**Day 7 – Payroll Feature with Queue-Based Batch Processing**

**What I Accomplished Today**

Today I successfully implemented the payroll management system with a focus on using **Queue data structure** for batch processing of payroll records. This was an important learning experience in applying data structures to solve real-world business problems.

**Database Schema Design**

**Payrolls Table Structure**

I created the payrolls table with comprehensive salary components:

CREATE TABLE payrolls (

id INT PRIMARY KEY AUTO\_INCREMENT,

employee\_id INT NOT NULL,

basic DOUBLE NOT NULL,

hra DOUBLE NOT NULL,

allowances DOUBLE NOT NULL,

deductions DOUBLE NOT NULL,

net\_salary DOUBLE NOT NULL,

pay\_date DATE NOT NULL,

FOREIGN KEY (employee\_id) REFERENCES employees(id)

);

**Why This Design?**

* **Detailed Breakdown**: Separate fields for basic, HRA, allowances, deductions
* **Net Salary**: Calculated final amount for easy reporting
* **Date Tracking**: Pay date for historical records
* **Foreign Key**: Links to employee table for data integrity

**. Payroll Model Class Implementation**

**Core Payroll Class Features**

package com.hrms.model;

import java.time.LocalDate;

public class Payroll {

// Core payroll fields

private int id;

private int employeeId;

private double basic;

private double hra;

private double allowances; // DA or other allowances

private double deductions;

private double netSalary;

private LocalDate payDate;

// Extra fields for JOIN operations

private String employeeName;

private String departmentName;

// Multiple constructors for different use cases...

}

**My Constructor Strategy**

I implemented **three different constructors** to handle various scenarios:

// 1. Full constructor (with JOIN data)

public Payroll(int id, int employeeId, double basic, double hra, double allowances,

double deductions, double netSalary, LocalDate payDate,

String employeeName, String departmentName)

// 2. Constructor for new payroll creation

public Payroll(int employeeId, double basic, double hra, double allowances,

double deductions, double netSalary, LocalDate payDate)

// 3. Constructor for DB reads without JOIN

public Payroll(int id, int employeeId, double basic, double hra, double allowances,

double deductions, double netSalary, LocalDate payDate)

**Why Multiple Constructors?**

* Different scenarios need different data
* JOIN queries return employee/department names
* New records don't have ID yet
* Flexible object creation based on context

**3. Complete CRUD Operations in PayrollDAO**

**CREATE - Adding Payroll Records**

public void addPayroll(Payroll p) {

String sql = "INSERT INTO payrolls (employee\_id, basic, hra, allowances, deductions, net\_salary, pay\_date) " +

"VALUES (?, ?, ?, ?, ?, ?, ?)";

try (Connection conn = DBConnection.getConnection();

PreparedStatement ps = conn.prepareStatement(sql)) {

ps.setInt(1, p.getEmployeeId());

ps.setDouble(2, p.getBasic());

ps.setDouble(3, p.getHra());

ps.setDouble(4, p.getAllowances());

ps.setDouble(5, p.getDeductions());

ps.setDouble(6, p.getNetSalary());

ps.setDate(7, Date.valueOf(p.getPayDate()));

ps.executeUpdate();

System.out.println("✅ Payroll saved for empId=" + p.getEmployeeId());

} catch (SQLException e) {

System.err.println("❌ Error saving payroll: " + e.getMessage());

}

}

**READ - Two Different Approaches**

**Simple Payroll Objects**

public List<Payroll> getAllPayrolls() {

List<Payroll> list = new ArrayList<>();

String sql = "SELECT \* FROM payrolls ORDER BY pay\_date DESC";

// ... implementation returns List<Payroll>

}

**Rich JOIN Query with Details**

public List<String> getAllPayrollDetails() {

List<String> list = new ArrayList<>();

String sql = """

SELECT p.id, e.id AS emp\_id, e.name AS emp\_name, d.name AS dept\_name,

p.basic, p.hra, p.allowances, p.deductions, p.net\_salary, p.pay\_date

FROM payrolls p

JOIN employees e ON p.employee\_id = e.id

JOIN departments d ON e.department\_id = d.id

ORDER BY p.pay\_date DESC

""";

// ... returns formatted strings with full details

}

**Learning Point**: I implemented two different read approaches - one for objects and one for formatted reporting strings.

**UPDATE and DELETE Operations**

public boolean updatePayroll(Payroll p) {

String sql = "UPDATE payrolls SET basic=?, hra=?, allowances=?, deductions=?, net\_salary=?, pay\_date=? WHERE id=?";

// ... returns boolean for success/failure

}

public boolean deletePayroll(int id) {

String sql = "DELETE FROM payrolls WHERE id=?";

// ... returns boolean for success/failure

}

**The Star Feature: Queue-Based Batch Processing**

**Why Use Queue for Payroll Processing?**

In real HR systems, payroll processing often happens in batches (monthly salary processing for all employees). Queue is perfect because:

* **FIFO Processing**: First employee processed first
* **Sequential Handling**: One payroll at a time
* **Memory Efficient**: Process and remove from queue
* **Error Isolation**: If one fails, others continue

