Distributed Job Scheduling System

System Overview & Architecture

1. Introduction

This document provides a **high-level overview** of a distributed job scheduling system designed to support **one-time, scheduled, and recurring jobs** with reliability, scalability, and fault tolerance in mind.

The goal of the system is to:

- Allow users to submit jobs for immediate or future execution
- Support recurring jobs using cron-based schedules
- Execute jobs asynchronously
- Handle failures gracefully
- Scale horizontally as workload increases

This document intentionally focuses on **system design, architectural decisions, and production-readiness**, rather than low-level implementation details.

---

2. High-Level System Overview

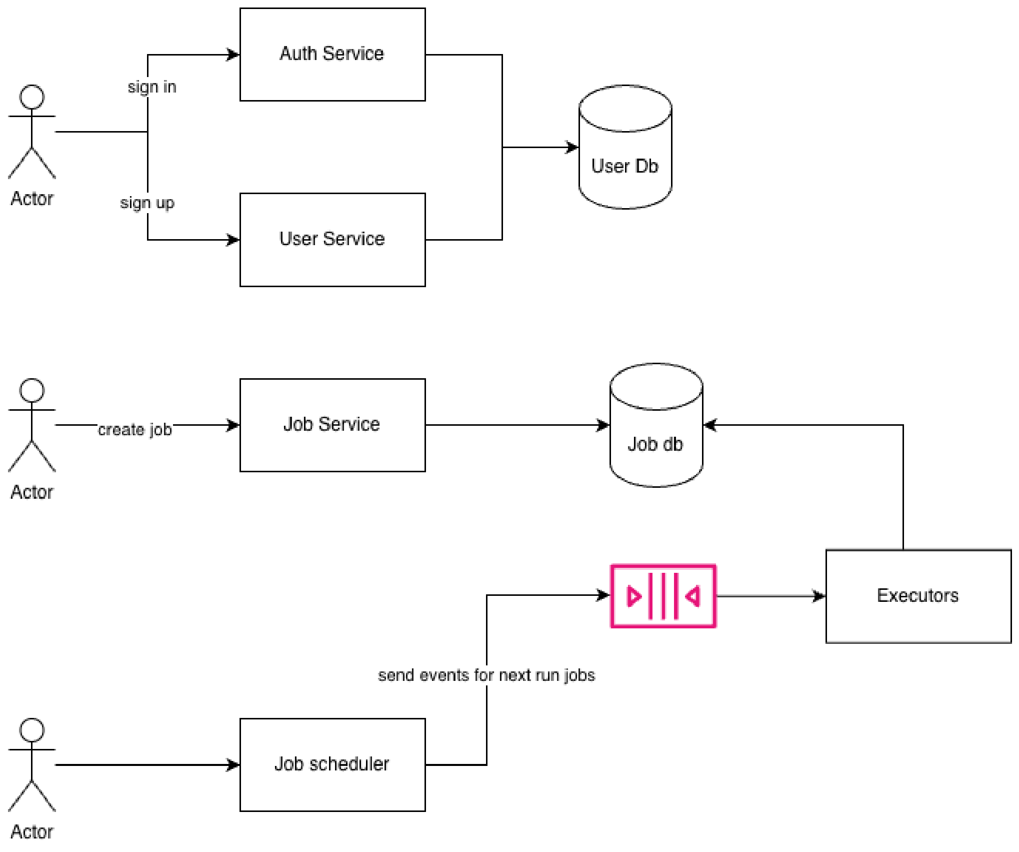
The system is built as a set of **loosely coupled, independently deployable services**, each with a well-defined responsibility.

At a high level:

- Users interact with the system through REST APIs
- Jobs are persisted in a database as the source of truth
- A scheduler dispatches due jobs asynchronously
- Executors consume jobs and perform execution
- Kafka is used to decouple scheduling from execution

High-Level Architecture

Core Flow



Key Architectural Principles

- **Database as Source of Truth**
  All job states, schedules, and retries are persisted in the database.
- **Kafka for Decoupling**
  Kafka is used for asynchronous delivery and buffering, not for correctness.
- **Clear Separation of Concerns**
  Scheduling, execution, and job management are handled by separate services.

