Vamos a hacer dos cosas en este tutorial:

1. **Refactorizar el ThemeSwitcher:** Lo cambiaremos de un botón de texto a un botón de icono profesional usando **SVG**, y en el proceso, explicaremos por qué es una mejor práctica que las fuentes de iconos.
2. **Mejorar el TodoList:** Introduciremos un concepto clave: **Composición de Componentes**. Dividiremos TodoList en TodoList (el contenedor) y TodoItem (el hijo), lo que nos permitirá añadir botones de "completar" y "eliminar" para cada tarea individual.

Tutorial: SVGs, Composición de Componentes y Flujo de Datos

En este tutorial, daremos un gran salto en "pensar en React", aprendiendo a dividir componentes complejos en piezas más pequeñas y cómo hacer que el padre y el hijo se comuniquen.

Parte 1: Iconos SVG para el ThemeSwitcher

Primero, actualicemos ese botón de cambio de tema.

¿Qué es un SVG? ¿Icono vs. SVG? ¿Rendimiento?

- **Icono (Fuente de Iconos):** Piensa en "Font Awesome". Es literalmente una fuente, como "Arial", pero en lugar de letras, tiene símbolos.
    - *Desventaja:* El navegador debe descargar un archivo de fuente completo (.woff, .ttf), lo cual es una petición HTTP extra que bloquea el renderizado. A veces son difíciles de alinear y estilizar.
- **SVG (Scalable Vector Graphics):** Es un formato de imagen basado en XML (texto). Describe formas y líneas en lugar de píxeles.
    - *Ventaja:* ¡Puedes pegar el código SVG **directamente en tu JSX**! Se convierte en parte de tu bundle de JavaScript, lo que significa **cero peticiones HTTP extra**.
    - *Rendimiento:* Gana el SVG. Al estar "en línea" (inline), se carga instantáneamente con tu componente.
    - *Calidad:* Es "escalable", por lo que se ve perfectamente nítido en cualquier tamaño, a diferencia de una imagen .png.

Paso 1.1: Crear los Componentes de Iconos

Crearemos dos componentes simples que *solo* retornan un SVG.

1. Crea una nueva carpeta src/components/Icons.
2. Dentro, crea IconMoon.js:

JavaScript

```
import React from 'react';

const IconMoon = ({ size = 24 }) => (
  <svg
    xmlns="http://www.w3.org/2000/svg"
    width={size}
    height={size}
    viewBox="0 0 24 24"
    fill="none"
    stroke="currentColor"
    strokeWidth="2"
    strokeLinecap="round"
    strokeLinejoin="round"
  >
    <path d="M21 12.79A9 9 0 1 1 11.21 3 7 7 0 0 0 21 12.79z"></path>
  </svg>
);

export default IconMoon;
```

3. Ahora crea IconSun.js en la misma carpeta:

```
JavaScript
import React from 'react';

const IconSun = ({ size = 24 }) => (
  <svg
    xmlns="http://www.w3.org/2000/svg"
    width={size}
    height={size}
    viewBox="0 0 24 24"
```

```
    fill="none"
    stroke="currentColor"
    strokeWidth="2"
    strokeLinecap="round"
    strokeLinejoin="round"
  >
    <circle cx="12" cy="12" r="5"></circle>
    <line x1="12" y1="1" x2="12" y2="3"></line>
    <line x1="12" y1="21" x2="12" y2="23"></line>
    <line x1="4.22" y1="4.22" x2="5.64" y2="5.64"></line>
    <line x1="18.36" y1="18.36" x2="19.78" y2="19.78"></line>
    <line x1="1" y1="12" x2="3" y2="12"></line>
    <line x1="21" y1="12" x2="23" y2="12"></line>
    <line x1="4.22" y1="19.78" x2="5.64" y2="18.36"></line>
    <line x1="18.36" y1="5.64" x2="19.78" y2="4.22"></line>
  </svg>
);

export default IconSun;
```

(Estos SVGs son del popular set de iconos "Feather Icons")

Paso 1.2: Actualizar el ThemeSwitcher

Ahora, reemplacemos el texto del botón con nuestros nuevos iconos.

1. Abre src/components/ThemeSwitcher/ThemeSwitcher.js.
2. Impórtalos y úsalos:

JavaScript

```
import React, { useContext } from 'react';
import ThemeContext from '../../context/ThemeContext';
import IconMoon from '../Icons/IconMoon'; // <-- Importar
import IconSun from '../Icons/IconSun';   // <-- Importar
```

```
import './ThemeSwitcher.css'; // Crearemos este archivo

const ThemeSwitcher = () => {
  const { theme, toggleTheme } = useContext(ThemeContext);

  return (
    <button onClick={toggleTheme} className="theme-switcher-btn">
      {theme === 'light' ? <IconMoon /> : <IconSun />}
    </button>
  );
};

export default ThemeSwitcher;
```

