Design Overview

I suggest separating the following services:

- **Task Registration Service**: Provides APIs to register, edit, delete, and list tasks.
- **Task Scheduler Service**: Manages task schedules and ensures tasks are executed on time.
- **Task Execution Service**: Executes tasks as they become due.
- **Logging and Monitoring**

We can deploy these services using Docker containers and Kubernetes, utilizing load balancers and auto-scaling policies. For cost efficiency, a serverless approach (e.g., AWS Lambda, Azure Functions, Google Cloud Functions...) can be considered. For heavy tasks, we can use node Worker threads, and for high load scenarios, we can also utilize node Cluster to improve performance and scalability.

Services

1. **Task Registration Service**: Provides APIs for task registration, editing, deletion, and listing. For storage, consider options like DynamoDB, MongoDB, Cassandra, or PostgreSQL. In our demo, we'll use MongoDB due to its flexibility and scalability.

API Endpoints:

- `POST /tasks`: Register a new task.

- `PUT /tasks/:id`: Edit an existing task.

- `DELETE /tasks/:id`: Delete a task.

- `GET /tasks`: List all scheduled tasks.

2. **Task Scheduler Service**: Manages task schedules and ensures timely execution.

<u>One-time Tasks:</u>

- Use libraries like Bull or Agenda for delayed task execution. Bull uses Redis, which is an in-memory data store, but Redis also supports data persistence to ensure that scheduled tasks and their states are not lost.

- Alternatively, consider Kafka or RabbitMQ for message-based delay.

- In our demo, we'll use Bull for its integration and robust scheduling capabilities.

After execution, make sure to remove the task from the scheduler service to prevent it from being executed again.

Recurring Tasks

- Libraries like Bull or Agenda support recurring tasks using Cron syntax.

- Dedicated cron libraries (e.g., Cron, node-cron) are also suitable.

- Kafka or RabbitMQ can achieve cron-like behavior.

- We'll stick with Bull (using Redis) for both one-time and recurring tasks.

3. **Task Execution Service**: Executes tasks when due.

    1. Retrieve due tasks from the Task Scheduler Service.

    2. Execute task logic.

    3. Log execution time and status.

4. **Logging and Monitoring**:

- Consider ELK stack (Elasticsearch, Logstash, Kibana) or Prometheus with Grafana.

- In our demo, we'll use MongoDB for logging.

5. Client Side

- For the client side, I recommend using React along with related technologies. For the demo I will use React Router Dom, React Hook Form, RTK-Query, and MUI…

Data Flow

1. The client registers a task through the Client Interface.
2. The Task Registration Service processes the request, stores the task in the database, and forwards it to the Task Scheduler Service.
3. The Task Scheduler Service schedules the task for execution at the appropriate time.
4. When a task is due, the Task Scheduler Service triggers the Task Execution Service to execute the task.
5. The Task Execution Service logs the execution time and status, ensuring accurate tracking and monitoring of tasks.

Demo

For the demo, the following technologies will be used:

- **Backend:** Nest.js framework with Bull for task scheduling, and Redis for in-memory data storage.
- **Storage:** MongoDB for data persistence. Since this is a demo and MongoDB requires a replica set for transactions, transactions will not be used. However, it is important to use transactions in production to ensure data consistency.
- For the demo, I will run a single service. After each startup, the service will retrieve all active tasks from storage in chunks and insert them into Bull.

To simplify the demo setup and usage, no username and password will be required for MongoDB and Redis.

Setup:

1. **Start Services:**
    - Run `docker-compose up -d` to start the necessary services in detached mode.
2. **Backend Setup:**
    - Navigate to the `backend` directory.
    - npm install
    - Rename `example.env` to `.env`.
    - Update the `.env` file with suitable values. You can keep the existing values and only change the port if it is already in use.
    - Start the backend service: `npm start`.
3. **Frontend Setup:**
    - Navigate to the `frontend` directory.
    - npm install
    - Rename `example.env` to `.env`.

- Update the `.env` file with suitable values. You can keep the existing values and only change the port if it is already in use.
- Start the frontend service: `npm start`.