Tutorial: Creando un Proyecto en React 2025 - Paso a Paso

1. Configuración del Proyecto en React

Primero, vamos a crear un proyecto en React utilizando Vite, que es una herramienta moderna y rápida para inicializar proyectos en 2025. Asegúrate de tener Node.js instalado (versión 18 o superior recomendada).

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
```

Esto creará un proyecto React con Vite. La estructura inicial será algo como:

```
my-react-app/
  ├── node_modules/
  ├── public/
  ├── src/
  │   ├── App.jsx
  │   ├── main.jsx
  │   └── index.css
  ├── package.json
  └── vite.config.js
```

Paso 2: Iniciar el servidor de desarrollo 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:

```jsx
function App() {
  return (
    <div>
      <h1>Mi Aplicación React 2025</h1>
    </div>
  );
}

export default App;
```

También limpia src/index.css si deseas empezar desde cero.

Desarrollo de los Temas

1. Conociendo JSX

¿Qué es JSX? JSX es una extensión de sintaxis para JavaScript que permite escribir código similar a HTML dentro de JavaScript. Es una de las características principales de React, ya que te permite describir la interfaz de usuario de manera declarativa.

Ejemplo Práctico Vamos a crear un componente simple que muestre un saludo.

Edita src/App.jsx:

```
function App() {
  const nombre = "Juan";
  return (
    <div>
      <h1>¡Hola, {nombre}!</h1>
      <p>Este es un ejemplo de JSX.</p>
    </div>
  );
}

export default App;
```

Explicación:

- {nombre} dentro del JSX es una expresión de JavaScript. Todo lo que esté entre llaves {} será evaluado como JavaScript.
- JSX se compila a funciones de React como React.createElement. Por ejemplo, el código anterior se traduce internamente a:

```
React.createElement("div", null,
  React.createElement("h1", null, "¡Hola, Juan!"),
  React.createElement("p", null, "Este es un ejemplo de JSX.")
);
```

---

2. Concepto de Interpolación

¿Qué es la Interpolación? La interpolación en JSX se refiere a la capacidad de insertar valores dinámicos de JavaScript dentro de tu JSX usando llaves {}. Esto permite que tu interfaz sea dinámica.

Ejemplo Práctico Vamos a crear un componente que muestre una lista de tareas dinámicamente.

Crea un nuevo componente en src/components/TodoList.jsx:

```jsx
function TodoList() {
  const tareas = ["Aprender React", "Hacer un proyecto", "Descansar"];
  const usuario = "Ana";

  return (
    <div>
      <h2>Lista de Tareas de {usuario}</h2>
      <ul>
        {tareas.map((tarea, index) => (
          <li key={index}>{tarea}</li>
        ))}
      </ul>
    </div>
  );
}

export default TodoList;
```

Luego, importa y usa este componente en src/App.jsx:

```jsx
import TodoList from './components/TodoList';

function App() {
  return (
    <div>
      <h1>Mi Aplicación React 2025</h1>
      <TodoList />
    </div>
  );
}

export default App;
```

Explicación:

- {usuario} interpola el valor de la variable usuario.
- {tareas.map(...)} interpola una lista de elementos renderizados dinámicamente.
- La interpolación puede incluir cualquier expresión de JavaScript, como operaciones matemáticas ({2 + 2}), funciones, o incluso componentes.
-

---

3. Fragment

¿Qué es un Fragment? En React, cada componente debe devolver un solo elemento. Si necesitas devolver múltiples elementos sin un contenedor adicional (como un), puedes usar un Fragment. Los Fragments no añaden nodos extra al DOM.

Ejemplo Práctico Modifica TodoList.jsx para usar un Fragment:

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

function TodoList() {
  const tareas = ["Aprender React", "Hacer un proyecto", "Descansar"];
  const usuario = "Ana";

  return (
    <Fragment>
      <h2>Lista de Tareas de {usuario}</h2>
      <ul>
        {tareas.map((tarea, index) => (
          <li key={index}>{tarea}</li>
        ))}
      </ul>
    </Fragment>
  );
}

export default TodoList;
```

