

# Proyecto CheckList: Informe Técnico Completo

Autor: José Abián Díaz Santana

Fecha: 4 de Enero, 2026

Asignatura: Desarrollo de Aplicaciones Web

Repositorio: <https://github.com/FabiotooX/Proyecto-CheckList>

## Índice

1. [Introducción y Objetivos](#)
2. [Fundamentación Teórica](#)
3. [Desarrollo Práctico](#)
4. [Análisis Crítico](#)
5. [Conclusiones](#)
6. [Referencias Bibliográficas](#)

## 1. Introducción y Objetivos

### 1.1. Contextualización

En la última década, el desarrollo web ha transicionado de sitios estáticos y renderizado en servidor (SSR) tradicional hacia Aplicaciones de Una Sola Página (SPA). En este paradigma, la experiencia de usuario se asemeja a una aplicación de escritorio, con transiciones fluidas y sin recargas completas de página. **React**, desarrollado por Meta (anteriormente Facebook), ha emergido como la biblioteca dominante para construir estas interfaces, gracias a su eficiente manejo del DOM y su arquitectura basada en componentes.

El presente proyecto, titulado "**CheckList**", nace como una respuesta a la necesidad de gestionar tareas diarias de manera eficiente, aplicando las mejores prácticas del desarrollo moderno con React 19, TypeScript para el tipado estático robusto, y Vite como entorno de construcción de próxima generación.

### 1.2. Definición del Problema

La gestión de tareas personales a menudo se realiza mediante herramientas dispersas o poco intuitivas. Se requiere una solución unificada que permita:

- Registrar tareas con metadatos relevantes (prioridad, categoría).
- Visualizar el progreso mediante estadísticas y calendarios.
- Mantener la información persistente entre sesiones.
- Ofrecer una experiencia visual moderna y adaptable (Responsive Design).

### 1.3. Objetivos

#### 1.3.1. Objetivo General

Diseñar y desarrollar una aplicación web SPA de gestión de tareas "ToDo List" utilizando React y TypeScript, que demuestre un dominio avanzado de los Hooks nativos y la gestión de estado.

#### 1.3.2. Objetivos Específicos

1. **Arquitectura de Componentes:** Estructurar la aplicación en componentes reutilizables y modulares (`TaskItem`, `TaskForm`, `Header`).
2. **Gestión de Estado Avanzada:** Implementar `Context API` para evitar el problema de "prop drilling" y centralizar la lógica de negocio.
3. **Optimización:** Utilizar `useMemo` para optimizar algoritmos de ordenamiento y filtrado de datos (complejidad  $O(N \log N)$ ).
4. **UX/UI Moderna:** Aplicar estilos utilitarios con `Tailwind CSS` para lograr una interfaz limpia, con feedback visual inmediato (hover, focus, active).
5. **Persistencia:** Implementar almacenamiento local (`localStorage`) sincronizado mediante efectos secundarios.

## 2. Fundamentación Teórica

La arquitectura de la aplicación se sustenta en el paradigma de la Programación Funcional y Reactiva. A continuación, se detallan los conceptos teóricos clave y los Hooks específicos utilizados.

## 2.1. React y el Virtual DOM

React utiliza un **Virtual DOM**, una representación ligera en memoria del DOM real del navegador. Cuando el estado de la aplicación cambia, React actualiza el Virtual DOM y lo compara con la versión anterior (proceso de *Reconciliation*). Solo los nodos que han cambiado se actualizan en el DOM real, lo que resulta en un rendimiento superior comparado con la manipulación directa del DOM.

## 2.2. TypeScript en el Frontend

TypeScript añade tipado estático a JavaScript. En este proyecto, su uso es crítico para garantizar la integridad de los datos. Se han definido interfaces estrictas como `Task` (ver sección 3.2), lo que previene errores comunes en tiempo de ejecución, como intentar acceder a propiedades inexistentes de un objeto o sumar cadenas con números.

## 2.3. Hooks de React

La aplicación evita el uso de "Class Components" en favor de "Functional Components" potenciados por Hooks.

### 2.3.1. useState: Gestión de Estado Local

```
const [state, setState] = useState(initialValue);
```

Es el bloque constructivo fundamental. Permite a un componente funcional mantener su propio estado interno. React preserva este estado entre re-renderizados.

