Sección 1: Creación de Layout con Bootstrap

1. Instalación y Configuración de Bootstrap

¿Qué es Bootstrap? Bootstrap es un framework CSS que proporciona componentes y estilos predefinidos para construir interfaces responsivas rápidamente. Incluye un sistema de cuadrícula, botones, formularios, barras de navegación, y más.

Paso 1: Instalar Bootstrap En tu terminal, dentro del directorio del proyecto, ejecuta:

```
npm install bootstrap
```

Esto instalará la versión más reciente de Bootstrap (en 2025, probablemente estemos usando Bootstrap 5.x). Verifica que se haya agregado a tu package.json:

```
"dependencies": {
  "bootstrap": "^5.3.2",
  // otras dependencias
}
```

Paso 2: Importar Bootstrap en el proyecto Para usar los estilos de Bootstrap, necesitamos importar su CSS. Modifica src/main.jsx:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App.jsx';
import 'bootstrap/dist/css/bootstrap.min.css'; // Importar los estilos de Bootstrap
import './index.css';

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

Explicación:

- Importamos el archivo CSS de Bootstrap desde node_modules.
- Esto hace que los estilos de Bootstrap estén disponibles en toda la aplicación.

---

2. Configuración de Bootstrap

Configuración Adicional Bootstrap no requiere mucha configuración adicional si solo usamos sus estilos y componentes predefinidos. Sin embargo, si deseas personalizar Bootstrap (por ejemplo, cambiar los colores del

tema), puedes usar variables de Sass. Para este tutorial, usaremos Bootstrap tal cual, pero te mostraré cómo podrías personalizarlo si lo deseas.

Opción: Personalizar Bootstrap con Sass (Opcional) Si quieres personalizar Bootstrap, primero instala sass:

```
npm install sass
```

Luego, crea un archivo src/styles/custom.scss:

```scss
// src/styles/custom.scss
$primary: #ff6347; // Cambiar el color primario a un tono de rojo
$body-bg: #f8f9fa; // Fondo claro para el cuerpo

// Importar Bootstrap después de definir las variables
@import "../node_modules/bootstrap/scss/bootstrap";
```

Actualiza src/main.jsx para usar el archivo personalizado en lugar del CSS predeterminado:

```jsx
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App.jsx';
import './styles/custom.scss'; // Importar los estilos personalizados
import './index.css';

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

Nota: Para este tutorial, seguiremos usando el CSS predeterminado de Bootstrap (bootstrap.min.css) para mantener las cosas simples.

---

3. Creación de Header y Footer Componentes

Creando el Header Vamos a crear un componente Header que use la barra de navegación de Bootstrap.

Crea src/components/Header.jsx:

```jsx
import { Link } from 'react-router-dom'; // Lo usaremos más adelante con React Router

function Header() {
  return (
```

```
    <nav className="navbar navbar-expand-lg navbar-dark bg-dark">
      <div className="container-fluid">
        <Link className="navbar-brand" to="/">
          Task App
        </Link>
        <button
          className="navbar-toggler"
          type="button"
          data-bs-toggle="collapse"
          data-bs-target="#navbarNav"
          aria-controls="navbarNav"
          aria-expanded="false"
          aria-label="Toggle navigation"
        >
          <span className="navbar-toggler-icon"></span>
        </button>
        <div className="collapse navbar-collapse" id="navbarNav">
          <ul className="navbar-nav">
            <li className="nav-item">
              <Link className="nav-link" to="/">
                Inicio
              </Link>
            </li>
            <li className="nav-item">
              <Link className="nav-link" to="/tasks">
                Tareas
              </Link>
            </li>
            <li className="nav-item">
              <Link className="nav-link" to="/add-task">
                Agregar Tarea
              </Link>
            </li>
          </ul>
        </div>
      </div>
    </nav>
  );
}
```

```
export default Header;
```

Creando el Footer Crea src/components/Footer.jsx:

```
function Footer() {
  return (
    <footer className="bg-dark text-white text-center py-3 mt-auto">
      <p>&copy; 2025 Task App. Todos los derechos reservados.</p>
    </footer>
  );
}

export default Footer;
```

Explicación:

- **Header**: Usamos la clase navbar de Bootstrap para crear una barra de navegación responsiva. Incluye un botón toggler para pantallas pequeñas y enlaces de navegación. Por ahora, los enlaces Link no funcionan porque no hemos configurado React Router, pero lo haremos más adelante.
- **Footer**: Usamos clases de Bootstrap como bg-dark, text-white, text-center, y py-3 para estilizar el pie de página. mt-auto asegura que el footer se mantenga al final de la página.

---

4. Implementación del Layout

Integrando el Layout Vamos a usar Header y Footer en nuestra aplicación, asegurándonos de que el contenido principal ocupe el espacio entre ellos.

Modifica src/App.jsx:

```
import Header from './components/Header';
import Footer from './components/Footer';

function App() {
  return (
    <div className="d-flex flex-column min-vh-100">
      <Header />
      <main className="flex-grow-1 container my-4">
        <h1>Mi Aplicación de Tareas con Bootstrap</h1>
        <p>Bienvenido a la aplicación de tareas.</p>
      </main>
      <Footer />
    </div>
```