Alternativa con Sintaxis Corta React también permite usar <> </> como una sintaxis más corta para Fragments:

```
function TodoList() {
  const tareas = ["Aprender React", "Hacer un proyecto", "Descansar"];
  const usuario = "Ana";

  return (
    <>
      <h2>Lista de Tareas de {usuario}</h2>
      <ul>
        {tareas.map((tarea, index) => (
          <li key={index}>{tarea}</li>
        ))}
```

```
      </ul>
    </>
  );
}

export default TodoList;
```

Explicación:

- Sin Fragment, tendrías que envolver todo en un , lo que añadiría un nodo extra al DOM.
- Con <Fragment> o <>, evitas ese nodo extra, manteniendo el DOM más limpio.

---

4. className

¿Qué es className? En JSX, usamos className en lugar de class para definir clases de CSS, ya que class es una palabra reservada en JavaScript.

Ejemplo Práctico Vamos a estilizar nuestro componente TodoList. Primero, crea un archivo src/components/TodoList.css:

```
.titulo {
  color: blue;
  font-size: 24px;
}

.tarea {
  color: green;
  margin: 5px 0;
}
```

Luego, modifica TodoList.jsx para usar className:

```
import './TodoList.css';

function TodoList() {
  const tareas = ["Aprender React", "Hacer un proyecto", "Descansar"];
  const usuario = "Ana";

  return (
    <>
      <h2 className="titulo">Lista de Tareas de {usuario}</h2>
      <ul>
```

```
            {tareas.map((tarea, index) => (
              <li key={index} className="tarea">
                {tarea}
              </li>
            ))}
          </ul>
        </>
      );
    }

    export default TodoList;
```

Explicación:

- className="titulo" aplica la clase CSS titulo al elemento <h2>.
- className="tarea" aplica la clase tarea a cada <li>.
- También puedes usar className dinámicamente, por ejemplo:

```
<li key={index} className={index % 2 === 0 ? "tarea-par" : "tarea-impar"}>
  {tarea}
</li>
```

---

5. dangerouslySetInnerHTML

¿Qué es dangerouslySetInnerHTML? Es una propiedad en React que te permite insertar HTML directamente en un componente. Es "peligroso" porque puede exponer tu aplicación a ataques XSS (Cross-Site Scripting) si el HTML no está sanitizado.

Ejemplo Práctico Vamos a mostrar un texto con HTML crudo. Modifica TodoList.jsx:

```
import './TodoList.css';

function TodoList() {
  const tareas = ["Aprender React", "Hacer un proyecto", "Descansar"];
  const usuario = "Ana";
  const htmlCrudo = "<p>Este es un <strong>texto en negrita</strong>
renderizado con dangerouslySetInnerHTML.</p>";

  return (
    <>
      <h2 className="titulo">Lista de Tareas de {usuario}</h2>
      <div dangerouslySetInnerHTML={{ __html: htmlCrudo }} />
```

```
        <ul>
          {tareas.map((tarea, index) => (
            <li key={index} className="tarea">
              {tarea}
            </li>
          ))}
        </ul>
      </>
    );
}

export default TodoList;
```

Explicación:

- dangerouslySetInnerHTML toma un objeto con la propiedad __html que contiene el HTML crudo.
- Este método debe usarse con precaución. Si el HTML proviene de una fuente no confiable (como un input de usuario), podrías introducir vulnerabilidades.

Alternativa Segura En lugar de dangerouslySetInnerHTML, considera usar una librería como html-react-parser, que veremos en el siguiente punto.

---

6. Cómo instalar librerías | html-react-parser

Instalando una Librería Vamos a instalar html-react-parser, una librería que convierte HTML en componentes React de manera segura.

