

    return () => {
      console.log('Componente desmontado');
      localStorage.setItem('completedTasks', completedTasks);
    };
  }, []);

  useEffect(() => {
    console.log('El contador de tareas ha cambiado:', completedTasks);
  }, [completedTasks]);

  const handleIncrement = () => {
    setCompletedTasks(completedTasks + 1);
  };

  return (
    <div>
      <h2>Tareas Completadas:</h2>
      <Circle>{completedTasks}</Circle>
      <button onClick={handleIncrement}>Marcar Tarea como Completada</button>
    </div>
  );
}

export default TaskCounter;
```

Explicación:

- Creamos Circle usando styled.div, que genera un estilizado.
- Usamos propiedades CSS para hacer un círculo:

    - width y height definen el tamaño.
    - border-radius: 50% hace que el div sea circular.
    - display: flex con justify-content y align-items centra el texto dentro del círculo.
- El número de tareas completadas (completedTasks) se muestra dentro del círculo.

---

4. Creación de Triángulo con Diseño CSS Usando Styled

Creando un Triángulo Vamos a agregar un triángulo decorativo al componente TaskItem para indicar visualmente si una tarea está completada o no.

Modifica src/components/TaskItem.jsx:

```
import styled from '@emotion/styled';

const TaskWrapper = styled.li`
  display: flex;
  align-items: center;
  padding: 0.5rem;
  margin: 0.5rem 0;
`;

const Triangle = styled.div`
  width: 0;
  height: 0;
  border-left: 10px solid transparent;
  border-right: 10px solid transparent;
  border-bottom: 10px solid #e74c3c;
  margin-right: 10px;
`;

function TaskItem({ task, id }) {
  return (
    <TaskWrapper>
      <Triangle />
      {task} (ID: {id})
    </TaskWrapper>
  );
}

export default TaskItem;
```

Explicación:

- Creamos TaskWrapper para estilizar el <li> y alinear sus elementos.
- Creamos Triangle usando styled.div. Para hacer un triángulo con CSS:
  - Establecemos width y height a 0.
  - Usamos las propiedades border para crear un triángulo. En este caso, border-bottom define el color y tamaño del triángulo, mientras que border-left y border-right transparentes forman los lados inclinados.
- El triángulo aparece a la izquierda de cada tarea como un indicador visual.

---

5. Creación Input con Styled

Estilizando un Input Vamos a estilizar el input del formulario en TaskForm para que tenga un diseño más atractivo.

Modifica src/components/TaskForm.jsx:

```
import styled from '@emotion/styled';

const Form = styled.form`
  display: flex;
  flex-direction: column;
  gap: 1rem;
  max-width: 400px;
  margin: 0 auto;
`;

const StyledInput = styled.input`
  padding: 0.75rem;
  font-size: 1rem;
  border: 2px solid #3498db;
  border-radius: 5px;
  outline: none;
  transition: border-color 0.3s ease;

  &:focus {
    border-color: #2980b9;
  }

  &::placeholder {
    color: #95a5a6;
  }
```

```
`;

const Button = styled.button`
  padding: 0.75rem;
  font-size: 1rem;
  background-color: #2ecc71;
  color: white;
  border: none;
  border-radius: 5px;
  cursor: pointer;
  transition: background-color 0.3s ease;

  &:hover {
    background-color: #27ae60;
  }
`;

function TaskForm({ onAddTask }) {
  const [taskText, setTaskText] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    if (taskText.trim()) {
      onAddTask(taskText);
      setTaskText('');
    }
  };

  return (
    <Form onSubmit={handleSubmit}>
      <h2>Agregar Nueva Tarea</h2>
      <StyledInput
        type="text"
        value={taskText}
        onChange={(e) => setTaskText(e.target.value)}
        placeholder="Escribe una tarea"
      />
      <Button type="submit">Agregar</Button>
    </Form>
  );
```

```
}

export default TaskForm;
```

Explicación:

- Creamos StyledInput para estilizar el <input>:
  - padding y font-size mejoran la legibilidad.
  - border y border-radius le dan un diseño moderno.
  - &:focus cambia el color del borde cuando el input está enfocado.
  - &::placeholder estiliza el texto del placeholder.
- También estilizamos el formulario (Form) y el botón (Button) para que el diseño sea coherente.
- Usamos pseudo-clases como &:hover para agregar interactividad.

---

6. Props en Styled

Usando Props en Styled Components Puedes pasar props a un styled component y usarlas para modificar los estilos dinámicamente.

Ejemplo Práctico Vamos a modificar el componente TaskItem para que el triángulo cambie de color según si la tarea está completada o no. Agregaremos una prop completed a TaskItem.

Primero, modifica src/components/TaskList.jsx para pasar una prop completed:

```
import { useState } from 'react';
import TaskItem from './TaskItem';
import TaskForm from './TaskForm';
import Card from './Card';

function TaskList() {
  const [tasks, setTasks] = useState([
    { id: 1, text: 'Aprender React', completed: false },
    { id: 2, text: 'Hacer un proyecto', completed: true },
    { id: 3, text: 'Descansar', completed: false },
  ]);
  const [view, setView] = useState('list');

  const addTask = (taskText) => {
    const newTask = { id: tasks.length + 1, text: taskText, completed: false
};
    setTasks([...tasks, newTask]);
    setView('list');
  };
```

