El Problema Actual: Nuestra TodoList es genial, pero si refrescas la página, ¡todas las tareas desaparecen! Esto se debe a que useState solo guarda datos en la memoria del navegador, que se borra al recargar.

La Solución: Usaremos una **base de datos en la nube**. Para mantener la coherencia con tus tutoriales anteriores, la mejor opción es **Cloud Firestore**, la base de datos de Firebase.

¿Qué es Firestore?

Es una base de datos NoSQL (como MongoDB) que vive en la nube. Es "en tiempo real", lo que significa que si los datos cambian en la base de datos, ¡tu aplicación se actualizará automáticamente! Es la pareja perfecta para React.

Tutorial: Conectando tu App de React a una Base de Datos (Firestore)

Objetivo: Modificar nuestra TodoList para que lea y escriba tareas en una base de datos de Firestore, haciendo que los datos sean **persistentes** (sobreviven a las recargas) y **en tiempo real**.

Paso 1: Configurar Firebase y Firestore (en la Nube)

Esto es similar al inicio del tutorial de hosting, pero esta vez nos centraremos en la base de datos.

1. Ve a la Consola de Firebase.
2. Si ya creaste un proyecto (ej: mi-app-react-demo), ábrelo. Si no, crea uno nuevo.
3. En el menú de la izquierda, haz clic en **"Build"** y luego en **"Firestore Database"**.
4. Haz clic en el botón grande **"Crear base de datos"**.
5. **¡IMPORTANTE!** Te preguntará por las reglas de seguridad. Elige **"Comenzar en modo de prueba"** (Start in test mode).
    - Esto permite que cualquiera lea y escriba en tu base de datos por 30 días. Es perfecto para el desarrollo.
    - **Advertencia:** Nunca uses el modo de prueba en producción.
6. Elige una ubicación para tu base de datos (la más cercana a ti, como us-central, está bien).
7. Haz clic en "Habilitar". ¡Ya tienes una base de datos en la nube!

Paso 2: Conectar nuestra App de React a Firebase

Ahora, le enseñaremos a nuestra app de React cómo "hablar" con esta base de datos.

1. En tu terminal, dentro de tu proyecto (mi-app-modular), instala el paquete de Firebase:

Bash

npm install firebase

2. En la consola de Firebase, ve a la configuración de tu proyecto (haz clic en el ícono de engranaje ⚙️ junto a "Project Overview" -> "Configuración del proyecto").
3. Baja hasta "Tus apps" y haz clic en el ícono web </> para "Añadir app".
4. Dale un apodo (ej: "Mi App Web") y haz clic en "Registrar app".
5. Firebase te dará un objeto de configuración (const firebaseConfig = { ... }). **¡Copia este objeto!**
6. En tu proyecto de VS Code, dentro de src, crea un nuevo archivo llamado firebaseConfig.js. Pega tu configuración allí y expórtala:

JavaScript

```
// src/firebaseConfig.js

import { initializeApp } from "firebase/app";

import { getFirestore } from "firebase/firestore";


// Tu configuración personal de Firebase

const firebaseConfig = {

  apiKey: "TU_API_KEY",

  authDomain: "TU_AUTH_DOMAIN",

  projectId: "TU_PROJECT_ID",

  storageBucket: "TU_STORAGE_BUCKET",

  messagingSenderId: "TU_MESSAGING_SENDER_ID",

  appId: "TU_APP_ID"

};


// Inicializar Firebase

const app = initializeApp(firebaseConfig);


// Inicializar Firestore y exportarlo para usarlo en la app

export const db = getFirestore(app);
```

🔒 Práctica Recomendada (Opcional pero importante):

No es bueno subir tus API keys a GitHub. La forma correcta es usar variables de entorno.

1. Crea un archivo .env en la raíz de tu proyecto.
2. Dentro, añade tus keys: REACT_APP_API_KEY=TU_API_KEY
3. En firebaseConfig.js, léelas así: apiKey: process.env.REACT_APP_API_KEY
4. Asegúrate de que tu archivo .gitignore incluya la línea .env.

Paso 3: Refactorizar TodoList.js - Leer Datos (GET)

Aquí viene la magia. Vamos a reemplazar el useState hardcodeado por una llamada a Firestore en tiempo real.

1. Abre src/components/TodoList/TodoList.js.
2. Importa lo necesario de Firebase y React:

JavaScript

```
// Arriba de todo en TodoList.js

import React, { useState, useEffect } from 'react'; // <-- Añade useEffect

import { db } from '../../firebaseConfig'; // <-- Importa nuestra config

import { collection, query, orderBy, onSnapshot, addDoc, doc, updateDoc, deleteDoc,
serverTimestamp } from "firebase/firestore"; // <-- Importa funciones de Firestore

import './TodoList.css';

import TodoItem from '../TodoItem/TodoItem';
```

3. Vamos a cambiar cómo inicializamos el estado tasks y usamos useEffect para **escuchar** los cambios de la base de datos.