Paso 1: Instalar la librería En tu terminal, ejecuta:

```
npm install html-react-parser
```

Paso 2: Usar la librería Modifica TodoList.jsx para usar html-react-parser en lugar de dangerouslySetInnerHTML:

```
import './TodoList.css';
import parse from 'html-react-parser';

function TodoList() {
  const tareas = ["Aprender React", "Hacer un proyecto", "Descansar"];
  const usuario = "Ana";
  const htmlCrudo = "<p>Este es un <strong>texto en negrita</strong>
renderizado con html-react-parser.</p>";
```

```
  return (
    <>
      <h2 className="titulo">Lista de Tareas de {usuario}</h2>
      <div>{parse(htmlCrudo)}</div>
      <ul>
        {tareas.map((tarea, index) => (
          <li key={index} className="tarea">
            {tarea}
          </li>
        ))}
      </ul>
    </>
  );
}

export default TodoList;
```

Explicación:

- parse de html-react-parser convierte el HTML en componentes React, lo que es más seguro que dangerouslySetInnerHTML.
- Esta librería también te permite personalizar cómo se renderizan los elementos HTML.

---

7. Renderización Condicional

¿Qué es la Renderización Condicional? En React, puedes renderizar elementos de manera condicional usando operadores lógicos (&&, ||), el operador ternario (?:), o sentencias if fuera del JSX.

Ejemplo Práctico Vamos a mostrar un mensaje si no hay tareas. Modifica TodoList.jsx:

```
import './TodoList.css';
import parse from 'html-react-parser';

function TodoList() {
  const tareas = []; // Cambia esto para probar con tareas
  const usuario = "Ana";
  const htmlCrudo = "<p>Este es un <strong>texto en negrita</strong>
renderizado con html-react-parser.</p>";

  return (
    <>
```

```
      <h2 className="titulo">Lista de Tareas de {usuario}</h2>
      <div>{parse(htmlCrudo)}</div>
      {tareas.length === 0 ? (
        <p>No hay tareas pendientes.</p>
      ) : (
        <ul>
          {tareas.map((tarea, index) => (
            <li key={index} className="tarea">
              {tarea}
            </li>
          ))}
        </ul>
      )}
    </>
  );
}

export default TodoList;
```

Explicación:

- Usamos el operador ternario {condición ? elementoSiVerdadero : elementoSiFalso} para decidir qué renderizar.
- También puedes usar && para renderizar algo solo si una condición es verdadera:

```
{tareas.length === 0 && <p>No hay tareas pendientes.</p>}
```

---

8. Renderización Switch Case

Renderización con Switch Case A veces, necesitas renderizar diferentes elementos según un valor. Puedes usar una estructura switch dentro de una función.

Ejemplo Práctico Vamos a mostrar un mensaje diferente según el número de tareas. Modifica TodoList.jsx:

```
import './TodoList.css';
import parse from 'html-react-parser';

function TodoList() {
  const tareas = ["Aprender React", "Hacer un proyecto"];
  const usuario = "Ana";
  const htmlCrudo = "<p>Este es un <strong>texto en negrita</strong>
renderizado con html-react-parser.</p>";
```

```
  const renderMensaje = () => {
    switch (tareas.length) {
      case 0:
        return <p>No hay tareas pendientes.</p>;
      case 1:
        return <p>Tienes 1 tarea pendiente.</p>;
      case 2:
        return <p>Tienes 2 tareas pendientes.</p>;
      default:
        return <p>Tienes muchas tareas pendientes.</p>;
    }
  };

  return (
    <>
      <h2 className="titulo">Lista de Tareas de {usuario}</h2>
      <div>{parse(htmlCrudo)}</div>
      {renderMensaje()}
      <ul>
        {tareas.map((tarea, index) => (
          <li key={index} className="tarea">
            {tarea}
          </li>
        ))}
      </ul>
    </>
  );
}

export default TodoList;
```

Explicación:

- Creamos una función renderMensaje que usa un switch para decidir qué renderizar.
- Llamamos a renderMensaje() dentro del JSX para incluir el resultado.

