Task Scheduler App Design Document

Overview This document outlines the design for a task scheduler application that schedules tasks, updates task statuses, and notifies users upon task execution. The system must be simple, scalable, and efficient, ensuring minimal delays and accurate notifications.

System Components

1. **Task Scheduler API**
    - Handles task creation, deletion, updation, listing and status updates.
    - Provides endpoints for task management.
2. **Database**
    - Stores tasks, their statuses, and user information using MongoDB with Mongoose for schema management and interactions.
    - Should support efficient read and write operations.
3. **Task Processor**
    - Monitors tasks and updates their statuses.
    - Executes tasks and moves them to the execution list.
    - Handles notifications to users.
4. **Notification Service**
    - Sends calendar invites to users when tasks are in the queue.
    - Sends email notifications to users when tasks are executed.
5. **WebSocket Integration**
    - Provides real-time updates to users about task status changes.

Critical Design Decisions

1. **Component Separation:** Separate components for task management, processing, and notification to ensure modularity and ease of maintenance.
2. **Event-Driven Architecture:** Use event-driven architecture for task processing and notifications to ensure timely updates and notifications.
3. **Database Choice:** Use MongoDB for its scalability and flexibility with document-based data storage.
4. **Scalability Considerations:** Ensure that each component can be scaled independently to handle increasing load.

Trade-offs

1. **Complexity vs. Simplicity:** Avoid unnecessary complexity by keeping components minimal and focused on their specific roles.
2. **Latency vs. Throughput:** Balance between processing tasks quickly and handling a high volume of tasks efficiently.
3. **Immediate vs. Scheduled Processing:** Opt for immediate processing of tasks nearing execution time to ensure timely execution and notification.

Components and Services

1. **Task Scheduler API**
    - Endpoints:
        - Create Task: `POST /tasks`
        - Delete Task: `DELETE /tasks/{id}`
        - List Tasks: `GET /tasks`
        - Update Task: `PUT /tasks/{id}`
        - Execute task list: `GET /executedTasks`
    - Responsibilities:
        - Accepts and validates task creation requests.
        - Manages task life-cycle (creation, deletion, updation, listing).
2. **Database**
    - Schema:
        - Tasks Model:
            - id: String
            - title: String
            - description: String
            - status: String (pending, queued, executed)
            - execution_time: DateTime
            - user_email: String
    - Responsibilities:
        - Stores tasks and user information.
        - Supports efficient querying for tasks nearing execution time.
3. **Task Processor**
    - Components:
        - Task Monitor: Checks for tasks nearing execution time and updates their status to queued.

        - Task Executor: Executes tasks and moves them to the execution list.
    - Responsibilities:
        - Polls database for tasks nearing execution time.
        - Updates task statuses and executes tasks.
        - Triggers notifications upon task execution.
4. **Notification Service**
    - Components:
        - Email Sender: Sends email notifications to users.
        - Calendar Inviter: Sends calendar invites to users when tasks are queued.
    - Responsibilities:
        - Listens for task execution events.
        - Sends email notifications to users when tasks are executed.
        - Sends calendar invites to users when tasks are queued.
5. **WebSocket Integration**
    - Components:
        - WebSocket Server: Manages WebSocket connections.
    - Responsibilities:
        - Provides real-time updates to users about task status changes.
        - Ensures users are informed promptly about task status updates.

Scaling Considerations

1. **Database:**
    - Use indexing on `execution_time` and `status` to improve query performance.
    - Scale vertically or horizontally (e.g., sharding) as task volume increases.
2. **Task Processor:**
    - Deploy multiple instances to handle increased load.
    - Use distributed task queues (e.g., RabbitMQ, Kafka) to manage task distribution and processing.
3. **Notification Service:**
    - Deploy multiple instances to handle high volume of notifications.
    - Use a dedicated email service (e.g., AWS SES, SendGrid) for reliable email delivery.

Potential Chokepoints

1. **Database Performance:**

- High volume of tasks could lead to slow queries and updates. Mitigate with indexing and sharding.

2. **Task Processor Load:**
    - A large number of tasks nearing execution time could overwhelm the processor. Mitigate by distributing load across multiple instances.
3. **Notification Bottleneck:**
    - High volume of email notifications could delay sending. Mitigate by scaling notification service and using dedicated email services.

Summary This design for the task scheduler application focuses on simplicity, modularity, and scalability. By separating concerns into distinct components and planning for scaling, the system can handle increasing load efficiently. Key considerations include efficient database operations, scalable task processing, reliable notifications, and real-time updates using WebSockets. This design ensures timely task execution and user notifications while maintaining a manageable level of complexity.