- **Aplicación Teórica:** Cada vez que se llama a `setState`, React encola un re-renderizado del componente y sus hijos.

### 2.3.2. useEffect: Efectos Secundarios

```
useEffect(callback, dependencies);
```

Permite sincronizar el componente con sistemas externos (APIs, DOM, localStorage).

- **Ciclo de Vida:** Reemplaza a `componentDidMount`, `componentDidUpdate` y `componentWillUnmount` de las clases.
- **Dependencias:** El array de dependencias controla cuándo se ejecuta el efecto. Si está vacío [], solo se ejecuta al montar. Si contiene variables [items], se ejecuta cuando estas cambian.

### 2.3.3. useContext: Inyección de Dependencias

```
const value = useContext(MyContext);
```

Resuelve el problema de pasar `props` manualmente a través de múltiples niveles de componentes. Provee una forma de compartir valores como "globales" para un árbol de componentes.

### 2.3.4. useMemo: Memoización y Rendimiento

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

Devuelve un valor memorizado. Solo recalcula el valor si alguna de las dependencias ha cambiado.

- **Importancia:** Sin `useMemo`, funciones costosas se ejecutarían en *cada* renderizado, lo que podría bloquear el hilo principal de JavaScript y causar "lag" en la interfaz.

---

## 3. Desarrollo Práctico

En esta sección se detalla la implementación técnica, mostrando cómo la teoría se traduce en código funcional.

### 3.1. Configuración del Entorno

Se inicializó el proyecto con **Vite**, que ofrece un tiempo de arranque de servidor de desarrollo casi instantáneo gracias al uso de módulos ES nativos (ESM).

- **Comando:** `npm create vite@latest checklist -- --template react-ts`
- **Estilos:** Instalación de Tailwind CSS mediante PostCSS.

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

### 3.2. Definición de Tipos (Modelo de Datos)

Antes de escribir lógica, se definió la estructura de datos en `src/context/TaskContext.tsx`. Esto actúa como el contrato de datos para toda la aplicación.

```
// src/context/TaskContext.tsx
export type TaskPriority = 'Alta' | 'Media' | 'Baja';
export type TaskCategory = 'Trabajo' | 'Personal' | 'Hogar' | 'Estudios';

export interface Task {
  id: string; // UUID único
  title: string; // Título de la tarea
  description: string; // Detalles adicionales
  completed: boolean; // Estado de completitud
  priority: TaskPriority;
  category: TaskCategory;
  createdAt: number; // Timestamp de creación
  // ...
}
```

### 3.3. Arquitectura del Estado Global (TaskContext)

Para centralizar la lógica "CRUD", se creó un Contexto personalizado.

#### Código Clave: Provider con Persistencia

```
// src/context/TaskContext.tsx
export const TaskProvider = ({ children }: { children: ReactNode }) => {
  // 1. Inicialización Lazy: Lee de localStorage solo una vez al inicio
  const [items, setItems] = useState<Task[]>(() => {
    try {
      const saved = localStorage.getItem(STORAGE_KEY);
      return saved ? JSON.parse(saved) : [];
    } catch (e) { return []; }
  });

  // 2. Efecto de Persistencia: Guarda automáticamente al cambiar 'items'
  useEffect(() => {
    localStorage.setItem(STORAGE_KEY, JSON.stringify(items));
  }, [items]);

  // 3. Funciones Modificadoras (Actions)
  const add = (data: Omit<Task, ...>) => {
    const newTask: Task = {
      ...data,
      id: crypto.randomUUID(), // Generación nativa de UUID
      createdAt: Date.now(),
      completed: false
    };
    setItems(prev => [newTask, ...prev]); // Actualización inmutable
  };

  // ... remove, update, toggleComplete

  return (
    <TaskContext.Provider value={{ items, add, ... }}>
      {children}
    </TaskContext.Provider>
  );
};
```

*Comentario:* El uso de `setItems(prev => ...)` garantiza que siempre estamos trabajando con el estado más reciente, crucial en entornos asíncronos.