```
const EmptyMessage = () => <p>No hay tareas disponibles.</p>;
const TaskListView = () => (
  <>
    <h2>Lista de Tareas</h2>
    <button onClick={() => setView('form')}>Agregar Tarea</button>
    <ul>
      {tasks.map((task) => (
        <TaskItem
          key={task.id}
          id={task.id}
          task={task.text}
          completed={task.completed}
        />
      ))}
    </ul>
  </>
);

const renderView = () => {
  switch (view) {
    case 'list':
      return (
        <Card>
          {tasks.length === 0 ? <EmptyMessage /> : <TaskListView />}
        </Card>
      );
    case 'form':
      return (
        <Card>
          <button onClick={() => setView('list')}>Volver a la Lista</button>
          <TaskForm onAddTask={addTask} />
        </Card>
      );
    default:
      return <p>Vista no encontrada</p>;
  }
};

return <div>{renderView()}</div>;
```

```
}

export default TaskList;
```

Ahora, modifica src/components/TaskItem.jsx para usar la prop completed en los estilos:

```
import styled from '@emotion/styled';

const TaskWrapper = styled.li`
  display: flex;
  align-items: center;
  padding: 0.5rem;
  margin: 0.5rem 0;
  text-decoration: ${(props) => (props.completed ? 'line-through' : 'none')};
`;

const Triangle = styled.div`
  width: 0;
  height: 0;
  border-left: 10px solid transparent;
  border-right: 10px solid transparent;
  border-bottom: 10px solid ${(props) => (props.completed ? '#2ecc71' :
'#e74c3c')};
  margin-right: 10px;
`;

function TaskItem({ task, id, completed }) {
  return (
    <TaskWrapper completed={completed}>
      <Triangle completed={completed} />
      {task} (ID: {id})
    </TaskWrapper>
  );
}

export default TaskItem;
```

Explicación:

- Pasamos la prop completed a TaskItem desde TaskList.
- En TaskWrapper, usamos text-decoration para tachar el texto si completed es true.
- En Triangle, cambiamos el color del triángulo según el valor de completed:

    - Verde (#2ecc71) si está completada.
    - Rojo (#e74c3c) si no está completada.
- Los styled components pueden acceder a las props mediante una función que recibe props como argumento.

---

7. Guardando Styled en Archivo Externo Reutilizable

Creando un Archivo de Estilos Reutilizables Para mantener el código organizado y reutilizable, podemos mover los styled components a un archivo separado.

Paso 1: Crear un archivo de estilos Crea un nuevo archivo src/styles/StyledComponents.js:

```
import styled from '@emotion/styled';

// Reutilizable para títulos
export const StyledTitle = styled.h1`
  font-size: 2.5rem;
  color: #2c3e50;
  text-align: center;
  margin-bottom: 0.5rem;
  text-transform: uppercase;
  letter-spacing: 2px;
`;

// Reutilizable para círculos
export const Circle = styled.div`
  width: 60px;
  height: 60px;
  background-color: #3498db;
  border-radius: 50%;
  display: flex;
  justify-content: center;
  align-items: center;
  color: white;
  font-size: 1.5rem;
  font-weight: bold;
  margin: 0 auto 1rem;
`;

// Reutilizable para formularios
export const Form = styled.form`
```

```
  display: flex;
  flex-direction: column;
  gap: 1rem;
  max-width: 400px;
  margin: 0 auto;
`;

export const StyledInput = styled.input`
  padding: 0.75rem;
  font-size: 1rem;
  border: 2px solid #3498db;
  border-radius: 5px;
  outline: none;
  transition: border-color 0.3s ease;

  &:focus {
    border-color: #2980b9;
  }

  &::placeholder {
    color: #95a5a6;
  }
`;

export const Button = styled.button`
  padding: 0.75rem;
  font-size: 1rem;
  background-color: #2ecc71;
  color: white;
  border: none;
  border-radius: 5px;
  cursor: pointer;
  transition: background-color 0.3s ease;

  &:hover {
    background-color: #27ae60;
  }
`;
```

Paso 2: Usar los estilos reutilizables Ahora, actualiza los componentes para importar estos estilos.

- **Header.jsx**:

```
import { StyledTitle } from '../styles/StyledComponents';

function Header() {
  return (
    <header>
      <StyledTitle>Gestor de Tareas</StyledTitle>
      <p>Bienvenido a tu aplicación de tareas en React 2025</p>
    </header>
  );
}

export default Header;
```

- **TaskCounter.jsx**:

```
import { useState, useEffect } from 'react';
import { Circle } from '../styles/StyledComponents';

function TaskCounter() {
  const [completedTasks, setCompletedTasks] = useState(0);

  useEffect(() => {
    console.log('Componente montado');
    setTimeout(() => {
      setCompletedTasks(5);
    }, 1000);

    return () => {
      console.log('Componente desmontado');
      localStorage.setItem('completedTasks', completedTasks);
    };
  }, []);

  useEffect(() => {
    console.log('El contador de tareas ha cambiado:', completedTasks);
  }, [completedTasks]);

  const handleIncrement = () => {
    setCompletedTasks(completedTasks + 1);
```

```
  };

  return (
    <div>
      <h2>Tareas Completadas:</h2>
      <Circle>{completedTasks}</Circle>
      <button onClick={handleIncrement}>Marcar Tarea como Completada</button>
    </div>
  );
}

export default TaskCounter;
```

- **TaskForm.jsx**:

```
import { Form, StyledInput, Button } from '../styles/StyledComponents';

function TaskForm({ onAddTask }) {
  const [taskText, setTaskText] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    if (taskText.trim()) {
      onAddTask(taskText);
      setTaskText('');
    }
  };

  return (
    <Form onSubmit={handleSubmit}>
      <h2>Agregar Nueva Tarea</h2>
      <StyledInput
        type="text"
        value={taskText}
        onChange={(e) => setTaskText(e.target.value)}
        placeholder="Escribe una tarea"
      />
      <Button type="submit">Agregar</Button>
    </Form>
  );
```

```
}

export default TaskForm;
```

Explicación:

- Movimos los styled components a un archivo separado (StyledComponents.js).
- Exportamos cada styled component para que pueda ser importado y reutilizado en cualquier parte del proyecto.
- Esto mejora la organización del código y permite que los estilos sean consistentes en toda la aplicación.

Resumen hasta el momento:

Hemos cubierto todos los temas relacionados con **Styled Components** en React 2025, integrándolos en nuestra aplicación de gestión de tareas:

1. **Instalar emotion/react y emotion/styled**: Instalamos las librerías necesarias.
2. **Creación Styled Título**: Estilizamos el título del Header con un styled component.
3. **Creación de Div Círculo con Texto Dentro Usando Styled**: Creamos un círculo para mostrar el contador de tareas.
4. **Creación de Triángulo con Diseño CSS Usando Styled**: Agregamos un triángulo decorativo a TaskItem.
5. **Creación Input con Styled**: Estilizamos el input y el formulario en TaskForm.
6. **Props en Styled**: Usamos props para cambiar dinámicamente los estilos del triángulo y el texto en TaskItem.
7. **Guardando Styled en Archivo Externo Reutilizable**: Movimos los styled components a un archivo separado para reutilizarlos.

8. Recordar Configuración Inicial del Proyecto (si no lo tienes)

Si no tienes un proyecto React configurado, sigue estos pasos para crear uno con Vite:

Paso 1: Crear el proyecto Abre tu terminal y ejecuta:

```
npm create vite@latest my-react-app -- --template react
cd my-react-app
npm install
```

Paso 2: Iniciar el servidor Ejecuta:

```
npm run dev
```

Abre http://localhost:5173 en tu navegador para ver la aplicación inicial.

Paso 3: Limpiar el proyecto Edita src/App.jsx para que quede limpio:

```
function App() {
```

```
  return (
    <div>
      <h1>Mi Aplicación de Tareas</h1>
    </div>
  );
}

export default App;
```

Limpia también src/index.css si deseas empezar desde cero.

---

9. Instalación de React Router DOM

¿Qué es React Router DOM? React Router DOM es una librería que permite manejar la navegación y el enrutamiento en aplicaciones React de una sola página (SPA). Nos permite definir rutas, navegar entre páginas y manejar parámetros de URL, todo sin recargar la página.

Paso 1: Instalar React Router DOM En tu terminal, dentro del directorio del proyecto, ejecuta:

```
npm install react-router-dom
```

Esto instalará la versión más reciente de react-router-dom (en 2025, probablemente estemos usando la versión 6.x o superior). Verifica que se haya agregado a tu package.json:

```
"dependencies": {
  "react-router-dom": "^6.22.0",
  // otras dependencias
}
```

Paso 2: Verificar la instalación No necesitas hacer nada más por ahora; simplemente asegúrate de que la instalación se completó sin errores.

---

10. Creación Componentes de Página

Creando Páginas Para usar React Router, necesitamos crear componentes que representen las diferentes páginas de nuestra aplicación. Vamos a crear tres páginas: una página principal (Home), una página para la lista de tareas (Tasks), y una página para agregar tareas (AddTask).

Paso 1: Crear los componentes de página Crea una carpeta src/pages y dentro de ella los siguientes archivos:

- **src/pages/Home.jsx**:

```
function Home() {
```

```
  return (
    <div>
      <h1>Bienvenido a la Aplicación de Tareas</h1>
      <p>Esta es la página principal.</p>
    </div>
  );
}

export default Home;
```

- **src/pages/Tasks.jsx**: Vamos a reutilizar nuestro componente TaskList que ya tenemos, pero lo adaptaremos como una página.

```
import TaskList from '../components/TaskList';

function Tasks() {
  return (
    <div>
      <h1>Mis Tareas</h1>
      <TaskList />
    </div>
  );
}

export default Tasks;
```

- **src/pages/AddTask.jsx**: Vamos a reutilizar nuestro componente TaskForm, pero lo adaptaremos como una página.

