Task Scheduler - Design Document

1. Overview

The Task Scheduler prototype empowers users to seamlessly schedule, edit, and execute one-time and recurring tasks. Designed with a focus on simplicity and scalability, the system prioritizes efficiency while avoiding unnecessary complexity. Given the time constraints for implementation, this document outlines a strategic approach to scaling the system effectively.

2. Critical Design Decisions & Tradeoffs

Minimal Component Design

- **Backend:** Single Node.js/Express service for task scheduling and API handling.
- **Frontend:** React with Context API for state management, reducing reliance on external global stores.
- **Task Execution:** Uses in-memory scheduling (setTimeout and setInterval) for lightweight task execution, avoiding unnecessary dependencies like external cron jobs.
- **Data Storage:** In-memory approach for this prototype, with an easy transition to a database for scalability.

Tradeoffs

- **Pros**:
  - **Simplicity**: The system is lightweight and easy to maintain.
  - **Low latency:** Immediate task execution without external queue dependencies.
  - Easy to scale in a distributed environment when moving to a database.
- **Cons:**
  - Not durable for high availability since in-memory storage is volatile.
  - No built-in retry mechanism if the server crashes.

3. System Components & Communication

Backend (Node.js + Express + TypeScript)

- **Components:**
  - taskController.ts: Manages API requests for tasks.
  - taskScheduler.ts: Handles scheduling and execution of tasks in memory.
  - taskRoutes.ts: Defines API endpoints.
- **Communication:**
  - REST API endpoints allow frontend interaction with the backend.
  - Task execution happens within the backend service itself.

Frontend (React + Styled Components + Context API)

- **Components:**
  - TaskModal.tsx: Modal UI for task creation/editing.
  - TaskList.tsx: Displays scheduled tasks.
  - TaskLog.tsx: Displays executed tasks.
  - TaskContext.tsx: Manages global state and API communication.
- **Communication:**
  - Uses Axios to fetch and modify task data from the backend.

4. Scaling Considerations

Scaling Up (Hundreds/Thousands/Millions of Tasks)

- Move from In-Memory Storage to a Database
  - PostgreSQL for structured data (better ACID compliance).
  - Redis for task execution caching (fast retrieval).
- Introduce Task Queues for Better Execution
  - Use a message queue system (RabbitMQ, Kafka) for distributed task processing.
  - Separate workers to handle task execution instead of blocking the main API service.
- Load Balancing
  - Multiple backend instances behind a load balancer (NGINX, AWS ALB) to handle API requests at scale.
  - Database replication and partitioning to distribute data load.

Chokepoints & Limitations

- In-Memory Execution Limitations:
  - If tasks grow beyond a few hundred per second, memory constraints will slow execution.
  - A persistent store and dedicated workers are needed to handle large-scale workloads.
- Single Server Bottleneck:
  - The current setup is single-threaded; high traffic will cause slow API responses.
  - Implementing clustering or moving execution to workers would solve this.

Scaling Down (Small Deployments)

- Lightweight SQLite DB Instead of PostgreSQL for easier persistence.
- Reduce API Overhead by caching task results in-memory for quick retrieval.