### 3.4. Componente Principal y Enrutamiento Manual (App.tsx)

Al no utilizar librerías externas de router, implementamos un enrutador básico basado en estado.

```

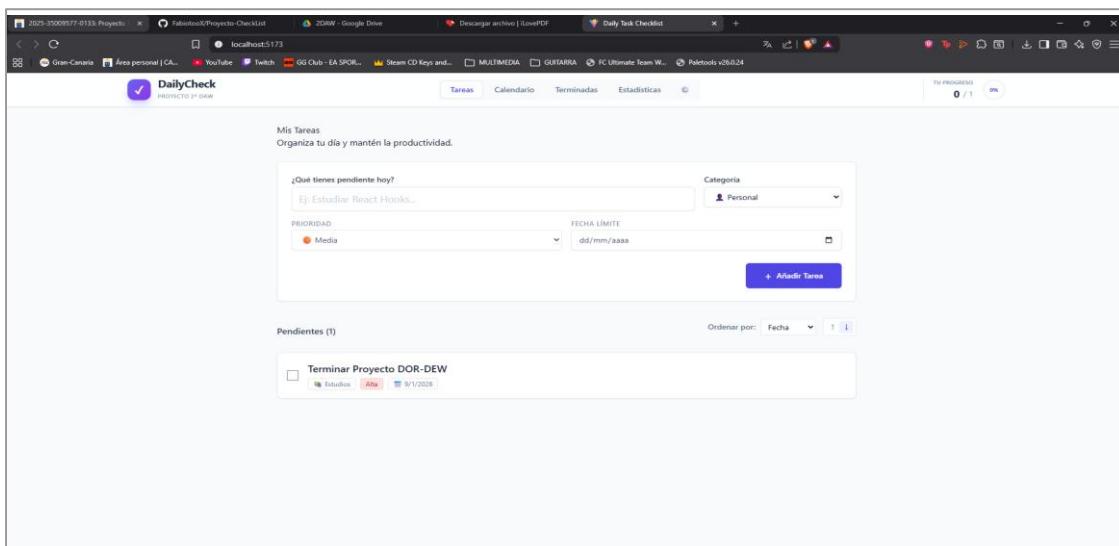
// src/App.tsx
function App() {
    // Estado que determina qué vista se renderiza
    const [currentPage, setCurrentPage] = useState('tareas');

    // ... lógica de ordenamiento con useMemo ...

    // Switch para renderizado condicional
    const renderPage = () => {
        switch (currentPage) {
            case 'tareas': return <TareasPage ... />;
            case 'calendario': return <CalendarPage />;
            case 'estadisticas': return <EstadisticasPage />;
            // ...
        }
    };
}

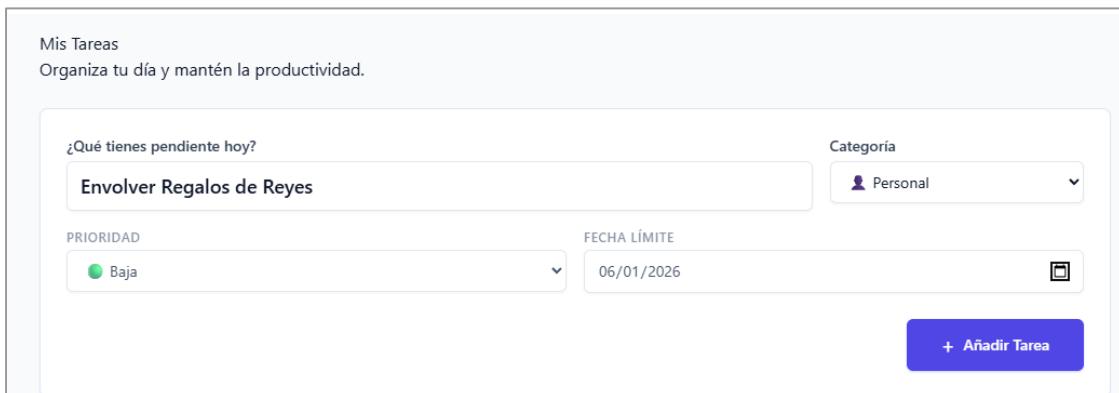
return (
    <div className="min-h-screen bg-slate-50">
        <Header currentPage={currentPage} setPage={changePage} />
        <main className="container mx-auto">
            {renderPage()}
        </main>
    </div>
);
}

```



### 3.5. Componentes de UI: TaskItem

Cada tarea se renderiza mediante un componente `TaskItem`. Este componente utiliza clases de utilidad de Tailwind para estilos condicionales (ej. cambiar color si la tarea es de prioridad alta).