3. Crea src/components/ThemeSwitcher/ThemeSwitcher.css para darle estilo:

CSS

```
.theme-switcher-btn {
  background-color: transparent;
  border: 1px solid transparent; /* Ocultamos borde */
  color: inherit; /* Hereda el color del texto (blanco o negro) */
  cursor: pointer;
  padding: 5px;
  border-radius: 50%; /* Lo hace circular */
  display: inline-flex;
  align-items: center;
  justify-content: center;

  /* Posicionarlo en la esquina */
  position: absolute;
  top: 20px;
```

```
  right: 20px;
}

.theme-switcher-btn:hover {
  background-color: rgba(128, 128, 128, 0.2);
}
```

4. Finalmente, quita el <ThemeSwitcher /> de App.js y colócalo *dentro* de src/components/Header/Header.js. Tiene más sentido que el interruptor viva en el encabezado.
    - En Header.js:

```
JavaScript
import React from 'react';
import './Header.css';
import ThemeSwitcher from '../ThemeSwitcher/ThemeSwitcher'; // Importar

const Header = () => {
  return (
    <header className="app-header">
      <h1>Mi Aplicación Modular</h1>
      <ThemeSwitcher /> {/* <-- Añadir aquí */}
    </header>
  );
};
// ...
```

    - En App.js (quítalo de aquí).

¡Felicidades! Ahora tienes un interruptor de tema profesional que usa SVGs en línea.

Parte 2: Mejorando el TodoList (Composición)

Ahora, la parte más importante. Nuestro TodoList.js actual hace demasiado: gestiona la lista, gestiona el input *y* renderiza cada ítem. Vamos a dividirlo.

El Concepto: Composición y "Levantar el Estado" (Lifting State Up)

1. **TodoList.js (Padre):** Su trabajo será tener el **estado** (la lista de tareas) y las **funciones** para modificar ese estado (handleDelete, handleToggle).
2. **TodoItem.js (Hijo):** Su trabajo será solo **mostrar una tarea**. No tendrá estado propio. Recibirá los datos (task) y las funciones (onDelete, onToggle) como **props**. Cuando el usuario haga clic en "borrar", llamará a la función onDelete que le pasó su padre.

Paso 2.1: Crear el Icono de Basura

Igual que antes, creemos un icono para borrar.

1. Crea src/components/Icons/IconTrash.js:

JavaScript

```
import React from 'react';


const IconTrash = ({ size = 18 }) => (
 <svg
  xmlns="http://www.w3.org/2000/svg"
  width={size}
  height={size}
  viewBox="0 0 24 24"
  fill="none"
  stroke="currentColor"
  strokeWidth="2"
  strokeLinecap="round"
  strokeLinejoin="round"
 >
  <polyline points="3 6 5 6 21 6"></polyline>
  <path d="M19 6v14a2 2 0 0 1-2 2H7a2 2 0 0 1-2-2V6m3 0V4a2 2 0 0 1 2-2h4a2 2 0 0 1 2 2v2"></path>
  <line x1="10" y1="11" x2="10" y2="17"></line>
  <line x1="14" y1="11" x2="14" y2="17"></line>
 </svg>
```

);

export default IconTrash;

Paso 2.2: Crear el Componente TodoItem

Este es nuestro nuevo componente hijo.

1. Crea una nueva carpeta src/components/TodoItem.
2. Dentro, crea TodoItem.js y TodoItem.css.
3. Añade el código a TodoItem.js:

JavaScript

```
import React from 'react';
import './TodoItem.css';
import IconTrash from '../Icons/IconTrash';

const TodoItem = ({ task, onToggleComplete, onDeleteTask }) => {
  return (
    <li className={`todo-item ${task.isComplete ? 'completed' : ''}`}>
      <div className="task-content">
        <input
          type="checkbox"
          checked={task.isComplete}
          onChange={() => onToggleComplete(task.id)}
        />
        <span className="task-text">{task.text}</span>
      </div>
      <button
        className="delete-btn"
        onClick={() => onDeleteTask(task.id)}
      >
        <IconTrash />
```

```
      </button>
    </li>
  );
};

export default TodoItem;
```

Desglose:

- Este componente es "tonto". No sabe *cómo* borrar o completar.
- Recibe task (los datos a mostrar) y dos funciones, onToggleComplete y onDeleteTask, como props.
- Cuando haces clic en el checkbox, llama a onToggleComplete(task.id), pasando el ID de la tarea que debe ser marcada.
- Cuando haces clic en el botón de basura, llama a onDeleteTask(task.id).