---

9. Distintas Variantes para Loop

Diferentes Formas de Hacer Loops En React, puedes iterar sobre listas usando map, forEach, o incluso un bucle for tradicional. La más común es map.

Ejemplo Práctico Vamos a mostrar las tareas de diferentes maneras. Modifica TodoList.jsx:

```
import './TodoList.css';
import parse from 'html-react-parser';

function TodoList() {
  const tareas = ["Aprender React", "Hacer un proyecto", "Descansar"];
  const usuario = "Ana";
  const htmlCrudo = "<p>Este es un <strong>texto en negrita</strong>
renderizado con html-react-parser.</p>";

  // Variante 1: Usando map (la más común)
  const renderConMap = tareas.map((tarea, index) => (
    <li key={index} className="tarea">
      {tarea}
    </li>
  ));

  // Variante 2: Usando forEach
  const renderConForEach = [];
  tareas.forEach((tarea, index) => {
    renderConForEach.push(
      <li key={index} className="tarea">
        {tarea}
      </li>
    );
  });

  // Variante 3: Usando un bucle for tradicional
  const renderConFor = [];
  for (let i = 0; i < tareas.length; i++) {
    renderConFor.push(
      <li key={i} className="tarea">
        {tareas[i]}
      </li>
    );
```

```
  }

  return (
    <>
      <h2 className="titulo">Lista de Tareas de {usuario}</h2>
      <div>{parse(htmlCrudo)}</div>
      <h3>Con map:</h3>
      <ul>{renderConMap}</ul>
      <h3>Con forEach:</h3>
      <ul>{renderConForEach}</ul>
      <h3>Con for:</h3>
      <ul>{renderConFor}</ul>
    </>
  );
}

export default TodoList;
```

Explicación:

- map es la forma más idiomática en React porque es declarativa y devuelve un nuevo array.
- forEach y for son más imperativos y requieren que manejes un array manualmente, pero son válidos.

---

10. Recorrer Elementos con map

Profundizando en map Ya usamos map anteriormente, pero vamos a explorar más detalles, como pasar índices y manejar objetos.

Ejemplo Práctico Vamos a trabajar con una lista de objetos. Modifica TodoList.jsx:

```
import './TodoList.css';
import parse from 'html-react-parser';

function TodoList() {
  const tareas = [
    { id: 1, texto: "Aprender React", completada: false },
    { id: 2, texto: "Hacer un proyecto", completada: true },
    { id: 3, texto: "Descansar", completada: false },
  ];
  const usuario = "Ana";
```

```
  const htmlCrudo = "<p>Este es un <strong>texto en negrita</strong>
renderizado con html-react-parser.</p>";

  return (
    <>
      <h2 className="titulo">Lista de Tareas de {usuario}</h2>
      <div>{parse(htmlCrudo)}</div>
      <ul>
        {tareas.map((tarea) => (
          <li
            key={tarea.id}
            className="tarea"
            style={{ textDecoration: tarea.completada ? "line-through" :
"none" }}
          >
            {tarea.texto} (ID: {tarea.id})
          </li>
        ))}
      </ul>
    </>
  );
}

export default TodoList;
```

Explicación:

- Usamos tarea.id como key en lugar del índice, lo que es una mejor práctica porque los IDs son únicos.
- Accedemos a las propiedades del objeto (tarea.texto, tarea.completada) dentro del map.
- Usamos un estilo inline para tachar las tareas completadas.

---

11. Helper Personalizados (helpers.js)

¿Qué son los Helpers? Los helpers son funciones reutilizables que puedes crear para manejar lógica repetitiva o compleja fuera de tus componentes.