Containerisation & Scalability

Each service in the system is:

- Containerised
- Stateless
- Independently deployable

Scalability Characteristics

- **Job Service**
  - Scales horizontally to handle API throughput
- **Scheduler Service**
  - Multiple instances can run for availability
  - Database locking ensures a job is dispatched only once
- **Executor Service**
  - Scales horizontally to increase execution throughput
  - Kafka partitions enable parallel consumption
- **Kafka**
  - Handles backpressure and buffering during spikes

This design allows the system to scale by **adding more containers**, without changing application logic.

Configuration & Secrets Management

All services follow a **configuration-first design**.

Configuration Strategy

- Each service reads configuration from:
  - `service.properties` (non-sensitive configuration)
  - `secrets.properties` (sensitive configuration)
- Configuration files are external to the application binary
- Services load configuration using: **SPRING_CONFIG_LOCATION**

Benefits

- No secrets are hardcoded in code or containers
- Configuration can be updated without rebuilding images.

- Same container image can be used across environments
  This approach ensures **secure and flexible configuration management** across all services.

6. Services Overview & Production-Ready Decisions

6.1 User Service

Responsibility

- User registration
- Default role assignment

Production-Ready Design

- Stateless service
- Database-backed persistence
- Role-based access model
- Externalised configuration

6.2 Authentication Service

Responsibility

- User authentication
- JWT token issuance

Production-Ready Design

- JWT-based stateless authentication
- Token-based authorization
- Decoupled from job execution logic

6.3 Job Service

Responsibility

- Job creation and management
- Validation of job requests
- Scheduling metadata computation

Production-Ready Design

- JWT + scope-based authorization
- Deterministic calculation of `nextRunAt`
- Database-backed job lifecycle
- Stateless and horizontally scalable

6.4 Scheduler Service

Responsibility

- Identify due jobs
- Dispatch jobs for execution

Production-Ready Design

- Database-level locking to avoid duplicate dispatch
- Batch-based processing
- Controlled inline retries
- Safe sleep strategy with capped idle time
- Jobs marked `RUNNING` before Kafka dispatch to ensure correctness

6.5 Executor Service

Responsibility

- Consume job events from Kafka
- Execute jobs
- Update job state

Production-Ready Design

- Kafka-based asynchronous execution
- Strategy + Registry pattern for extensibility
- Clear validation vs execution failure semantics
- Safe handling of recurring job rescheduling
- Idempotent execution requirements

Job Lifecycle & State Management

Jobs follow a clearly defined lifecycle:

```
SCHEDULED → RUNNING → COMPLETED
                    →  FAILED
```

For recurring jobs:

RUNNING → SCHEDULED (nextRunAt updated, attempts reset)

Key principles:

- Scheduler owns the transition to `RUNNING`
- Executor owns completion or failure
- Retry attempts are tracked per execution run

8. Failure Handling & Retry Semantics

The system distinguishes between failure types:

- **Validation Failures**
  - Permanent
  - Job is marked as FAILED
  - No retries
- **Transient Failures**
  - Temporary issues (network, downstream services)
  - Automatically retried via Kafka
  - Retry count tracked in database

Dead Letter Queue (DLQ) and max retry policies are **intentionally deferred** to keep the core engine simple and extensible.

9. Logging & Monitoring

- Structured logging using SLF4J and Logback
- Logs written to console and files
- Job-level correlation using job identifiers
- Spring Boot Actuator used for health and basic metrics
- Docker-friendly logging approach

10. Why Low-Level Design (LLD) Is Not Included

Detailed low-level design documents were intentionally avoided.

The system prioritizes:

- Clear service boundaries
- Explicit state transitions
- Well-defined execution semantics

Per-service READMEs provide sufficient implementation detail where required, while this document focuses on architectural clarity and design intent.

Trade-offs & Future Enhancements

Planned improvements include:

- Dead Letter Queue (DLQ) support
- Configurable max retry policies
- Recovery of stuck RUNNING jobs
- Advanced metrics and alerting
- Job pause and cancellation enhancements

The system is designed so these features can be added **without refactoring the core architecture**.