4. Añade los estilos a TodoItem.css:

CSS

```
.todo-item {
  display: flex;
  justify-content: space-between;
  align-items: center;
  padding: 10px;
  background-color: #fff;
  border-bottom: 1px solid #eee;
}

/* Estilo cuando el tema es oscuro */
.App.dark .todo-item {
  background-color: #3a3f4b;
  border-color: #555;
}
```

```
.task-content {
  display: flex;
  align-items: center;
  gap: 10px;
}

.task-text {
  font-size: 1.1em;
}

.todo-item.completed .task-text {
  text-decoration: line-through;
  color: #999;
}

.delete-btn {
  background: none;
  border: none;
  color: #ff4d4d;
  cursor: pointer;
  display: flex;
  align-items: center;
  padding: 5px;
  border-radius: 4px;
}
.delete-btn:hover {
  background-color: rgba(255, 77, 77, 0.1);
}
```

Paso 2.3: Refactorizar TodoList.js (El Padre)

Ahora, actualicemos el componente padre para que *use* TodoItem.

1. Abre src/components/TodoList/TodoList.js.
2. Importa TodoItem y modifica el componente:

JavaScript

```
import React, { useState } from 'react';
import './TodoList.css';
import TodoItem from '../TodoItem/TodoItem'; // <-- Importar el hijo

const TodoList = () => {
  // Estado 'tasks' ahora necesita saber si está completo
  const [tasks, setTasks] = useState([
    { id: 1, text: 'Aprender React', isComplete: true },
    { id: 2, text: 'Construir una App', isComplete: false },
    { id: 3, text: 'Modularizar componentes', isComplete: false }
  ]);

  const [inputValue, setInputValue] = useState('');

  const handleAddTask = (e) => {
    e.preventDefault();
    if (inputValue.trim() === '') return;

    const newTask = {
      id: Date.now(),
      text: inputValue,
      isComplete: false // Nueva propiedad
    };

    // Usamos '.concat' o '...' para inmutabilidad
```

```
  setTasks(tasks.concat(newTask));
  setInputValue('');
};

// --- NUEVAS FUNCIONES ---

// Función para marcar/desmarcar una tarea
const handleToggleComplete = (idToToggle) => {
  setTasks(
    tasks.map(task =>
      task.id === idToToggle
        ? { ...task, isComplete: !task.isComplete } // Crea un nuevo objeto
        : task // Devuelve el objeto original
    )
  );
};

// Función para eliminar una tarea
const handleDeleteTask = (idToDelete) => {
  setTasks(
    tasks.filter(task => task.id !== idToDelete)
  );
};

// --- RENDER ACTUALIZADO ---

return (
  <div className="todo-list-container">
    <h2>Mi Lista de Tareas</h2>
```

```
      <form onSubmit={handleAddTask} className="add-task-form">
        <input
          type="text"
          value={inputValue}
          onChange={(e) => setInputValue(e.target.value)}
          placeholder="Añade una nueva tarea..."
        />
        <button type="submit">Añadir</button>
      </form>

      <ul>
        {/* Aquí está la magia:
          Mapeamos las tareas y por cada una, renderizamos un <TodoItem />
          pasándole los datos y las FUNCIONES como props.
        */}
        {tasks.map(task => (
          <TodoItem
            key={task.id}
            task={task}
            onToggleComplete={handleToggleComplete}
            onDeleteTask={handleDeleteTask}
          />
        ))}
      </ul>
    </div>
  );
};
```

export default TodoList;

Desglose de las nuevas funciones:

- **handleToggleComplete**: Este es un patrón fundamental. Usamos .map() para crear un **nuevo arreglo**. Si el id de la tarea coincide, creamos un **nuevo objeto de tarea** ({ ...task }) pero invertimos el valor de isComplete. Si no coincide, solo devolvemos la tarea tal como estaba.
- **handleDeleteTask**: Este es más simple. Usamos .filter() para crear un **nuevo arreglo** que contiene todas las tareas *excepto* la que tiene el idToDelete.
- **Actualización del Render**: El <ul> ya no tiene s. Ahora tiene un .map() que genera nuestros componentes <TodoItem>, a los que "pasamos" las funciones que acabamos de crear.

Resultado Final

¡Ejecuta npm start! Verás tu aplicación con:

1. Un elegante icono de sol/luna en la esquina para cambiar el tema.
2. Tu lista de tareas, donde cada ítem tiene un checkbox para tacharlo y un icono de basura para eliminarlo.

Resumen de Conceptos Clave

- **SVGs en línea:** La forma moderna y de alto rendimiento de usar iconos en React.
- **Composición de Componentes:** El pilar de React. Dividir la UI en componentes pequeños y reutilizables (un TodoList que contiene muchos TodoItem).
- **Flujo de Datos Unidireccional:** El estado vive en el padre (TodoList).
- **Pasar Funciones como Props:** La forma de que un hijo (TodoItem) le "diga" al padre (TodoList) que algo debe cambiar (ej: "¡Bórrame!").
- **Actualizaciones Inmutables:** Usar .map() y .filter() para crear nuevos arreglos de estado en lugar de modificar los existentes, lo cual es esencial para que React detecte los cambios.