```
import TaskForm from '../components/TaskForm';

function AddTask() {
  const handleAddTask = (taskText) => {
    // Por ahora, solo mostramos la tarea en la consola
    console.log('Tarea agregada:', taskText);
  };

  return (
    <div>
      <h1>Agregar Nueva Tarea</h1>
      <TaskForm onAddTask={handleAddTask} />
```

```
      </div>
    );
  }

  export default AddTask;
```

Paso 2: Asegurarnos de que los componentes reutilizados existan Si no tienes los componentes TaskList y TaskForm de las respuestas anteriores, aquí te los proporciono de forma simplificada:

- **src/components/TaskList.jsx**:

```
import { useState } from 'react';
import TaskItem from './TaskItem';

function TaskList() {
  const [tasks] = useState([
    { id: 1, text: 'Aprender React', completed: false },
    { id: 2, text: 'Hacer un proyecto', completed: true },
    { id: 3, text: 'Descansar', completed: false },
  ]);

  return (
    <div>
      <ul>
        {tasks.map((task) => (
          <TaskItem
            key={task.id}
            id={task.id}
            task={task.text}
            completed={task.completed}
          />
        ))}
      </ul>
    </div>
  );
}

export default TaskList;
```

- **src/components/TaskItem.jsx**:

```
function TaskItem({ task, id, completed }) {
  return (
    <li style={{ textDecoration: completed ? 'line-through' : 'none' }}>
      {task} (ID: {id})
    </li>
  );
}

export default TaskItem;
```

- **src/components/TaskForm.jsx**:

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

function TaskForm({ onAddTask }) {
  const [taskText, setTaskText] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    if (taskText.trim()) {
      onAddTask(taskText);
      setTaskText('');
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={taskText}
        onChange={(e) => setTaskText(e.target.value)}
        placeholder="Escribe una tarea"
      />
      <button type="submit">Agregar</button>
    </form>
  );
}

export default TaskForm;
```

Explicación:

- Cada componente de página (Home, Tasks, AddTask) representa una vista diferente de nuestra aplicación.
- Reutilizamos componentes como TaskList y TaskForm para mantener el código modular.

---

11. BrowserRouter, Routes y Route

Configurando el Enrutamiento Vamos a usar BrowserRouter, Routes y Route para definir las rutas de nuestra aplicación.

Paso 1: Configurar BrowserRouter BrowserRouter es el componente que envuelve toda la aplicación y habilita el enrutamiento basado en el historial del navegador.

Modifica src/main.jsx:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import { BrowserRouter } from 'react-router-dom';
import App from './App.jsx';
import './index.css';

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </React.StrictMode>
);
```

Paso 2: Definir las rutas en App.jsx Ahora, en App.jsx, usaremos Routes y Route para mapear las URLs a los componentes de página.

Modifica src/App.jsx:

```
import { Routes, Route } from 'react-router-dom';
import Home from './pages/Home';
import Tasks from './pages/Tasks';
import AddTask from './pages/AddTask';

function App() {
  return (
    <div>
      <h1>Mi Aplicación de Tareas</h1>
      <Routes>
```

```
        <Route path="/" element={<Home />} />
        <Route path="/tasks" element={<Tasks />} />
        <Route path="/add-task" element={<AddTask />} />
      </Routes>
    </div>
  );
}

export default App;
```

Paso 3: Agregar navegación Para navegar entre las páginas, vamos a agregar enlaces usando el componente Link de React Router. Modifica App.jsx para incluir una barra de navegación:

```
import { Routes, Route, Link } from 'react-router-dom';
import Home from './pages/Home';
import Tasks from './pages/Tasks';
import AddTask from './pages/AddTask';

function App() {
  return (
    <div>
      <nav style={{ marginBottom: '20px' }}>
        <ul style={{ listStyle: 'none', display: 'flex', gap: '20px' }}>
          <li>
            <Link to="/">Inicio</Link>
          </li>
          <li>
            <Link to="/tasks">Tareas</Link>
          </li>
          <li>
            <Link to="/add-task">Agregar Tarea</Link>
          </li>
        </ul>
      </nav>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/tasks" element={<Tasks />} />
        <Route path="/add-task" element={<AddTask />} />
      </Routes>
    </div>
```

```
  );
}

export default App;
```

Explicación:

- BrowserRouter envuelve toda la aplicación y habilita el enrutamiento.
- Routes es un contenedor que define todas las rutas de la aplicación.
- Route mapea una URL (path) a un componente (element). Por ejemplo, path="/tasks" renderiza el componente Tasks.
- Link genera enlaces que permiten navegar entre rutas sin recargar la página. Se traduce a una etiqueta <a>, pero evita la recarga completa del navegador.
- Ahora puedes navegar entre las páginas haciendo clic en los enlaces.

---

12. Error 404 Personalizado (con BrowserRouter)

Creando una Página de Error 404 Si el usuario accede a una ruta que no existe (por ejemplo, /ruta-inexistente), queremos mostrar una página de error personalizada.

Paso 1: Crear el componente de error Crea src/pages/NotFound.jsx:

```
function NotFound() {
  return (
    <div>
      <h1>Error 404 - Página No Encontrada</h1>
      <p>Lo sentimos, la página que estás buscando no existe.</p>
      <Link to="/">Volver al Inicio</Link>
    </div>
  );
}

export default NotFound;
```

Paso 2: Agregar la ruta 404 Modifica src/App.jsx para incluir una ruta "catch-all" que maneje las URLs no definidas:

```
import { Routes, Route, Link } from 'react-router-dom';
import Home from './pages/Home';
import Tasks from './pages/Tasks';
import AddTask from './pages/AddTask';
import NotFound from './pages/NotFound';
```

```jsx
function App() {
  return (
    <div>
      <nav style={{ marginBottom: '20px' }}>
        <ul style={{ listStyle: 'none', display: 'flex', gap: '20px' }}>
          <li>
            <Link to="/">Inicio</Link>
          </li>
          <li>
            <Link to="/tasks">Tareas</Link>
          </li>
          <li>
            <Link to="/add-task">Agregar Tarea</Link>
          </li>
        </ul>
      </nav>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/tasks" element={<Tasks />} />
        <Route path="/add-task" element={<AddTask />} />
        <Route path="*" element={<NotFound />} />
      </Routes>
    </div>
  );
}

export default App;
```

Explicación:

- La ruta path="*" actúa como un "catch-all": captura cualquier URL que no coincida con las rutas definidas anteriormente.
- Cuando el usuario accede a una ruta no existente (por ejemplo, /ruta-inexistente), se renderiza el componente NotFound.
- Incluimos un enlace para que el usuario pueda volver a la página principal.

Prueba:

- Ve a http://localhost:5173/ruta-inexistente y verifica que se muestre la página de error 404.

13. React Router DOM con createBrowserRouter y RouterProvider

En React Router DOM 6.x, se introdujo una nueva API basada en createBrowserRouter y RouterProvider, que es más moderna y ofrece mejor soporte para características avanzadas como la carga de datos (data loading). Vamos a reconfigurar nuestra aplicación para usar esta API.

Configuración createBrowserRouter y RouterProvider

¿Qué es createBrowserRouter y RouterProvider? createBrowserRouter es una función que te permite definir las rutas de tu aplicación de manera programática, en lugar de usar componentes JSX como Routes y Route. RouterProvider es el componente que conecta el router creado con tu aplicación.

Paso 1: Crear las rutas con createBrowserRouter Modifica src/main.jsx para usar createBrowserRouter y RouterProvider:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import { createBrowserRouter, RouterProvider } from 'react-router-dom';
import Home from './pages/Home';
import Tasks from './pages/Tasks';
import AddTask from './pages/AddTask';
import NotFound from './pages/NotFound';
import './index.css';

// Definir las rutas
const router = createBrowserRouter([
  {
    path: '/',
    element: <Home />,
  },
  {
    path: '/tasks',
    element: <Tasks />,
  },
  {
    path: '/add-task',
    element: <AddTask />,
  },
  {
    path: '*',
    element: <NotFound />,
  },
]);
```

```
ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <RouterProvider router={router} />
  </React.StrictMode>
);
```

Paso 2: Actualizar App.jsx para eliminar el enrutamiento Dado que ahora las rutas están definidas en main.jsx, App.jsx ya no necesita manejar el enrutamiento. Sin embargo, vamos a usarlo como un componente de layout más adelante. Por ahora, simplifiquémoslo:

```
function App() {
  return (
    <div>
      <h1>Mi Aplicación de Tareas</h1>
    </div>
  );
}

export default App;
```

Explicación:

- createBrowserRouter toma un array de objetos, donde cada objeto define una ruta (path) y el componente a renderizar (element).
- RouterProvider recibe el router creado y lo conecta con la aplicación.
- Ya no necesitamos BrowserRouter, Routes ni Route en App.jsx, porque el enrutamiento ahora se maneja en main.jsx.

Nota: En este punto, las rutas seguirán funcionando, pero no tenemos una barra de navegación. Vamos a agregarla en el siguiente paso usando un layout.

---

14. Creación de Layout y Outlet

¿Qué es un Layout y Outlet? Un layout es un componente que define una estructura común para varias páginas (por ejemplo, una barra de navegación). Outlet es un componente de React Router que indica dónde se renderizarán los componentes de las rutas hijas.

Paso 1: Crear un componente de layout Vamos a usar App.jsx como nuestro componente de layout, y agregaremos una barra de navegación.

Modifica src/App.jsx:

```
import { Outlet, Link } from 'react-router-dom';
```

```
function App() {
  return (
    <div>
      <nav style={{ marginBottom: '20px' }}>
        <ul style={{ listStyle: 'none', display: 'flex', gap: '20px' }}>
          <li>
            <Link to="/">Inicio</Link>
          </li>
          <li>
            <Link to="/tasks">Tareas</Link>
          </li>
          <li>
            <Link to="/add-task">Agregar Tarea</Link>
          </li>
        </ul>
      </nav>
      <Outlet />
    </div>
  );
}

export default App;
```