Paso 1: Crear un archivo de helpers Crea un archivo src/helpers.js:

```
// Función para formatear el estado de una tarea
export const formatearEstadoTarea = (completada) => {
  return completada ? "Completada" : "Pendiente";
```

```
};

// Función para contar tareas completadas
export const contarTareasCompletadas = (tareas) => {
  return tareas.filter((tarea) => tarea.completada).length;
};
```

Paso 2: Usar los helpers en el componente Modifica TodoList.jsx:

```
import './TodoList.css';
import parse from 'html-react-parser';
import { formatearEstadoTarea, contarTareasCompletadas } from '../helpers';

function TodoList() {
  const tareas = [
    { id: 1, texto: "Aprender React", completada: false },
    { id: 2, texto: "Hacer un proyecto", completada: true },
    { id: 3, texto: "Descansar", completada: false },
  ];
  const usuario = "Ana";
  const htmlCrudo = "<p>Este es un <strong>texto en negrita</strong> renderizado con
html-react-parser.</p>";

  return (
    <>
      <h2 className="titulo">Lista de Tareas de {usuario}</h2>
      <div>{parse(htmlCrudo)}</div>
      <p>Tareas completadas: {contarTareasCompletadas(tareas)} de {tareas.length}</p>
      <ul>
        {tareas.map((tarea) => (
          <li
            key={tarea.id}
            className="tarea"
            style={{ textDecoration: tarea.completada ? "line-through" : "none" }}
          >
            {tarea.texto} (Estado: {formatearEstadoTarea(tarea.completada)})
          </li>
        ))}
      </ul>
    </>
```

```
  );
}

export default TodoList;
```

Explicación:

- formatearEstadoTarea convierte el booleano completada en un texto legible.
- contarTareasCompletadas calcula cuántas tareas están completadas.
- Los helpers mantienen el componente limpio y la lógica reutilizable.

12. Mi Primer Componente

¿Qué es un Componente? Un componente en React es una pieza reutilizable de la interfaz de usuario. Puede ser tan simple como un botón o tan complejo como una página completa. Los componentes pueden ser funcionales (los más comunes en 2025) o de clase (menos usados hoy en día).

Ejemplo Práctico Vamos a crear nuestro primer componente, un encabezado para nuestra aplicación de tareas.

Crea un nuevo archivo src/components/Header.jsx:

```
function Header() {
  return (
    <header>
      <h1>Gestor de Tareas</h1>
      <p>Bienvenido a tu aplicación de tareas en React 2025</p>
    </header>
  );
}

export default Header;
```

Ahora, importa y usa este componente en src/App.jsx:

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

function App() {
  return (
    <div>
      <Header />
    </div>
  );
```

```
}

export default App;
```

Explicación:

- Header es un componente funcional, que es simplemente una función que devuelve JSX.
- Lo importamos en App.jsx y lo usamos como si fuera una etiqueta HTML (<Header />).
- Los componentes deben empezar con mayúscula (por convención y para que React los reconozca como componentes).

---

13. Componente de Clase(no se usa tanto pero hacer la practica)

¿Qué es un Componente de Clase? Antes de que los componentes funcionales se volvieran predominantes (gracias a los Hooks en React 16.8), los componentes de clase eran la forma principal de manejar estado y ciclo de vida en React. Aunque hoy se usan menos, es importante entenderlos.

Ejemplo Práctico Vamos a reescribir el componente Header como un componente de clase.

Modifica src/components/Header.jsx:

```
import React, { Component } from 'react';

class Header extends Component {
  render() {
    return (
      <header>
        <h1>Gestor de Tareas</h1>
        <p>Bienvenido a tu aplicación de tareas en React 2025</p>
      </header>
    );
  }
}

export default Header;
```

Explicación:

- Importamos React y Component porque los componentes de clase extienden de React.Component.
- Definimos una clase Header que hereda de Component.
- El método render() es obligatorio y devuelve el JSX que se renderizará.
- El uso en App.jsx sigue siendo el mismo (<Header />).

Aunque este componente no hace mucho, los componentes de clase son útiles cuando necesitas manejar estado o ciclo de vida, lo que veremos a continuación.

---

14. state y setState

¿Qué es state y setState? El state es un objeto que contiene datos dinámicos de un componente. Cuando el state cambia, React vuelve a renderizar el componente. setState es el método que usas para actualizar el state en componentes de clase.

Ejemplo Práctico Vamos a crear un componente de clase que muestre un contador de tareas completadas y permita incrementarlo.

Crea un nuevo componente src/components/TaskCounter.jsx:

```
import React, { Component } from 'react';

class TaskCounter extends Component {
  constructor(props) {
    super(props);
    this.state = {
      completedTasks: 0,
    };
  }

  handleIncrement = () => {
    this.setState({ completedTasks: this.state.completedTasks + 1 });
  };

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

export default TaskCounter;
```

Ahora, actualiza src/App.jsx para incluir este componente:

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

function App() {
  return (
    <div>
      <Header />
      <TaskCounter />
    </div>
  );
}

export default App;
```

Explicación:

- this.state se inicializa en el constructor con completedTasks: 0.
- setState actualiza el estado y desencadena un nuevo renderizado.
- this.handleIncrement es un método que incrementa el contador al hacer clic en el botón.
- Usamos onClick para asociar el evento de clic al método handleIncrement.

Nota: En componentes funcionales, usaríamos el Hook useState en lugar de state y setState. Lo veremos más adelante en "Componente Funcional".

---

15. Ciclo de Vida (componentDidMount, componentWillUnmount, componentDidUpdate)

¿Qué es el Ciclo de Vida? Los componentes de clase tienen métodos especiales que se ejecutan en diferentes etapas de su "vida": cuando se montan (componentDidMount), cuando se actualizan (componentDidUpdate), y cuando se desmontan (componentWillUnmount).

Ejemplo Práctico Vamos a modificar TaskCounter para usar los métodos del ciclo de vida. Simularemos que el contador se inicializa con datos de una API y se guarda en el almacenamiento local cuando se desmonta.

Modifica src/components/TaskCounter.jsx:

```
import React, { Component } from 'react';

class TaskCounter extends Component {
  constructor(props) {
    super(props);
    this.state = {
      completedTasks: 0,
    };
```

```
  }

  componentDidMount() {
    // Simulamos una llamada a una API para obtener el número inicial de tareas
completadas
    console.log('Componente montado');
    setTimeout(() => {
      this.setState({ completedTasks: 5 }); // Supongamos que la API devuelve 5
    }, 1000);
  }

  componentDidUpdate(prevProps, prevState) {
    // Se ejecuta cada vez que el estado o las props cambian
    if (prevState.completedTasks !== this.state.completedTasks) {
      console.log('El contador de tareas ha cambiado:', this.state.completedTasks);
    }
  }

  componentWillUnmount() {
    // Guardamos el estado en localStorage antes de desmontar el componente
    console.log('Componente desmontado');
    localStorage.setItem('completedTasks', this.state.completedTasks);
  }

  handleIncrement = () => {
    this.setState({ completedTasks: this.state.completedTasks + 1 });
  };

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

export default TaskCounter;
```

Explicación:

- componentDidMount: Se ejecuta una vez que el componente se ha montado en el DOM. Aquí simulamos una llamada a una API que establece el contador en 5 después de 1 segundo.
- componentDidUpdate: Se ejecuta cada vez que el estado o las props cambian. Comparamos el estado anterior (prevState) con el actual para registrar cambios.
- componentWillUnmount: Se ejecuta justo antes de que el componente se elimine del DOM. Guardamos el estado en localStorage.