```
  );
}

export default App;
```

Explicación:

- Usamos d-flex flex-column min-vh-100 para crear un layout flexible que ocupe toda la altura de la ventana (min-vh-100).
- Header y Footer se colocan en la parte superior e inferior, respectivamente.
- main con flex-grow-1 asegura que el contenido principal ocupe el espacio restante entre el header y el footer.
- container y my-4 (margen vertical) son clases de Bootstrap que centran el contenido y añaden espaciado.

Prueba:

- Inicia el servidor (npm run dev) y verifica que el layout se vea correctamente, con una barra de navegación en la parte superior, el contenido en el centro, y el footer en la parte inferior.

Sección 2: Creación de Layout con TailwindCSS

Vamos a crear un nuevo proyecto para usar TailwindCSS, ya que queremos mantener los dos enfoques separados.

5. Creación de Proyecto con Vite

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

```
npm create vite@latest task-app-tailwind -- --template react
cd task-app-tailwind
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:

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

```
  );
}

export default App;
```

Limpia src/index.css:

```
/* src/index.css */
body {
  margin: 0;
  font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, Oxygen,
Ubuntu, Cantarell, 'Open Sans', 'Helvetica Neue', sans-serif;
}
```

---

6. Instalación TailwindCSS y React Router DOM

Instalando TailwindCSS TailwindCSS es un framework CSS basado en utilidades, que te permite estilizar tu aplicación directamente en el HTML/JSX usando clases predefinidas.

Paso 1: Instalar TailwindCSS En el directorio task-app-tailwind, ejecuta:

```
npm install -D tailwindcss postcss autoprefixer
npx tailwindcss init -p
```

Esto instala TailwindCSS y sus dependencias, y genera los archivos de configuración tailwind.config.js y postcss.config.js.

Paso 2: Configurar tailwind.config.js Edita tailwind.config.js:

```
/** @type {import('tailwindcss').Config} */
export default {
  content: [
    "./index.html",
    "./src/**/*.{js,ts,jsx,tsx}",
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

Paso 3: Agregar las directivas de Tailwind a index.css Edita src/index.css:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Paso 4: Instalar React Router DOM Vamos a necesitar React Router para manejar la navegación, así que instalémoslo:

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

---

7. Configuración React Router DOM

Configurando el Enrutamiento Vamos a configurar React Router usando createBrowserRouter y RouterProvider, como hicimos en el tutorial anterior.

Paso 1: Crear páginas Crea una carpeta src/pages y los siguientes archivos:

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

```
function Home() {
  return (
    <div>
      <h1 className="text-3xl font-bold">Bienvenido a la Aplicación de Tareas</h1>
      <p>Esta es la página principal.</p>
    </div>
  );
}

export default Home;
```

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

```
function Tasks() {
  return (
    <div>
      <h1 className="text-3xl font-bold">Mis Tareas</h1>
      <p>Aquí puedes ver tus tareas.</p>
    </div>
  );
}

export default Tasks;
```

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

```
function AddTask() {
  return (
    <div>
      <h1 className="text-3xl font-bold">Agregar Nueva Tarea</h1>
      <p>Aquí puedes agregar una nueva tarea.</p>
    </div>
  );
}

export default AddTask;
```

Paso 2: Configurar las rutas Modifica src/main.jsx:

```
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 './index.css';

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

```
]);


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

Explicación:

- Usamos createBrowserRouter para definir las rutas.
- App será nuestro componente de layout (lo configuraremos en el siguiente paso).
- Definimos rutas para /, /tasks, y /add-task.

---

8. Configuración Base de TailwindCSS

Configuración Adicional TailwindCSS ya está configurado con los pasos anteriores, pero podemos personalizar el tema si lo deseamos.

Opción: Personalizar TailwindCSS (Opcional) Edita tailwind.config.js para agregar colores personalizados:

```
/** @type {import('tailwindcss').Config} */
export default {
  content: [
    "./index.html",
    "./src/**/*.{js,ts,jsx,tsx}",
  ],
  theme: {
    extend: {
      colors: {
        primary: '#ff6347',
        secondary: '#4682b4',
      },
    },
  },
  plugins: [],
}
```

Explicación:

- Agregamos colores personalizados (primary y secondary) que podemos usar como bg-primary, text-secondary, etc.

- Para este tutorial, usaremos principalmente las clases predeterminadas de Tailwind, pero ahora tienes la opción de usar colores personalizados.

---

9. Creación de Header y Footer Componentes

Creando el Header Crea src/components/Header.jsx:

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

function Header() {
  return (
    <nav className="bg-gray-800 text-white p-4">
      <div className="container mx-auto flex justify-between items-center">
        <NavLink to="/" className="text-2xl font-bold">
          Task App
        </NavLink>
        <ul className="flex space-x-6">
          <li>
            <NavLink
              to="/"
              className={({ isActive }) =>
                isActive ? 'text-red-400 font-bold' : 'hover:text-gray-300'
              }
            >
              Inicio
            </NavLink>
          </li>
          <li>
            <NavLink
              to="/tasks"
              className={({ isActive }) =>
                isActive ? 'text-red-400 font-bold' : 'hover:text-gray-300'
              }
            >
              Tareas
            </NavLink>
          </li>
          <li>
            <NavLink
              to="/add-task"
```

```
              className={({ isActive }) =>
                isActive ? 'text-red-400 font-bold' : 'hover:text-gray-300'
              }
            >
              Agregar Tarea
            </NavLink>
          </li>
        </ul>
      </div>
    </nav>
  );
}

export default Header;
```