Paso 2: Actualizar las rutas para usar el layout Modifica src/main.jsx para usar App como layout y definir las rutas como rutas hijas:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import { createBrowserRouter, RouterProvider } from 'react-router-dom';
import App from './App';
import Home from './pages/Home';
import Tasks from './pages/Tasks';
import AddTask from './pages/AddTask';
import NotFound from './pages/NotFound';
import './index.css';

const router = createBrowserRouter([
  {
    element: <App />,
```

```
    children: [
      {
        path: '/',
        element: <Home />,
      },
      {
        path: '/tasks',
        element: <Tasks />,
      },
      {
        path: '/add-task',
        element: <AddTask />,
      },
      {
        path: '*',
        element: <NotFound />,
      },
    ],
  },
]);

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <RouterProvider router={router} />
  </React.StrictMode>
);
```

Explicación:

- App ahora actúa como un layout que incluye la barra de navegación y un <Outlet />.
- Outlet es donde se renderizarán los componentes de las rutas hijas (Home, Tasks, AddTask, NotFound).
- En main.jsx, definimos App como el componente principal (element) y las rutas específicas como children.
- Esto asegura que todas las páginas tengan la barra de navegación, y el contenido de cada página se renderice en el lugar del Outlet.

---

15. NavLink, Link

¿Qué son Link y NavLink?

- Link es un componente que genera un enlace para navegar entre rutas sin recargar la página.
- NavLink es similar a Link, pero agrega la capacidad de aplicar estilos o clases cuando la ruta está activa.

Paso 1: Usar NavLink en la barra de navegación Modifica src/App.jsx para usar NavLink y resaltar la ruta activa:

```
import { Outlet, NavLink } from 'react-router-dom';


function App() {
  return (
    <div>
      <nav style={{ marginBottom: '20px' }}>
        <ul style={{ listStyle: 'none', display: 'flex', gap: '20px' }}>
          <li>
            <NavLink
              to="/"
              style={({ isActive }) => ({
                color: isActive ? '#e74c3c' : '#2c3e50',
                textDecoration: 'none',
                fontWeight: isActive ? 'bold' : 'normal',
              })}
            >
              Inicio
            </NavLink>
          </li>
          <li>
            <NavLink
              to="/tasks"
              style={({ isActive }) => ({
                color: isActive ? '#e74c3c' : '#2c3e50',
                textDecoration: 'none',
                fontWeight: isActive ? 'bold' : 'normal',
              })}
            >
              Tareas
            </NavLink>
          </li>
          <li>
            <NavLink
              to="/add-task"
              style={({ isActive }) => ({
                color: isActive ? '#e74c3c' : '#2c3e50',
                textDecoration: 'none',
```

```
                fontWeight: isActive ? 'bold' : 'normal',
              })}
            >
              Agregar Tarea
            </NavLink>
          </li>
        </ul>
      </nav>
      <Outlet />
    </div>
  );
}

export default App;
```

Paso 2: Usar Link en las páginas Ya usamos Link en NotFound.jsx para volver al inicio. Vamos a agregar un enlace en Tasks.jsx para ver los detalles de una tarea (lo usaremos más adelante con parámetros):

Modifica src/pages/Tasks.jsx:

```
import { Link } from 'react-router-dom';
import TaskList from '../components/TaskList';

function Tasks() {
  return (
    <div>
      <h1>Mis Tareas</h1>
      <TaskList />
      <p>
        <Link to="/tasks/1">Ver detalles de la tarea 1</Link>
      </p>
    </div>
  );
}

export default Tasks;
```

Explicación:

- NavLink recibe una función en la prop style que devuelve estilos dinámicos según el estado isActive. Cuando la ruta coincide con el enlace, el texto se vuelve rojo y negrita.
- También puedes usar la prop className en lugar de style para aplicar clases CSS:

```
<NavLink
  to="/"
  className={({ isActive }) => (isActive ? 'active-link' : '')}
>
  Inicio
</NavLink>
```

- Link se usa para navegación simple, como en Tasks.jsx y NotFound.jsx.

---

16. Parámetros Path (useParams)

¿Qué es useParams? useParams es un Hook de React Router que te permite acceder a los parámetros dinámicos de la URL (por ejemplo, /tasks/:id).

Paso 1: Crear una página para los detalles de una tarea Crea src/pages/TaskDetail.jsx:

```
import { useParams } from 'react-router-dom';

function TaskDetail() {
  const { id } = useParams();

  // Simulamos una tarea obtenida de una base de datos
  const task = {
    id,
    text: `Tarea ${id}`,
    description: `Descripción de la tarea ${id}`,
    completed: false,
  };

  return (
    <div>
      <h1>Detalles de la Tarea</h1>
      <h2>{task.text}</h2>
      <p>ID: {task.id}</p>
      <p>Descripción: {task.description}</p>
      <p>Estado: {task.completed ? 'Completada' : 'Pendiente'}</p>
    </div>
  );
}
```

