End-to-End Logic of Payment Management System

repoLink :- https://github.com/aomwankhede/MiniProject-1

This project is a Payment Management System (PMS) built in Java. It follows a layered architecture with clear separation of concerns. The system allows user authentication, role-based authorization, management of parties (clients, vendors, employees), management of payments (client, vendor, salary), and generation of reports.

1. Architecture

- Frontend (CLI)
  Uses simple Java console menus (AuthInterface, UserEntityInterface, PartyEntityInterface, PaymentEntityInterface, RoleEntityInterface, ReportGenerationInterface). It handles user input and delegates work to the service layer.
- Service Layer
  Contains business logic and validation. Each entity or concept has a corresponding service (UserService, RoleService, AuthService, ClientService, VendorService, EmployeeService, ClientPaymentService, VendorPaymentService, SalaryPaymentService, ReportGenerationService).
- Repository Layer
  Responsible for persistence using JDBC. Repositories (UserRepository, RoleRepository, PermissionRepository, ClientRepository, VendorRepository, EmployeeRepository, ClientPaymentRepository, VendorPaymentRepository, SalaryPaymentRepository) map database rows to model classes.
- Database
  A PostgreSQL database stores users, roles, permissions, parties, and payments.
- Models / Enums
  Entities like User, Role, Permission, Client, Vendor, Employee, Payment, ClientPayment, VendorPayment, SalaryPayment.
  Enums: Action, Entity, PaymentDirection, PaymentStatus.

2. Entities and Inheritance

The system uses inheritance to reduce duplication and model hierarchies properly.

1. Party hierarchy
    - Party (abstract) → base class with fields: id, name, bankAccount, contactEmail.
    - Client extends Party (adds company, contractId).
    - Vendor extends Party (adds gstNumber, invoiceTerms).
    - Employee extends Party (adds department, panNumber).

2. Payment hierarchy
    - Payment (abstract) → base class with fields: id, amount, direction, status, createdAt, updatedAt.
    - ClientPayment extends Payment (adds clientId).
    - VendorPayment extends Payment (adds vendorId).
    - SalaryPayment extends Payment (adds employeeId).

3. Role and Permission
    - Role contains a list of Permission.
    - Permission defines allowed Action on an Entity.

3. Authentication and Authorization

- Login / Logout:
  Implemented in AuthService. Login checks credentials from the database and sets the currentUser. Logout clears it.
- Authorization:
  AuthService.hasPermission(Action, Entity) checks if the current user's role contains the required permission.

4. Business Logic Flows

a) User Management

- Create user → UserService.createUser() validates input, hashes password, and saves via UserRepository.
- Assign role → UserService.assignRole(userId, roleId) updates role mapping.
- Find user by username → UserService.findByUsername().

b) Role and Permission Management

- Create role → RoleService.createRole(Role) saves role and links to permissions.
- Assign permissions → RoleService.assignPermissions(roleId, permissionIds).
- Find role by name → RoleService.findByName(name).

c) Party Management

- Create client/vendor/employee through respective services.
- Each service validates fields and then calls the repository.

d) Payment Management

- Client payments must have INCOMING direction.
- Vendor and salary payments must have OUTGOING direction.
- Status can be updated (COMPLETED, FAILED).
- Each payment type has its own service and repository.

e) Report Generation

- Implemented in ReportGenerationService.
- Generates:
    - User report.
    - Role report.
    - Client, Vendor, Employee reports.
    - Payment reports (client, vendor, salary).
    - Overall payment summary (total incoming, total outgoing, completed/failed counts).

Reports are written to files for reference.

5. Persistence Conventions

- Repositories use JDBC with PreparedStatement.
- IDs are generated by the database (auto-increment).
- Enums are stored as strings.
- Timestamps are mapped with LocalDateTime.

6. Testing

- JUnit 5 tests for each service.
- Pattern:
    - @BeforeAll creates a test entity (user, client, vendor, employee).
    - Tests run sequentially to cover CRUD operations and status changes.
    - @AfterAll deletes the test entity so the database returns to its original state.

7. Logging

- SLF4J with Logback.
- Logs are written to console and rotating log files (application.log, error.log).

8. Example Workflow

1. A user registers via CLI.
2. Admin assigns them a role.
3. User logs in.
4. User creates a client (if permissions allow).
5. User records a client payment (INCOMING).
6. Later, payment is marked as COMPLETED.
7. Reports are generated to summarize activities.

9. Key Design Choices

- Inheritance: Party subclasses (Client, Vendor, Employee) and Payment subclasses (ClientPayment, VendorPayment, SalaryPayment).
- Role-based access control: Role → Permissions → (Action + Entity).

- Separation of concerns: frontend → services → repositories → database.
- Tests keep DB clean: temporary data is always deleted after tests.
- Singleton classes for Service Layer Classes