Creando el Footer Crea src/components/Footer.jsx:

```
function Footer() {
  return (
    <footer className="bg-gray-800 text-white text-center py-4 mt-auto">
      <p>&copy; 2025 Task App. Todos los derechos reservados.</p>
    </footer>
  );
}

export default Footer;
```

Integrando el Layout Modifica src/App.jsx para usar Header, Footer, y Outlet:

```
import { Outlet } from 'react-router-dom';
import Header from './components/Header';
import Footer from './components/Footer';

function App() {
  return (
    <div className="flex flex-col min-h-screen">
      <Header />
      <main className="flex-grow container mx-auto my-6">
        <Outlet />
      </main>
      <Footer />
```

```
      </div>
    );
  }

  export default App;
```

Explicación:

- **Header**: Usamos clases de Tailwind como bg-gray-800, text-white, p-4, flex, y space-x-6 para estilizar la barra de navegación. NavLink resalta la ruta activa.
- **Footer**: Usamos bg-gray-800, text-white, text-center, py-4, y mt-auto para estilizar el pie de página.
- **App**: Usamos flex flex-col min-h-screen para crear un layout flexible, con flex-grow en main para que ocupe el espacio restante. container mx-auto my-6 centra el contenido y añade márgenes.

Prueba:

- Inicia el servidor (npm run dev) y verifica que el layout se vea correctamente, con una barra de navegación en la parte superior, el contenido de las páginas en el centro, y el footer en la parte inferior. Navega entre las rutas para asegurarte de que funcionan.

Sección 3: Hooks y Eventos

Vamos a seguir trabajando en el proyecto con TailwindCSS (task-app-tailwind) para implementar los hooks y eventos. Vamos a construir una página de tareas más interactiva, integrando los conceptos de hooks y eventos.

10. Qué es un Hook

¿Qué es un Hook? Un Hook es una función especial en React que te permite "enganchar" (hook) funcionalidad a componentes funcionales. Los hooks permiten manejar estado, efectos secundarios, y otras características de React sin necesidad de usar componentes de clase. Los hooks más comunes son useState, useEffect, useRef, entre otros.

Ejemplo Conceptual Los hooks se introdujeron en React 16.8 para resolver problemas como:

- Reutilizar lógica de estado entre componentes.
- Evitar la complejidad de los componentes de clase.
- Organizar mejor el código relacionado con el estado y los efectos secundarios.

Vamos a usar varios hooks en los siguientes pasos para ilustrar su uso.

---

11. Evento onClick

Evento onClick El evento onClick se usa para manejar clics en elementos como botones.

Ejemplo Práctico Vamos a agregar un botón en Tasks.jsx que permita alternar la visibilidad de las tareas.

Modifica src/pages/Tasks.jsx:

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

function Tasks() {
  const [showTasks, setShowTasks] = useState(true);

  const handleToggleTasks = () => {
    setShowTasks(!showTasks);
  };

  return (
    <div>
      <h1 className="text-3xl font-bold mb-4">Mis Tareas</h1>
      <button
        onClick={handleToggleTasks}
        className="bg-blue-500 text-white px-4 py-2 rounded hover:bg-blue-600 mb-4"
      >
        {showTasks ? 'Ocultar Tareas' : 'Mostrar Tareas'}
      </button>
      {showTasks && (
        <ul className="space-y-2">
          <li>Tarea 1</li>
          <li>Tarea 2</li>
          <li>Tarea 3</li>
        </ul>
      )}
    </div>
  );
}