Para probar componentWillUnmount, puedes agregar un botón en App.jsx para montar/desmontar el componente:

```
import { useState } from 'react';
import Header from './components/Header';
import TaskCounter from './components/TaskCounter';

function App() {
  const [showCounter, setShowCounter] = useState(true);

  return (
    <div>
      <Header />
      <button onClick={() => setShowCounter(!showCounter)}>
        {showCounter ? 'Ocultar Contador' : 'Mostrar Contador'}
      </button>
      {showCounter && <TaskCounter />}
    </div>
  );
}

export default App;
```

Explicación Adicional:

- Al hacer clic en el botón, el componente TaskCounter se monta o desmonta, lo que activa componentDidMount y componentWillUnmount.

---

16. Componente Funcional (lo que se usa usualmente)

¿Qué es un Componente Funcional? Desde la introducción de los Hooks en React 16.8, los componentes funcionales se han convertido en la forma estándar de escribir componentes. Son más simples y pueden manejar estado y ciclo de vida con Hooks como useState y useEffect.

Ejemplo Práctico Vamos a reescribir TaskCounter como un componente funcional usando useState y useEffect.

Modifica src/components/TaskCounter.jsx:

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

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

  useEffect(() => {
    // Equivalente a componentDidMount
    console.log('Componente montado');
    setTimeout(() => {
      setCompletedTasks(5); // Simulamos una API
    }, 1000);

    return () => {
      // Equivalente a componentWillUnmount
      console.log('Componente desmontado');
      localStorage.setItem('completedTasks', completedTasks);
    };
  }, []); // Array vacío significa que se ejecuta solo al montar/desmontar

  useEffect(() => {
    // Equivalente a componentDidUpdate para completedTasks
    console.log('El contador de tareas ha cambiado:', completedTasks);
  }, [completedTasks]); // Se ejecuta cada vez que completedTasks cambia

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

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

```
export default TaskCounter;
```

Explicación:

- useState reemplaza state y setState. completedTasks es el estado, y setCompletedTasks lo actualiza.
- useEffect reemplaza los métodos del ciclo de vida:
  - El primer useEffect con un array vacío ([]) se ejecuta al montar y desmontar (equivalente a componentDidMount y componentWillUnmount).
  - El segundo useEffect con [completedTasks] se ejecuta cada vez que completedTasks cambia (equivalente a componentDidUpdate).
- Los componentes funcionales son más concisos y fáciles de leer.

---

17. Props

¿Qué son las Props? Las props (propiedades) son datos que se pasan de un componente padre a un componente hijo. Son inmutables dentro del componente hijo y permiten la comunicación entre componentes.

Ejemplo Práctico Vamos a crear un componente TaskItem que reciba una tarea como prop y la muestre.

Crea src/components/TaskItem.jsx:

```
function TaskItem(props) {
  return (
    <li>
      {props.task} (ID: {props.id})
    </li>
  );
}

export default TaskItem;
```

Ahora, crea un componente TaskList que use TaskItem. Crea src/components/TaskList.jsx:

```
import TaskItem from './TaskItem';

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

  return (
```

```
      <div>
        <h2>Lista de Tareas</h2>
        <ul>
          {tasks.map((task) => (
            <TaskItem key={task.id} id={task.id} task={task.text} />
          ))}
        </ul>
      </div>
    );
  }

  export default TaskList;
```

Actualiza src/App.jsx para incluir TaskList:

```
import { useState } from 'react';
import Header from './components/Header';
import TaskCounter from './components/TaskCounter';
import TaskList from './components/TaskList';

function App() {
  const [showCounter, setShowCounter] = useState(true);

  return (
    <div>
      <Header />
      <button onClick={() => setShowCounter(!showCounter)}>
        {showCounter ? 'Ocultar Contador' : 'Mostrar Contador'}
      </button>
      {showCounter && <TaskCounter />}
      <TaskList />
    </div>
  );
}

export default App;
```

Explicación:

- TaskItem recibe props como argumento y accede a props.task y props.id.
- En TaskList, pasamos las propiedades id y task a TaskItem usando la sintaxis de atributos (<TaskItem id={task.id} task={task.text} />).