**My Queue Implementation**

public void processPayrollQueue(List<Payroll> payrollList) {

if (payrollList.isEmpty()) {

System.out.println("❌ No payrolls to process.");

return;

}

Queue<Payroll> payrollQueue = new LinkedList<>(payrollList);

System.out.println("⏳ Starting queue-based payroll processing...");

while (!payrollQueue.isEmpty()) {

Payroll p = payrollQueue.poll();

try {

addPayroll(p);

System.out.println("✅ Payroll processed for empId=" + p.getEmployeeId());

} catch (Exception e) {

System.err.println("❌ Error processing payroll for empId=" + p.getEmployeeId() + ": " + e.getMessage());

}

}

System.out.println("✅ Queue-based payroll processing completed.");

}

**Key Queue Operations I Used:**

* **new LinkedList<>(payrollList)** - Convert list to queue
* **poll()** - Remove and return the head element
* **isEmpty()** - Check if queue is empty
* **Error Handling** - If one payroll fails, others continue processing

**Helper Methods and Utilities**

**ResultSet to Object Mapping**

private Payroll mapResultSetToPayroll(ResultSet rs) throws SQLException {

return new Payroll(

rs.getInt("id"),

rs.getInt("employee\_id"),

rs.getDouble("basic"),

rs.getDouble("hra"),

rs.getDouble("allowances"),

rs.getDouble("deductions"),

rs.getDouble("net\_salary"),

rs.getDate("pay\_date").toLocalDate()

);

}

**Why This Helper?**

* **Code Reusability**: Used in multiple read methods
* **Consistency**: Same mapping logic everywhere
* **Maintainability**: Change mapping in one place
* **Type Safety**: Proper LocalDate conversion

**What Each Method Accomplishes**

**Core CRUD Methods:**

1. **addPayroll(Payroll p)** - Single payroll insertion
2. **getAllPayrolls()** - Returns Payroll objects for further processing
3. **getAllPayrollDetails()** - Returns formatted strings for reports/display
4. **getPayrollByEmployee(int empId)** - Employee-specific payroll history
5. **updatePayroll(Payroll p)** - Modify existing payroll records
6. **deletePayroll(int id)** - Remove payroll records

**Advanced Features:**

1. **processPayrollQueue(List<Payroll> payrollList)** - **Queue-based batch processing**
2. **mapResultSetToPayroll(ResultSet rs)** - Object mapping utility

**Technical Learning Points**

**Data Structure Application:**

* **Queue Choice**: Perfect for sequential batch processing
* **FIFO Behavior**: Natural for payroll processing order
* **Memory Management**: Process and remove, don't hold all in memory

**Database Design:**

* **Salary Components**: Separate fields for transparency
* **JOIN Queries**: Combining multiple tables for rich reports
* **LocalDate Handling**: Proper date conversion between Java and SQL

**Error Handling Strategy:**

* **Individual Processing**: One failed payroll doesn't stop the batch
* **User Feedback**: Clear success/error messages with emojis
* **Boolean Returns**: For update/delete operations to check success

**Real-World Applications**

**Monthly Payroll Scenario:**

// HR prepares monthly payrolls

List<Payroll> marchPayrolls = prepareMonthlyPayrolls();

// Process all at once using queue

PayrollDAO dao = new PayrollDAO();

dao.processPayrollQueue(marchPayrolls);

// Generate reports

List<String> payrollReports = dao.getAllPayrollDetails();

payrollReports.forEach(System.out::println);

**Why Queue is Better Than Simple Loop:**

* **Structured Approach**: Clear FIFO processing
* **Scalability**: Easy to add priority queues later
* **Monitoring**: Can track queue size and progress
* **Error Recovery**: Failed items can be re-queued

**What I Learned About Data Structures**

**Queue Benefits:**

1. **Natural FIFO**: Perfect for sequential processing
2. **Memory Efficient**: Process and discard
3. **Error Isolation**: One failure doesn't stop the queue
4. **Scalable**: Easy to extend with more complex queue types

**When to Use Queue:**

* ✅ **Batch Processing**: Multiple similar operations
* ✅ **Sequential Tasks**: Order matters
* ✅ **Resource Management**: Control processing flow
* ✅ **Background Jobs**: Queue tasks for later processing

**Implementation Insights:**

* **LinkedList as Queue**: Good performance for add/remove operations
* **Error Handling**: Continue processing even if individual items fail
* **Logging**: Important for tracking batch processing progress

**Integration with HRMS Architecture**

This payroll feature integrates seamlessly with my existing HRMS:

* **Employee Reference**: Uses existing employee IDs
* **Department Integration**: JOIN queries for reporting
* **Consistent Patterns**: Same DAO structure as other features
* **Data Types**: LocalDate consistency across all modules

**Complete Feature Set Now Available:**

* ✅ **Day 5**: Attendance tracking
* ✅ **Day 6**: Leave requests in DynamoDB
* ✅ **Day 7**: Payroll with Queue-based processing

Each feature uses appropriate data structures and storage solutions for their specific requirements.