export default Tasks;
```

Explicación:

- Usamos onClick para asociar el evento de clic al botón.
- handleToggleTasks alterna el estado showTasks entre true y false.
- Según el valor de showTasks, mostramos u ocultamos la lista de tareas.

---

12. useState

¿Qué es useState? useState es un Hook que te permite agregar estado a un componente funcional. Devuelve un array con dos elementos: el valor del estado y una función para actualizarlo.

Ejemplo Práctico Ya usamos useState en el ejemplo anterior para showTasks. Vamos a extender Tasks.jsx para manejar una lista de tareas dinámica.

Modifica src/pages/Tasks.jsx:

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

function Tasks() {
  const [showTasks, setShowTasks] = useState(true);
  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 handleToggleTasks = () => {
    setShowTasks(!showTasks);
  };

  const toggleTaskCompletion = (id) => {
    setTasks(
      tasks.map((task) =>
        task.id === id ? { ...task, completed: !task.completed } : task
      )
    );
  };

  return (
    <div>
      <h1 className="text-3xl font-bold mb-4">Mis Tareas</h1>
      <button
        onClick={handleToggleTasks}
        className="bg-blue-500 text-white px-4 py-2 rounded hover:bg-blue-600 mb-4"
      >
        {showTasks ? 'Ocultar Tareas' : 'Mostrar Tareas'}
      </button>
      {showTasks && (
```

```
        <ul className="space-y-2">
          {tasks.map((task) => (
            <li
              key={task.id}
              className={`flex items-center space-x-2 ${
                task.completed ? 'line-through text-gray-500' : ''
              }`}
            >
              <span>{task.text}</span>
              <button
                onClick={() => toggleTaskCompletion(task.id)}
                className={`px-2 py-1 rounded ${
                  task.completed
                    ? 'bg-red-500 text-white hover:bg-red-600'
                    : 'bg-green-500 text-white hover:bg-green-600'
                }`}
              >
                {task.completed ? 'Desmarcar' : 'Completar'}
              </button>
            </li>
          ))}
        </ul>
      )}
    </div>
  );
}

export default Tasks;
```

Explicación:

- Usamos useState para manejar la lista de tareas (tasks).
- toggleTaskCompletion actualiza el estado de una tarea específica, alternando su propiedad completed.
- Usamos onClick en los botones de cada tarea para marcarlas como completadas o no.

---

13. Evento onChange

Evento onChange El evento onChange se usa para manejar cambios en elementos de formulario como inputs.

Ejemplo Práctico Vamos a implementar un formulario en AddTask.jsx que permita agregar nuevas tareas.

Modifica src/pages/AddTask.jsx:

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

function AddTask() {
  const [taskText, setTaskText] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault();
    if (taskText.trim()) {
      console.log('Tarea agregada:', taskText);
      setTaskText('');
    }
  };

  return (
    <div>
      <h1 className="text-3xl font-bold mb-4">Agregar Nueva Tarea</h1>
      <form onSubmit={handleSubmit} className="space-y-4">
        <div>
          <label htmlFor="task" className="block text-lg font-medium">
            Nueva Tarea
          </label>
          <input
            type="text"
            id="task"
            value={taskText}
            onChange={(e) => setTaskText(e.target.value)}
            className="w-full p-2 border rounded focus:outline-none focus:ring-2
focus:ring-blue-500"
            placeholder="Escribe una tarea"
          />
        </div>
        <button
          type="submit"
          className="bg-blue-500 text-white px-4 py-2 rounded hover:bg-blue-600"
        >
          Agregar
        </button>
      </form>
```

```
    </div>
  );
}

export default AddTask;
```

Explicación:

- onMouseMove actualiza las coordenadas del mouse y establece isHovering en true.
- onMouseOut establece isHovering en false cuando el mouse sale del área.
- Mostramos las coordenadas solo cuando el mouse está sobre el área.

---

15. useEffect

¿Qué es useEffect? useEffect es un Hook que te permite manejar efectos secundarios en componentes funcionales, como suscripciones, temporizadores, o solicitudes a APIs.

Ejemplo Práctico Vamos a usar useEffect en Tasks.jsx para simular la carga de tareas desde una API.

Modifica src/pages/Tasks.jsx:

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

function Tasks() {
  const [showTasks, setShowTasks] = useState(true);
  const [tasks, setTasks] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    // Simulamos una llamada a una API
    setTimeout(() => {
      setTasks([
        { id: 1, text: 'Aprender React', completed: false },
        { id: 2, text: 'Hacer un proyecto', completed: true },
        { id: 3, text: 'Descansar', completed: false },
      ]);
      setLoading(false);
    }, 2000);
  }, []); // Array vacío significa que se ejecuta solo al montar

  const handleToggleTasks = () => {
```

```
  setShowTasks(!showTasks);
};

const toggleTaskCompletion = (id) => {
  setTasks(
    tasks.map((task) =>
      task.id === id ? { ...task, completed: !task.completed } : task
    )
  );
};

if (loading) {
  return <p className="text-center text-lg">Cargando tareas...</p>;
}

return (
  <div>
    <h1 className="text-3xl font-bold mb-4">Mis Tareas</h1>
    <button
      onClick={handleToggleTasks}
      className="bg-blue-500 text-white px-4 py-2 rounded hover:bg-blue-600 mb-4"
    >
      {showTasks ? 'Ocultar Tareas' : 'Mostrar Tareas'}
    </button>
    {showTasks && (
      <ul className="space-y-2">
        {tasks.map((task) => (
          <li
            key={task.id}
            className={`flex items-center space-x-2 ${
              task.completed ? 'line-through text-gray-500' : ''
            }`}
          >
            <span>{task.text}</span>
            <button
              onClick={() => toggleTaskCompletion(task.id)}
              className={`px-2 py-1 rounded ${
                task.completed
                  ? 'bg-red-500 text-white hover:bg-red-600'
                  : 'bg-green-500 text-white hover:bg-green-600'
```

```
                }`}
              >
                {task.completed ? 'Desmarcar' : 'Completar'}
              </button>
            </li>
          ))}
        </ul>
      )}
    </div>
  );
}

export default Tasks;
```

Explicación:

- useEffect simula una llamada a una API que carga las tareas después de 2 segundos.
- El array vacío [] asegura que el efecto se ejecute solo al montar el componente.
- Mostramos un mensaje de "Cargando..." mientras loading es true.