JavaScript

```
const TodoList = () => {

  // El estado 'tasks' ahora empieza vacío

  const [tasks, setTasks] = useState([]);

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


  // --- LEER TAREAS (GET) ---

  // useEffect se ejecutará cuando el componente se monte

  useEffect(() => {
```

```
  // 1. Creamos una referencia a nuestra colección "tasks" en Firestore
  const collectionRef = collection(db, "tasks");

  // 2. Creamos una consulta (query) para ordenar las tareas por fecha
  const q = query(collectionRef, orderBy("createdAt", "asc"));

  // 3. onSnapshot es el ¡ESCUCHADOR EN TIEMPO REAL!
  // Se dispara una vez al inicio y luego CADA VEZ que los datos cambian
  const unsubscribe = onSnapshot(q, (querySnapshot) => {
    const newTasks = [];
    querySnapshot.forEach((doc) => {
      newTasks.push({
        ...doc.data(),
        id: doc.id // El ID del documento es importante
      });
    });
    setTasks(newTasks); // Actualizamos nuestro estado de React
  });

  // Esta función de limpieza se ejecuta cuando el componente se "desmonta"
  // Evita fugas de memoria
  return () => unsubscribe();

}, []); // El '[]' asegura que esto se ejecute solo una vez

// ... (El resto del componente)
```

🔍 **Desglose:**

- **collection(db, "tasks")**: Apunta a una "colección" (como una tabla) llamada tasks. Si no existe, Firestore la creará.

- **query(..., orderBy(...))**: Le pedimos a Firestore que nos dé las tareas ordenadas por su fecha de creación.
- **onSnapshot**: Este es el hook principal. Nos suscribimos a cualquier cambio en esa consulta. Cuando algo cambie (añadas, borres, edites), este código se ejecuta de nuevo y setTasks actualiza la UI. ¡Es Reactividad Pura!

Paso 4: Refactorizar las Funciones de Escritura (CREATE, UPDATE, DELETE)

Ahora, en lugar de solo actualizar el estado local, ¡actualizaremos la base de datos!

1. **Añadir Tarea (CREATE):**

JavaScript

```
const handleAddTask = async (e) => { // La hacemos 'async'
  e.preventDefault();
  if (inputValue.trim() === '') return;

  // ¡En lugar de solo 'setTasks', escribimos en la BD!
  await addDoc(collection(db, "tasks"), {
    text: inputValue,
    isComplete: false,
    createdAt: serverTimestamp() // Marca de tiempo de Firebase
  });

  setInputValue('');
  // NOTA: No necesitamos 'setTasks' aquí.
  // ¡'onSnapshot' detectará el nuevo documento y actualizará el estado por nosotros!
};
```

2. **Marcar Tarea (UPDATE):**

JavaScript

```
const handleToggleComplete = async (task) => { // Pasamos el objeto 'task' entero
  // 1. Creamos una referencia al documento específico por su ID
  const taskRef = doc(db, "tasks", task.id);
```

```
  // 2. Actualizamos ese documento
  await updateDoc(taskRef, {
    isComplete: !task.isComplete // Invertimos el valor
  });
  // De nuevo, ¡onSnapshot se encarga de actualizar la UI!
};
```

3. **Borrar Tarea (DELETE):**

```
JavaScript
const handleDeleteTask = async (idToDelete) => {
  // 1. Creamos una referencia al documento
  const taskRef = doc(db, "tasks", idToDelete);

  // 2. Borramos el documento
  await deleteDoc(taskRef);
  // ¡onSnapshot se encarga del resto!
};
```

4. Actualizar TodoItem (¡Importante!):

Nuestro componente TodoItem ahora llama a onToggleComplete(task.id) pero nuestra nueva función espera el objeto task completo. Ajustemos la llamada en TodoList.js (en la parte del return):

```
JavaScript
// ... dentro del return de TodoList.js
      {tasks.map(task => (
        <TodoItem
          key={task.id}
          task={task}
          // ¡Pasa la función correctamente!
          onToggleComplete={() => handleToggleComplete(task)} // Pasa el objeto 'task'
          onDeleteTask={handleDeleteTask} // Esta ya pasaba solo el ID
```

```
        />
      ))}
// ...
```

Y ajusta TodoItem.js para que llame a onToggleComplete sin argumentos:

JavaScript

```
// ... en TodoItem.js
      <input
        type="checkbox"
        checked={task.isComplete}
        onChange={onToggleComplete} // ¡Ya no pasamos el ID aquí!
      />
// ...
```

🚀 **¡Prueba Final!**

1. Ejecuta npm start.
2. Tu lista de tareas aparecerá vacía (o con las tareas que tenías, si useEffect falló en limpiar).
3. **Añade una nueva tarea.** Verás que aparece.
4. **Abre la consola de Firebase** en tu navegador. Ve a la sección "Firestore Database". Verás tu colección tasks y un documento nuevo con tu tarea. ¡Acabas de escribir en la nube!
5. **Refresca tu página web (F5).**
6. **Tu tarea sigue ahí.** useEffect la ha leído desde Firestore.
7. **Abre tu app en una segunda pestaña del navegador.** Verás la misma lista.
8. **Marca la tarea como completa en una pestaña.** ¡Observa cómo se marca automáticamente en la segunda pestaña!

¡Felicidades! Has construido una aplicación React "full-stack" y en tiempo real.

Próximos Pasos

Ahora que tu aplicación tiene una base de datos real y persistente, **estás listo para los tutoriales de despliegue.**