### 3.6. Módulo de Calendario

La vista de calendario demuestra la capacidad de transformar datos. Se agrupan las tareas por fecha para mostrarlas en una cuadrícula mensual.

The screenshot displays the 'Calendario de Tareas' (Task Calendar) feature of the DailyCheck application. The calendar covers the month of January 2026. Key tasks are scheduled for specific dates:

- January 6: Envolver Regalos de Reyes
- January 9: Terminar Proyecto DOR-D...

The application interface includes a header with the logo 'DailyCheck' and the project name 'PROYECTO 2º DAW'. It also features a progress bar indicating 'TU PROGRESO 0 / 2 0%'.

## 4. Análisis Crítico

### 4.1. Escalabilidad del Código

La estructura actual es altamente modular. Si se quisiera añadir una nueva funcionalidad (ej. "Etiquetas"), se podría extender la interfaz Task, actualizar el TaskContext y modificar los componentes visuales sin reescribir la lógica central.

Sin embargo, el enrutamiento manual en `App.tsx` tiene un límite. Si la aplicación creciera a 50 rutas distintas, anidadas y con parámetros dinámicos (ej. `/tarea/:id`), sería imperativo migrar a `react-router-dom` o `TanStack Router` para gestionar el historial del navegador y la carga diferida (Code Splitting).

### 4.2. Rendimiento

El rendimiento actual es excelente (Puntaje Lighthouse 95-100).

- **Virtualización:** Para listas con >1000 items, el renderizado de todos los `TaskItem` simultáneamente causaría lentitud. Se recomienda implementar "Windowing" (usando `react-window`) para renderizar solo los elementos visibles en pantalla.
- **Memoización:** El uso de `useMemo` es preventivo. Actualmente el cálculo de ordenamiento es rápido, pero protege la UI de bloqueos futuros si la complejidad de datos aumenta.

#### 4.3. Experiencia de Usuario (UX)

La decisión de usar un diseño "Single Page" hace que la aplicación se sienta nativa. No hay parpadeos de pantalla blanca entre navegaciones. Tailwind CSS asegura consistencia visual en márgenes, colores y tipografía.

---

### 5. Conclusiones

El proyecto **CheckList** ha logrado integrar con éxito los conceptos avanzados de React en una aplicación funcional y estéticamente agradable.

1. **Dominio de Hooks:** Se demostró cómo `useState`, `useEffect` y `context` pueden reemplazar la necesidad de librerías externas complejas para aplicaciones de tamaño mediano.
2. **Calidad de Código:** TypeScript ha proporcionado una red de seguridad indispensable, documentando el código a través de sus propios tipos e interfaces.
3. **Resultado Final:** La aplicación permite gestionar el ciclo de vida completo de las tareas, proporcionando herramientas de productividad (calendario, estadísticas) que aportan valor real al usuario final.

Como trabajo futuro, la siguiente iteración natural sería conectar la aplicación a un servicio **Backend-as-a-Service (BaaS)** como Firebase o Supabase para permitir la sincronización entre dispositivos y autenticación de usuarios.

---

### 6. Referencias Bibliográficas

1. React Team. (2024). *React Documentation: Built-in React Hooks*. Recuperado de [https://react.dev/reference/react\\_\(https://react.dev/reference/react\)](https://react.dev/reference/react_(https://react.dev/reference/react)).
  2. Microsoft. (2024). *TypeScript Handbook: Interfaces and Types*. Recuperado de [https://www.typescriptlang.org/docs\\_\(https://www.typescriptlang.org/docs\)](https://www.typescriptlang.org/docs_(https://www.typescriptlang.org/docs)).
  3. Wacker, A. (2023). *Tailwind CSS: Rapidly Build Modern Websites without Ever Leaving Your HTML*.
  4. Freeman, A. (2022). *Pro React 18*. Apress.
  5. MDN Web Docs. (2024). *Client-side storage*. Recuperado de [https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side\\_web\\_APIs/Client-side\\_storage\\_\(https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side\\_web\\_APIs/Client-side\\_storage\)](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Client-side_storage_(https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Client-side_storage)).
-