---

16. Custom Hooks (Select Dependientes)

¿Qué es un Custom Hook? Un custom hook es una función que encapsula lógica reutilizable usando otros hooks. Por convención, los custom hooks comienzan con use.

Ejemplo Práctico Vamos a crear un custom hook para manejar selects dependientes (por ejemplo, seleccionar una categoría y luego una subcategoría).

Crea src/hooks/useDependentSelects.js:

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

function useDependentSelects() {
  const [category, setCategory] = useState('');
  const [subcategory, setSubcategory] = useState('');

  const categories = {
    work: ['Development', 'Design', 'Testing'],
    personal: ['Exercise', 'Reading', 'Hobbies'],
  };

  const subcategories = category ? categories[category] : [];
```

```
  return {
    category,
    setCategory,
    subcategories,
    subcategory,
    setSubcategory,
  };
}

export default useDependentSelects;
```

Usar el Custom Hook Modifica src/pages/AddTask.jsx para usar el custom hook:

```
import { useState } from 'react';
import useDependentSelects from '../hooks/useDependentSelects';

function AddTask() {
  const [taskText, setTaskText] = useState('');
  const { category, setCategory, subcategories, subcategory, setSubcategory } =
    useDependentSelects();

  const handleSubmit = (e) => {
    e.preventDefault();
    if (taskText.trim()) {
      console.log('Tarea agregada:', { taskText, category, subcategory });
      setTaskText('');
      setCategory('');
      setSubcategory('');
    }
  };

  return (
    <div>
      <h1 className="text-3xl font-bold mb-4">Agregar Nueva Tarea</h1>
      <form onSubmit={handleSubmit} className="space-y-4">
        <div>
          <label htmlFor="task" className="block text-lg font-medium">
            Nueva Tarea
          </label>
```

```
          <input
            type="text"
            id="task"
            value={taskText}
            onChange={(e) => setTaskText(e.target.value)}
            className="w-full p-2 border rounded focus:outline-none focus:ring-2
focus:ring-blue-500"
            placeholder="Escribe una tarea"
          />
        </div>
        <div>
          <label htmlFor="category" className="block text-lg font-medium">
            Categoría
          </label>
          <select
            id="category"
            value={category}
            onChange={(e) => setCategory(e.target.value)}
            className="w-full p-2 border rounded focus:outline-none focus:ring-2
focus:ring-blue-500"
          >
            <option value="">Selecciona una categoría</option>
            <option value="work">Trabajo</option>
            <option value="personal">Personal</option>
          </select>
        </div>
        {category && (
          <div>
            <label htmlFor="subcategory" className="block text-lg font-medium">
              Subcategoría
            </label>
            <select
              id="subcategory"
              value={subcategory}
              onChange={(e) => setSubcategory(e.target.value)}
              className="w-full p-2 border rounded focus:outline-none focus:ring-2
focus:ring-blue-500"
            >
              <option value="">Selecciona una subcategoría</option>
              {subcategories.map((sub) => (
```

```
                <option key={sub} value={sub}>
                  {sub}
                </option>
              ))}
            </select>
          </div>
        )}
        <button
          type="submit"
          className="bg-blue-500 text-white px-4 py-2 rounded hover:bg-blue-600"
        >
          Agregar
        </button>
      </form>
    </div>
  );
}

export default AddTask;
```

Explicación:

- useDependentSelects encapsula la lógica para manejar dos selects dependientes.
- Cuando el usuario selecciona una categoría, el segundo select se actualiza con las subcategorías correspondientes.
- Al enviar el formulario, mostramos la tarea, categoría, y subcategoría en la consola.

---

17. useLoaderData

¿Qué es useLoaderData? useLoaderData es un Hook de React Router que te permite acceder a los datos cargados por un loader definido en una ruta. Es útil para cargar datos antes de renderizar un componente.

Ejemplo Práctico Vamos a usar useLoaderData para cargar las tareas en Tasks.jsx.

Modifica src/main.jsx para agregar un loader:

```
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';
```

```jsx
import AddTask from './pages/AddTask';
import './index.css';

// Simulamos una función que carga datos
const tasksLoader = async () => {
  await new Promise((resolve) => setTimeout(resolve, 2000)); // Simulamos una demora
  return [
    { id: 1, text: 'Aprender React', completed: false },
    { id: 2, text: 'Hacer un proyecto', completed: true },
    { id: 3, text: 'Descansar', completed: false },
  ];
};

const router = createBrowserRouter([
  {
    element: <App />,
    children: [
      {
        path: '/',
        element: <Home />,
      },
      {
        path: '/tasks',
        element: <Tasks />,
        loader: tasksLoader,
      },
      {
        path: '/add-task',
        element: <AddTask />,
      },
    ],
  },
]);

