Simplified Walkthrough of Payment Management System Logic

This walkthrough outlines the logic used to build a **Payment Management System** for tracking incoming and outgoing payments, using **Java**, **Lombok**, and **PostgreSQL**. The system is designed to be simple, secure, and maintainable, focusing on core functionality.

1. Problem Breakdown

The system addresses the following needs:

- Track incoming (e.g., client invoices) and outgoing (e.g., salaries, vendor payments) transactions.
- Categorize payments (e.g., Salary, Vendor, Client) and track statuses (Pending, Processing, Completed, Failed, Cancelled).
- Generate financial reports for specific periods (Monthly, Quarterly).
- Ensure security with user roles (Admin, Finance Manager, Viewer) and audit logs for all actions.

2. System Design

The system uses a **layered architecture** to keep code organized and maintainable:

- **Entities**: Core data models (User, Payment, Category, AuditLog, Report) using **Lombok** for concise code (e.g., @Data, @Builder).
- **Repositories**: Handle database operations (e.g., PaymentRepository, UserRepository) using JDBC for PostgreSQL.
- **Services**: Contain business logic (e.g., PaymentService, ReportService) for processing payments and generating reports.
- **Main Application**: A simple console interface for user interaction.

3. Database Structure

The PostgreSQL schema includes:

- **Users**: Stores user details (ID, username, email, role, etc.) for role-based access.
- **Payments**: Tracks payment details (ID, amount, type, status, category ID, user ID, etc.).
- **Categories**: Stores payment categories (e.g., Salary, Vendor).
- **Audit Logs**: Logs actions (e.g., payment creation, status updates) with timestamps and user IDs.
- **Reports**: Stores report metadata (e.g., period, totals).

Lombok simplifies entity classes with annotations like @Getter, @Setter, and @Builder.

4. Core Workflows

a. Add Payment

1. User enters payment details (amount, type, category, etc.) via the console.
2. PaymentService validates inputs (e.g., positive amount, valid category).
3. PaymentRepository saves the payment to PostgreSQL using a JDBC INSERT query.
4. AuditLogRepository logs the action (PAYMENT_CREATED) with user ID and timestamp.
5. The system displays the saved payment ID.

b. Update Payment Status

1. User provides payment ID and new status (e.g., Completed) via the console.
2. PaymentService retrieves the payment using PaymentRepository (SELECT query).
3. Validates the status change (e.g., Pending to Processing is allowed).
4. Updates the payment in PostgreSQL (UPDATE query).
5. Logs the action (PAYMENT_STATUS_UPDATED) in the audit log.
6. Displays a success message or error if the payment is not found.

c. Generate Report

1. User specifies report type (Financial) and period (Monthly, Quarterly).
2. ReportService queries payments in the date range using PaymentRepository (SELECT query).
3. Calculates totals (incoming, outgoing, net amount).
4. Saves report metadata to the Reports table.

5. Outputs the report as text (or optionally as a PDF using a simple PDF library).
6. Displays the report summary to the user.

5. Security and Auditing

- **Role-Based Access**: UserService checks roles (Admin, Finance Manager, Viewer) before allowing actions (e.g., only Admin and Finance Manager can modify payments).
- **Data Validation**: Ensures valid inputs (e.g., unique email, valid status transitions) in service classes.
- **Audit Logging**: Every action is logged in the AuditLogs table with user ID, action, and timestamp for traceability.

6. Implementation Details

- **Java**: Used for all logic, with JDBC for PostgreSQL connectivity.
- **Lombok**: Reduces boilerplate code for entities (e.g., @Data for getters/setters, @Builder for object creation).
- **PostgreSQL**: Stores all data, with JDBC queries for CRUD operations.
- **Console Interface**: A simple Main class provides a menu for users to add payments, update statuses, or generate reports.

7. Key Features

- **Modularity**: Separated concerns (entities, repositories, services) for easy maintenance.
- **Simplicity**: Focused on core functionality without frameworks like Spring Boot.
- **Extensibility**: New payment types or report formats can be added by updating enums and services.
- **Reliability**: Input validation and error handling ensure robust operation.

Summary

The system was built by:

1. Defining a simple PostgreSQL schema for payments, users, categories, audits, and reports.
2. Using Java with Lombok for clean, concise code.

3. Implementing core workflows (add payment, update status, generate reports) with JDBC for database access.
4. Ensuring security with role-based access and audit logging.
5. Providing a console interface for user interaction.

This approach delivers a lightweight, secure, and functional system tailored for fintech payment management.

2. Link of the github repository :- link
https://github.com/swarnimsrijan/PayementManagementSystem

3. Steps to execute the application

3.1 Clone the Repository

Open a terminal and run:

git clone git@github-personal:swarnimsrijan/PayementManagementSystem.git

cd PayementManagementSystem

3.2 Build the Project

Use Maven to clean and build the project:

mvn clean install

3.3 Run the Application

Start the java application:

mvn exec:java

Or

```
mvn exec:java -Dexec.mainClass="paymentManagementSystem.Main"
```