```
export default TaskDetail;
```

Paso 2: Agregar la ruta dinámica Modifica src/main.jsx para incluir la ruta dinámica /tasks/:id:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import { createBrowserRouter, RouterProvider } from 'react-router-dom';
import App from './App';
import Home from './pages/Home';
import Tasks from './pages/Tasks';
import AddTask from './pages/AddTask';
import TaskDetail from './pages/TaskDetail';
import NotFound from './pages/NotFound';
import './index.css';

const router = createBrowserRouter([
  {
    element: <App />,
    children: [
      {
        path: '/',
        element: <Home />,
      },
      {
        path: '/tasks',
        element: <Tasks />,
      },
      {
        path: '/tasks/:id',
        element: <TaskDetail />,
      },
      {
        path: '/add-task',
        element: <AddTask />,
      },
      {
        path: '*',
        element: <NotFound />,
      },
    ],
```

```
  },
]);

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <RouterProvider router={router} />
  </React.StrictMode>
);
```

Paso 3: Actualizar TaskList para usar parámetros Modifica src/components/TaskList.jsx para que cada tarea enlace a su página de detalles:

```
import { useState } from 'react';
import { Link } from 'react-router-dom';
import TaskItem from './TaskItem';

function TaskList() {
  const [tasks] = useState([
    { id: 1, text: 'Aprender React', completed: false },
    { id: 2, text: 'Hacer un proyecto', completed: true },
    { id: 3, text: 'Descansar', completed: false },
  ]);

  return (
    <div>
      <ul>
        {tasks.map((task) => (
          <li key={task.id}>
            <TaskItem id={task.id} task={task.text} completed={task.completed} />
            <Link to={`/tasks/${task.id}`}>Ver Detalles</Link>
          </li>
        ))}
      </ul>
    </div>
  );
}

export default TaskList;
```

Explicación:

- La ruta /tasks/:id define un parámetro dinámico :id.
- useParams en TaskDetail extrae el valor de :id de la URL (por ejemplo, si la URL es /tasks/1, id será "1").
- En TaskList, generamos enlaces dinámicos para cada tarea usando Link y el ID de la tarea.

Prueba:

- Ve a /tasks, haz clic en "Ver Detalles" de una tarea, y verifica que te lleve a /tasks/1 (o el ID correspondiente) y muestre los detalles.

---

17. Parámetros Querystring (useLocation, URLSearchParams)

¿Qué son useLocation y URLSearchParams?

- useLocation es un Hook que te da acceso al objeto de ubicación actual, incluyendo la querystring (por ejemplo, ?sort=asc).
- URLSearchParams es una API del navegador que te permite parsear y manipular los parámetros de la querystring.

Paso 1: Agregar parámetros de querystring a Tasks Vamos a permitir que la página Tasks ordene las tareas según un parámetro de querystring (?sort=asc o ?sort=desc).

Modifica src/pages/Tasks.jsx:

```
import { useLocation } from 'react-router-dom';
import TaskList from '../components/TaskList';

function Tasks() {
  const location = useLocation();
  const queryParams = new URLSearchParams(location.search);
  const sort = queryParams.get('sort') || 'asc'; // Por defecto, orden ascendente

  return (
    <div>
      <h1>Mis Tareas</h1>
      <p>Orden: {sort === 'asc' ? 'Ascendente' : 'Descendente'}</p>
      <TaskList sort={sort} />
    </div>
  );
}

export default Tasks;
```

Paso 2: Actualizar TaskList para usar el parámetro sort Modifica src/components/TaskList.jsx:

```jsx
import { useState } from 'react';
import { Link } from 'react-router-dom';
import TaskItem from './TaskItem';

function TaskList({ sort }) {
  const [tasks] = useState([
    { id: 1, text: 'Aprender React', completed: false },
    { id: 2, text: 'Hacer un proyecto', completed: true },
    { id: 3, text: 'Descansar', completed: false },
  ]);

  // Ordenar las tareas según el parámetro sort
  const sortedTasks = [...tasks].sort((a, b) => {
    if (sort === 'asc') {
      return a.text.localeCompare(b.text);
    } else {
      return b.text.localeCompare(a.text);
    }
  });

  return (
    <div>
      <ul>
        {sortedTasks.map((task) => (
          <li key={task.id}>
            <TaskItem id={task.id} task={task.text} completed={task.completed} />
            <Link to={`/tasks/${task.id}`}>Ver Detalles</Link>
          </li>
        ))}
      </ul>
    </div>
  );
}

export default TaskList;
```

Paso 3: Agregar enlaces para cambiar el orden Modifica src/pages/Tasks.jsx para incluir enlaces que cambien el parámetro sort:

```jsx
import { useLocation, Link } from 'react-router-dom';
```

```
import TaskList from '../components/TaskList';

function Tasks() {
  const location = useLocation();
  const queryParams = new URLSearchParams(location.search);
  const sort = queryParams.get('sort') || 'asc';

  return (
    <div>
      <h1>Mis Tareas</h1>
      <p>Orden: {sort === 'asc' ? 'Ascendente' : 'Descendente'}</p>
      <div>
        <Link to="/tasks?sort=asc">Ordenar Ascendente</Link>{' '}
        <Link to="/tasks?sort=desc">Ordenar Descendente</Link>
      </div>
      <TaskList sort={sort} />
    </div>
  );
}

export default Tasks;
```

Explicación:

- useLocation nos da acceso a location.search, que contiene la querystring (por ejemplo, ?sort=asc).
- URLSearchParams parsea la querystring y nos permite obtener el valor de sort con get('sort').
- Pasamos el valor de sort como prop a TaskList, que ordena las tareas en consecuencia.
- Agregamos enlaces para cambiar el valor de sort en la URL.