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

Modifica src/pages/Tasks.jsx para usar useLoaderData:

```jsx
import { useState } from 'react';
import { useLoaderData } from 'react-router-dom';

function Tasks() {
  const [showTasks, setShowTasks] = useState(true);
  const tasks = useLoaderData();

  const handleToggleTasks = () => {
    setShowTasks(!showTasks);
  };

  const toggleTaskCompletion = (id) => {
    // Nota: useLoaderData devuelve datos inmutables, así que esto no funcionará directamente.
    // En un caso real, necesitarías manejar el estado en un contexto o API.
    console.log(`Tarea ${id} marcada como completada`);
  };

  return (
    <div>
      <h1 className="text-3xl font-bold mb-4">Mis Tareas</h1>
      <button
        onClick={handleToggleTasks}
        className="bg-blue-500 text-white px-4 py-2 rounded hover:bg-blue-600 mb-4"
      >
        {showTasks ? 'Ocultar Tareas' : 'Mostrar Tareas'}
      </button>
      {showTasks && (
        <ul className="space-y-2">
          {tasks.map((task) => (
            <li
              key={task.id}
              className={`flex items-center space-x-2 ${
                task.completed ? 'line-through text-gray-500' : ''
              }`}
            >
              <span>{task.text}</span>
              <button
```

```
                onClick={() => toggleTaskCompletion(task.id)}
                className={`px-2 py-1 rounded ${
                  task.completed
                    ? 'bg-red-500 text-white hover:bg-red-600'
                    : 'bg-green-500 text-white hover:bg-green-600'
                }`}
              >
                {task.completed ? 'Desmarcar' : 'Completar'}
              </button>
            </li>
          ))}
        </ul>
      )}
    </div>
  );
}

export default Tasks;
```

Explicación:

- Definimos un loader (tasksLoader) que simula la carga de datos.
- useLoaderData accede a los datos cargados por el loader.
- Nota: Los datos de useLoaderData son inmutables. Para manejar cambios (como marcar una tarea como completada), necesitarías un estado global o una API real.

---

18. useNavigate

¿Qué es useNavigate? useNavigate es un Hook de React Router que te permite navegar programáticamente entre rutas.

Ejemplo Práctico Vamos a usar useNavigate en AddTask.jsx para redirigir al usuario a la página de tareas después de agregar una tarea.

Modifica src/pages/AddTask.jsx:

```
import { useState } from 'react';
import { useNavigate } from 'react-router-dom';
import useDependentSelects from '../hooks/useDependentSelects';

function AddTask() {
  const [taskText, setTaskText] = useState('');
```

```
  const { category, setCategory, subcategories, subcategory, setSubcategory } =
    useDependentSelects();
  const navigate = useNavigate();

  const handleSubmit = (e) => {
    e.preventDefault();
    if (taskText.trim()) {
      console.log('Tarea agregada:', { taskText, category, subcategory });
      setTaskText('');
      setCategory('');
      setSubcategory('');
      navigate('/tasks'); // Redirigir a la página de tareas
    }
  };

  return (
    <div>
      <h1 className="text-3xl font-bold mb-4">Agregar Nueva Tarea</h1>
      <form onSubmit={handleSubmit} className="space-y-4">
        <div>
          <label htmlFor="task" className="block text-lg font-medium">
            Nueva Tarea
          </label>
          <input
            type="text"
            id="task"
            value={taskText}
            onChange={(e) => setTaskText(e.target.value)}
            className="w-full p-2 border rounded focus:outline-none focus:ring-2
focus:ring-blue-500"
            placeholder="Escribe una tarea"
          />
        </div>
        <div>
          <label htmlFor="category" className="block text-lg font-medium">
            Categoría
          </label>
          <select
            id="category"
            value={category}
```

```
            onChange={(e) => setCategory(e.target.value)}
            className="w-full p-2 border rounded focus:outline-none focus:ring-2
focus:ring-blue-500"
          >
            <option value="">Selecciona una categoría</option>
            <option value="work">Trabajo</option>
            <option value="personal">Personal</option>
          </select>
        </div>
        {category && (
          <div>
            <label htmlFor="subcategory" className="block text-lg font-medium">
              Subcategoría
            </label>
            <select
              id="subcategory"
              value={subcategory}
              onChange={(e) => setSubcategory(e.target.value)}
              className="w-full p-2 border rounded focus:outline-none focus:ring-2
focus:ring-blue-500"
            >
              <option value="">Selecciona una subcategoría</option>
              {subcategories.map((sub) => (
                <option key={sub} value={sub}>
                  {sub}
                </option>
              ))}
            </select>
          </div>
        )}
        <button
          type="submit"
          className="bg-blue-500 text-white px-4 py-2 rounded hover:bg-blue-600"
        >
          Agregar
        </button>
      </form>
    </div>
  );
}
```

```
export default AddTask;
```

Explicación:

- useNavigate devuelve una función navigate que usamos para redirigir al usuario a /tasks después de agregar una tarea.
- Esto mejora la experiencia del usuario al llevarlo automáticamente a la lista de tareas.

---

19. useLocation

¿Qué es useLocation? useLocation es un Hook de React Router que te da acceso al objeto de ubicación actual, incluyendo la ruta, querystring, y más.

