# HRMS + Payroll — Clear Low Level Design (LLD)

Purpose: A concise, implementation-ready LLD for building the HRMS and Payroll modules using **Java 21 + Spring Boot**. Focused, minimal, and actionable for developers.

## 1. Overview

* **Scope:** Employee management, Attendance, Leave, Payroll computation, Payslip generation, Notifications.
* **Architecture:** Modular microservices. Each service own DB. REST for sync, Kafka for async events.
* **Core services:** employee-service, attendance-service, leave-service, payroll-service, notification-service, reporting-service.

## 2. Tech stack

* Java 21, Spring Boot 3.x
* Spring Data JPA, Hibernate
* PostgreSQL (per-service DB)
* Kafka (events)
* Apache Camel (device ingestion / ETL)
* Keycloak (OIDC) for AuthN/AuthZ
* MapStruct for mapping, Flyway for migrations
* Testcontainers, JUnit 5 for tests

## 3. Service responsibilities (short)

* **employee-service:** CRUD employee, department, salary structure reference.
* **attendance-service:** Ingest device data, expose aggregated attendance per period.
* **leave-service:** Manage policies, requests, balances; publish approved leaves.
* **payroll-service:** Resolve salary structure, compute payslips, persist payslips, publish events.
* **notification-service:** Send payslip emails/SMS, retry logic.
* **reporting-service:** Generate PDF/Excel payslips and reports.

## 4. Data model (essential tables)

* **department**(id PK, name)
* **employee**(id PK, emp\_code, first\_name, last\_name, email, dept\_id FK, salary\_structure\_id FK)
* **salary\_structure**(id PK, employee\_id FK, effective\_from, effective\_to)
* **salary\_component\_line**(id PK, structure\_id FK, component\_code, amount, calc\_method, percent\_of)
* **attendance\_aggregate**(id PK, employee\_id FK, period, present\_days, absent\_days, work\_hours)
* **leave\_request**(id PK, employee\_id FK, start\_date, end\_date, days, status)
* **payroll\_run**(id PK, month, year, status)
* **payslip**(id PK, payroll\_run\_id FK, employee\_id FK, gross, deductions, net)
* **payslip\_line**(id PK, payslip\_id FK, code, type, amount)

## 5. Key APIs (payroll-service examples)

* POST /payroll/runs — start payroll run { period: "YYYY-MM" , employeeIds?: [] } → 202 Accepted
* GET /payroll/payslips/{employeeId}/{period} — fetch payslip
* GET /payroll/runs/{id} — run status and summary

Authentication: Bearer JWT (Keycloak). Use scopes payroll.run, payroll.read.

## 6. Payroll run — clear sequence (developer flow)

1. API receives POST /payroll/runs with period (YearMonth).
2. Create payroll\_run record status=RUNNING.
3. For each employee (all or provided list):
   * Resolve effective salary\_structure for period.
   * Fetch attendance\_aggregate and approved leaves for period.
   * Build computation context {basic, hra, allowances, presentDays, leaveDays}.
   * Evaluate components by priority:
     + FLAT → use amount
     + PERCENT → percent of referenced component (resolve value)
     + FORMULA → evaluate expression with safe engine (MVEL/JEXL)
   * Sum earnings, compute deductions, compute tax via TaxService.
   * Persist payslip + payslip\_line entries in a single transaction.
   * Publish payroll.v1.payslip\_generated event with minimal payload.
4. Update payroll\_run status=COMPLETED and publish payroll.v1.run\_completed.

Notes: run employees in parallel batches (worker thread pool). Use DB transactions and idempotency key for retries.

## 7. Component design (classes & responsibilities)

* PayrollController — Accepts run requests, returns immediate 202.
* PayrollOrchestrator — Coordinates batch processing, batching, retries.
* PayrollEngine — Pure calculation logic for one employee + period.
* PayslipService — Persist payslip, payslip lines; handles DB transactions.
* TaxService — Encapsulate tax slab logic (profile-based implementations).
* AttendanceClient, LeaveClient — REST clients to other services (use Feign/WebClient).

## 8. Persistence & transactions

* Use Spring Data JPA repositories.
* Persist payslip and lines in same transaction (@Transactional).
* Payroll run record serves as checkpoint to resume on failure.
* Use unique constraints: (employee\_id, payroll\_run\_id) on payslip to enforce idempotency.

## 9. SQL snippets (critical tables)

CREATE TABLE payroll\_run (  
 id BIGSERIAL PRIMARY KEY,  
 period VARCHAR(7) NOT NULL, -- 'YYYY-MM'  
 status VARCHAR(20) NOT NULL,  
 started\_at TIMESTAMP,  
 completed\_at TIMESTAMP  
);  
  
CREATE TABLE payslip (  
 id BIGSERIAL PRIMARY KEY,  
 payroll\_run\_id BIGINT REFERENCES payroll\_run(id),  
 employee\_id BIGINT NOT NULL,  
 gross DECIMAL(12,2),  
 total\_deductions DECIMAL(12,2),  
 net DECIMAL(12,2),  
 UNIQUE (payroll\_run\_id, employee\_id)  
);  
  
CREATE TABLE payslip\_line (  
 id BIGSERIAL PRIMARY KEY,  
 payslip\_id BIGINT REFERENCES payslip(id),  
 code VARCHAR(50),  
 type VARCHAR(20),  
 amount DECIMAL(12,2)  
);

## 10. Operational & non-functional notes

* **Batching:** Process employees in chunks (e.g., 500) to bound memory usage.
* **Parallelism:** Use a bounded thread-pool to process chunks concurrently.
* **Observability:** Emit metrics for payslips\_generated, errors, time per employee.
* **Retries:** Retry transient failures; on persistent failures mark employee payslip status=FAILED and continue.
* **Backups & audit:** Keep payslip historical records immutable; store audit trail for run operations.

## 11. Testing checklist

* Unit tests for PayrollEngine (edge cases: zero basic, percent loops, formula errors).
* Integration tests using Testcontainers (Postgres + Kafka).
* Contract tests for event schemas.
* Load test a payroll run of 10k employees.

## 12. Next steps (deliverables)

* Provide starter Spring Boot module for payroll-service (skeleton project).
* Flyway migrations for critical tables.
* Example PayrollEngine tests and sample payslip JSON.

*Prepared for: Implementation team — concise and developer-focused LLD.*