Prueba:

- Ve a /tasks?sort=asc y verifica que las tareas se ordenen alfabéticamente.
- Ve a /tasks?sort=desc y verifica que se ordenen en orden inverso.

---

18. Página de Error por Ruta

Manejando Errores por Ruta React Router permite definir un componente de error para manejar errores específicos de una ruta (por ejemplo, si falla la carga de datos). Vamos a simular un error en TaskDetail si el ID de la tarea no existe.

Paso 1: Agregar manejo de errores en TaskDetail Modifica src/pages/TaskDetail.jsx:

```
import { useParams } from 'react-router-dom';
```

```
function TaskDetail() {
  const { id } = useParams();

  // Simulamos una tarea obtenida de una base de datos
  const tasks = [
    { id: '1', text: 'Tarea 1', description: 'Descripción de la tarea 1', completed:
false },
    { id: '2', text: 'Tarea 2', description: 'Descripción de la tarea 2', completed: true
},
  ];

  const task = tasks.find((t) => t.id === id);

  if (!task) {
    throw new Error(`Tarea con ID ${id} no encontrada`);
  }

  return (
    <div>
      <h1>Detalles de la Tarea</h1>
      <h2>{task.text}</h2>
      <p>ID: {task.id}</p>
      <p>Descripción: {task.description}</p>
      <p>Estado: {task.completed ? 'Completada' : 'Pendiente'}</p>
    </div>
  );
}

export default TaskDetail;
```

Paso 2: Definir un componente de error por ruta Crea src/pages/ErrorPage.jsx:

```
import { useRouteError } from 'react-router-dom';
import { Link } from 'react-router-dom';

function ErrorPage() {
  const error = useRouteError();

  return (
```

```
    <div>
      <h1>Error</h1>
      <p>{error.message || 'Ocurrió un error inesperado.'}</p>
      <Link to="/tasks">Volver a Tareas</Link>
    </div>
  );
}

export default ErrorPage;
```

Paso 3: Asignar el componente de error a la ruta Modifica src/main.jsx para agregar el manejo de errores a la ruta /tasks/:id:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import { createBrowserRouter, RouterProvider } from 'react-router-dom';
import App from './App';
import Home from './pages/Home';
import Tasks from './pages/Tasks';
import AddTask from './pages/AddTask';
import TaskDetail from './pages/TaskDetail';
import NotFound from './pages/NotFound';
import ErrorPage from './pages/ErrorPage';
import './index.css';

const router = createBrowserRouter([
  {
    element: <App />,
    children: [
      {
        path: '/',
        element: <Home />,
      },
      {
        path: '/tasks',
        element: <Tasks />,
      },
      {
        path: '/tasks/:id',
        element: <TaskDetail />,
```

```
        errorElement: <ErrorPage />,
      },
      {
        path: '/add-task',
        element: <AddTask />,
      },
      {
        path: '*',
        element: <NotFound />,
      },
    ],
  },
]);

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <RouterProvider router={router} />
  </React.StrictMode>
);
```

Explicación:

- En TaskDetail, simulamos un error lanzando una excepción si la tarea no se encuentra.
- La prop errorElement en la ruta /tasks/:id especifica que ErrorPage se renderizará si ocurre un error.
- useRouteError en ErrorPage nos da acceso al error lanzado, y mostramos su mensaje.

Prueba:

- Ve a /tasks/1 y verifica que se muestre la tarea.
- Ve a /tasks/999 (un ID que no existe) y verifica que se muestre la página de error con el mensaje "Tarea con ID 999 no encontrada".

---

19. Error 404 Personalizado (con createBrowserRouter)

Reutilizando la Página 404 Ya definimos una página 404 (NotFound) en la sección anterior, y la estamos usando en nuestra configuración de createBrowserRouter. Sin embargo, vamos a personalizarla un poco más para que sea más visual.

Modifica src/pages/NotFound.jsx:

```
import { Link } from 'react-router-dom';

function NotFound() {
```

```
  return (
    <div style={{ textAlign: 'center', marginTop: '50px' }}>
      <h1 style={{ fontSize: '4rem', color: '#e74c3c' }}>404</h1>
      <h2>Página No Encontrada</h2>
      <p>Lo sentimos, la página que estás buscando no existe.</p>
      <Link
        to="/"
        style={{
          display: 'inline-block',
          marginTop: '20px',
          padding: '10px 20px',
          backgroundColor: '#3498db',
          color: 'white',
          textDecoration: 'none',
          borderRadius: '5px',
        }}
      >
        Volver al Inicio
      </Link>
    </div>
  );
}

export default NotFound;
```

Explicación:

- Mejoramos el diseño de la página 404 con estilos inline (puedes usar Styled Components si lo prefieres).
- La ruta path="*" ya está configurada para renderizar NotFound cuando no se encuentra una ruta.

Prueba:

- Ve a /ruta-inexistente y verifica que se muestre la página 404 personalizada.