Ejemplo Práctico Vamos a usar useLocation en Tasks.jsx para mostrar la URL actual.

Modifica src/pages/Tasks.jsx:

```
import { useState } from 'react';
import { useLoaderData, useLocation } from 'react-router-dom';

function Tasks() {
  const [showTasks, setShowTasks] = useState(true);
  const tasks = useLoaderData();
  const location = useLocation();

  const handleToggleTasks = () => {
    setShowTasks(!showTasks);
  };

  const toggleTaskCompletion = (id) => {
    console.log(`Tarea ${id} marcada como completada`);
  };

  return (
    <div>
      <h1 className="text-3xl font-bold mb-4">Mis Tareas</h1>
      <p className="mb-4">URL actual: {location.pathname}</p>
      <button
        onClick={handleToggleTasks}
        className="bg-blue-500 text-white px-4 py-2 rounded hover:bg-blue-600 mb-4"
      >
```

```
        {showTasks ? 'Ocultar Tareas' : 'Mostrar Tareas'}
      </button>
      {showTasks && (
        <ul className="space-y-2">
          {tasks.map((task) => (
            <li
              key={task.id}
              className={`flex items-center space-x-2 ${
                task.completed ? 'line-through text-gray-500' : ''
              }`}
            >
              <span>{task.text}</span>
              <button
                onClick={() => toggleTaskCompletion(task.id)}
                className={`px-2 py-1 rounded ${
                  task.completed
                    ? 'bg-red-500 text-white hover:bg-red-600'
                    : 'bg-green-500 text-white hover:bg-green-600'
                }`}
              >
                {task.completed ? 'Desmarcar' : 'Completar'}
              </button>
            </li>
          ))}
        </ul>
      )}
    </div>
  );
}

export default Tasks;
```

Explicación:

- useLocation nos da acceso a location.pathname, que muestra la ruta actual (por ejemplo, /tasks).
- Esto puede ser útil para debugging o para mostrar información contextual.

---

20. useRef (Continuación)

Completando el Código de AddTask.jsx

Vamos a asegurarnos de que el código de AddTask.jsx esté completo y funcional, incluyendo el uso de useRef para enfocar el input y un ejemplo adicional para demostrar cómo useRef puede usarse para almacenar un valor mutable sin causar re-renderizados.

Modifica src/pages/AddTask.jsx para que quede como sigue:

```
import { useState, useEffect, useRef } from 'react';
import { useNavigate } from 'react-router-dom';
import useDependentSelects from '../hooks/useDependentSelects';

function AddTask() {
  const [taskText, setTaskText] = useState('');
  const { category, setCategory, subcategories, subcategory, setSubcategory } =
    useDependentSelects();
  const navigate = useNavigate();
  const taskInputRef = useRef(null);
  const submitCountRef = useRef(0); // Para contar cuántas veces se ha enviado el
formulario

  // Enfocar el input al montar el componente
  useEffect(() => {
    taskInputRef.current.focus();
  }, []);

  // Incrementar el contador de envíos cada vez que se envía el formulario
  const handleSubmit = (e) => {
    e.preventDefault();
    if (taskText.trim()) {
      submitCountRef.current += 1; // Incrementar el contador sin causar re-render
      console.log('Tarea agregada:', { taskText, category, subcategory });
      console.log('Número de envíos:', submitCountRef.current);
      setTaskText('');
      setCategory('');
      setSubcategory('');
      navigate('/tasks');
    }
  };

  return (
    <div>
      <h1 className="text-3xl font-bold mb-4">Agregar Nueva Tarea</h1>
```

```
      <form onSubmit={handleSubmit} className="space-y-4">
        <div>
          <label htmlFor="task" className="block text-lg font-medium">
            Nueva Tarea
          </label>
          <input
            type="text"
            id="task"
            ref={taskInputRef} // Asignar la referencia al input
            value={taskText}
            onChange={(e) => setTaskText(e.target.value)}
            className="w-full p-2 border rounded focus:outline-none focus:ring-2
focus:ring-blue-500"
            placeholder="Escribe una tarea"
          />
        </div>
        <div>
          <label htmlFor="category" className="block text-lg font-medium">
            Categoría
          </label>
          <select
            id="category"
            value={category}
            onChange={(e) => setCategory(e.target.value)}
            className="w-full p-2 border rounded focus:outline-none focus:ring-2
focus:ring-blue-500"
          >
            <option value="">Selecciona una categoría</option>
            <option value="work">Trabajo</option>
            <option value="personal">Personal</option>
          </select>
        </div>
        {category && (
          <div>
            <label htmlFor="subcategory" className="block text-lg font-medium">
              Subcategoría
            </label>
            <select
              id="subcategory"
              value={subcategory}
```

```
              onChange={(e) => setSubcategory(e.target.value)}
              className="w-full p-2 border rounded focus:outline-none focus:ring-2
focus:ring-blue-500"
            >
              <option value="">Selecciona una subcategoría</option>
              {subcategories.map((sub) => (
                <option key={sub} value={sub}>
                  {sub}
                </option>
              ))}
            </select>
          </div>
        )}
        <button
          type="submit"
          className="bg-blue-500 text-white px-4 py-2 rounded hover:bg-blue-600"
        >
          Agregar
        </button>
      </form>
      <p className="mt-4">Número de tareas enviadas: {submitCountRef.current}</p>
    </div>
  );
}

export default AddTask;
```