- También puedes desestructurar las props para que el código sea más limpio:

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

export default TaskItem;
```

---

18. Renderizado Componente Dinámico

¿Qué es el Renderizado Dinámico? El renderizado dinámico implica decidir qué componente renderizar en función de una condición o estado. Esto es útil para cambiar entre diferentes vistas o componentes.

Ejemplo Práctico Vamos a crear un componente que cambie dinámicamente entre mostrar una lista de tareas y un formulario para agregar tareas.

Crea src/components/TaskForm.jsx:

```
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>
      <input
        type="text"
        value={taskText}
        onChange={(e) => setTaskText(e.target.value)}
        placeholder="Escribe una tarea"
      />
```

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

export default TaskForm;
```

Modifica src/components/TaskList.jsx para manejar el renderizado dinámico:

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

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

  const addTask = (taskText) => {
    const newTask = { id: tasks.length + 1, text: taskText };
    setTasks([...tasks, newTask]);
    setView('list'); // Volver a la lista después de agregar
  };

  const renderView = () => {
    switch (view) {
      case 'list':
        return (
          <>
            <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} />
              ))}
            </ul>
          </>
```

```
        );
      case 'form':
        return (
          <>
            <button onClick={() => setView('list')}>Volver a la Lista</button>
            <TaskForm onAddTask={addTask} />
          </>
        );
      default:
        return <p>Vista no encontrada</p>;
    }
  };

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

export default TaskList;
```

Explicación:

- Usamos el estado view para decidir qué renderizar: la lista de tareas o el formulario.
- La función renderView usa un switch para determinar qué componente o JSX devolver.
- Pasamos la función addTask como prop a TaskForm para que pueda agregar nuevas tareas.

---

19. Renderizado Componente Condicional

¿Qué es el Renderizado Condicional? Ya lo vimos en la respuesta anterior, pero aquí lo aplicaremos específicamente a componentes. Renderizamos componentes diferentes según una condición.

Ejemplo Práctico Vamos a mostrar un mensaje si no hay tareas en la lista. Modifica src/components/TaskList.jsx:

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

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

```
  ]);
  const [view, setView] = useState('list');

  const addTask = (taskText) => {
    const newTask = { id: tasks.length + 1, text: taskText };
    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} />
        ))}
      </ul>
    </>
  );

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

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

```
export default TaskList;
```

Explicación:

- Creamos dos componentes funcionales: EmptyMessage y TaskListView.
- En renderView, usamos un operador ternario para decidir si renderizar EmptyMessage o TaskListView según la longitud de tasks.
- Esto es renderizado condicional aplicado a componentes completos.

---

20. Children Components

¿Qué son los Children Components? La prop children permite pasar componentes o elementos JSX como hijos a otro componente. Es útil para crear componentes reutilizables que actúen como contenedores.

Ejemplo Práctico Vamos a crear un componente Card que actúe como un contenedor estilizado para otros componentes.

Crea src/components/Card.jsx:

```
import './Card.css';

function Card({ children }) {
  return <div className="card">{children}</div>;
}

export default Card;
```

Crea src/components/Card.css:

```
.card {
  border: 1px solid #ccc;
  border-radius: 8px;
  padding: 16px;
  margin: 16px 0;
  box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
}
```

Modifica src/components/TaskList.jsx para usar Card:

```
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' },
    { id: 2, text: 'Hacer un proyecto' },
    { id: 3, text: 'Descansar' },
  ]);
  const [view, setView] = useState('list');

  const addTask = (taskText) => {
    const newTask = { id: tasks.length + 1, text: taskText };
    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} />
        ))}
      </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;
```

Explicación:

- Card recibe children como prop y los renderiza dentro de un estilizado.
- En TaskList, envolvemos tanto la vista de la lista como el formulario en un componente Card.
- Esto hace que Card sea un contenedor reutilizable que puede envolver cualquier contenido.