Explicación Detallada:

1. **Uso de useRef para Enfocar el Input:**
   - Creamos una referencia taskInputRef usando useRef(null).
   - La asignamos al input con la prop ref={taskInputRef}.
   - Usamos useEffect para enfocar el input cuando el componente se monta (taskInputRef.current.focus()).
   - Esto mejora la experiencia del usuario, ya que el input está listo para escribir inmediatamente al cargar la página.
2. **Uso de useRef para Almacenar un Valor Mutable:**
   - Creamos otra referencia submitCountRef para contar cuántas veces se ha enviado el formulario.
   - A diferencia de useState, actualizar submitCountRef.current no causa un re-render del componente, lo que lo hace ideal para valores que no necesitan actualizar la UI inmediatamente.
   - En handleSubmit, incrementamos submitCountRef.current cada vez que se envía el formulario y mostramos el valor en la consola.
   - También mostramos el valor en la UI (Número de tareas enviadas: {submitCountRef.current}). Nota que, aunque el valor se actualiza en

submitCountRef.current, la UI no se re-renderiza automáticamente para reflejar este cambio porque useRef no dispara re-renders. Para que el cambio sea visible en la UI, necesitaríamos usar useState o forzar un re-render (como al navegar con navigate).

3. **Por Qué Usar useRef Aquí:**
    - **Acceso al DOM:** useRef es perfecto para interactuar directamente con elementos del DOM, como enfocar un input, hacer scroll a un elemento, o medir dimensiones.
    - **Valores Persistentes:** useRef es útil para almacenar valores que persisten entre renders sin causar re-renders, como contadores, referencias a intervalos/temporizadores, o valores temporales.

Prueba:

- Ve a /add-task y verifica que el input se enfoque automáticamente al cargar la página.
- Envía el formulario varias veces y observa la consola para ver cómo se incrementa submitCountRef.current. Nota que el valor en la UI no se actualiza hasta que el componente se re-renderiza (por ejemplo, al navegar y volver).

Ejemplo Adicional: Usar useRef para un Temporizador

Para ilustrar otro uso de useRef, vamos a agregar un temporizador en Home.jsx que cuente cuántos segundos el usuario ha estado en la página, usando useRef para almacenar el ID del intervalo.

Modifica src/pages/Home.jsx:

```
import { useState, useEffect, useRef } from 'react';

function Home() {
  const [coords, setCoords] = useState({ x: 0, y: 0 });
  const [isHovering, setIsHovering] = useState(false);
  const [seconds, setSeconds] = useState(0);
  const intervalRef = useRef(null);

  useEffect(() => {
    // Iniciar el temporizador
    intervalRef.current = setInterval(() => {
      setSeconds((prev) => prev + 1);
    }, 1000);

    // Limpiar el intervalo al desmontar el componente
    return () => {
      clearInterval(intervalRef.current);
    };
  }, []);
```

```
  const handleMouseMove = (e) => {
    setCoords({ x: e.clientX, y: e.clientY });
    setIsHovering(true);
  };

  const handleMouseOut = () => {
    setIsHovering(false);
  };

  return (
    <div>
      <h1 className="text-3xl font-bold mb-4">Bienvenido a la Aplicación de Tareas</h1>
      <p>Has estado en esta página por {seconds} segundos.</p>
      <div
        onMouseMove={handleMouseMove}
        onMouseOut={handleMouseOut}
        className="w-64 h-64 bg-gray-200 rounded-lg flex items-center justify-center mt-
4"
      >
        {isHovering ? (
          <p>
            Coordenadas: X: {coords.x}, Y: {coords.y}
          </p>
        ) : (
          <p>Pasa el mouse aquí</p>
        )}
      </div>
    </div>
  );
}

export default Home;
```

Explicación del Ejemplo Adicional:

1. **Almacenar el ID del Intervalo:**
   - Usamos intervalRef para almacenar el ID del intervalo creado por setInterval.
   - Esto nos permite limpiar el intervalo cuando el componente se desmonta, evitando fugas de memoria.
2. **Actualizar el Estado con useState:**

- Aunque intervalRef almacena el ID del intervalo, usamos useState (seconds) para actualizar la UI cada segundo.
- setInterval incrementa seconds cada 1000ms (1 segundo).

3. **Limpieza del Efecto:**
   - En el callback de limpieza de useEffect (return () => {...}), usamos clearInterval(intervalRef.current) para detener el intervalo cuando el componente se desmonta.

Prueba:

- Ve a la página principal (/) y observa cómo el contador de segundos aumenta cada segundo.
- Navega a otra página (por ejemplo, /tasks) y regresa a la página principal. Verifica que el contador se reinicia, lo que indica que el intervalo se limpió correctamente al desmontar el componente.
