Capstone Project: Payroll Management System

Problem Statement

Managing employee payroll manually is prone to errors, time-consuming, and lacks transparency. Organizations face challenges in maintaining salary records, handling tax deductions, tracking employee leave, and generating accurate salary slips.

A **Payroll Management System** is required to automate payroll operations, provide role-based access (**Admin** and **Employee**), and ensure secure interactions using **JWT Authentication**.

Scope of the System

Admin Role

- **Employee Management** – Add, update, delete, view employee details.
- **Payroll Processing** – Generate monthly salary based on employee details.
- **Leave Management** – Approve/reject employee leave requests.
- **Salary History** – Track payroll history for employees.
- **Departments & Jobs** – Define departments and job roles with base salaries.

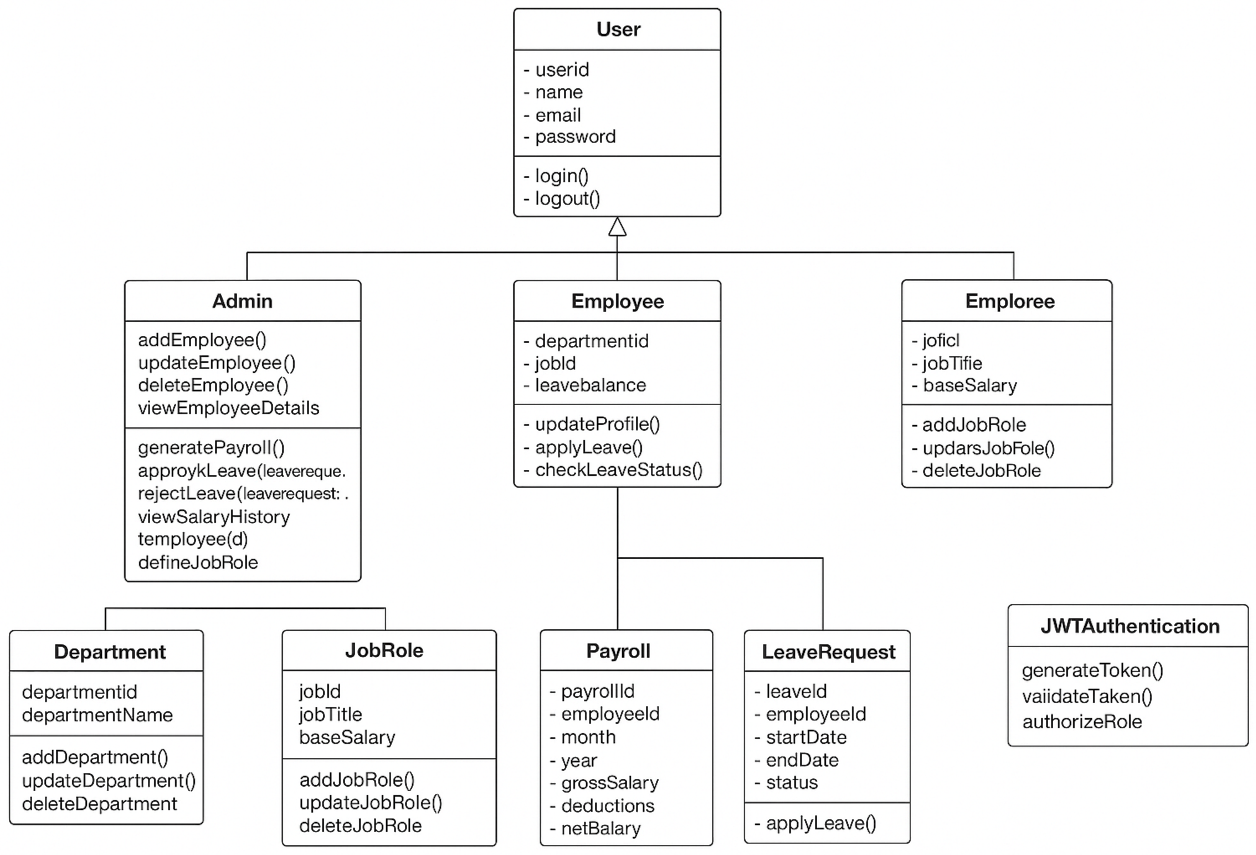
Employee Role

- **Profile Management** – View/update personal details.
- **Leave Requests** – Apply for leave and check leave status.
- **Salary Slip** – View monthly salary slips.

Security

- **JWT Token Authentication** for login and API authorization.
- **Role-based access control** (Admin vs. Employee).

UML Diagram



Project Development Guidelines

Backend (Spring Boot + MySQL)

Technology Stack

- Spring Boot (REST APIs)
- Spring Security with JWT Authentication
- MySQL Database (Employee, Payroll, Leave)
- JPA/Hibernate for ORM

Modules

1. **Authentication Module** – JWT-based login and role assignment (Admin/Employee).
2. **Employee Module** – CRUD operations for employee details.
3. **Payroll Module** – Salary calculation and history.

4. **Leave Module** – Leave request/approval workflow.
5. **Profile Module** – Employee personal data updates.
6. **Reports Module** – Payroll summary and department cost reports.

Frontend (React.js)

Technology Stack

- React.js (Functional Components + Hooks)
- React Router for navigation
- Axios for API calls
- JWT Token storage (localStorage/sessionStorage)
- Bootstrap (for responsive UI components)
- Custom CSS (for branding & extra styling)

Modules

- **Authentication Module** – Login page with JWT handling.
- **Admin Dashboard** – Manage employees, payroll, leave approvals, departments, jobs.
- **Employee Dashboard** – View profile, request leave, view salary slip.
- **Role-Based Routing** – Conditional navigation (Admin vs. Employee).

Frontend Flow

- Login → Store JWT → Decode role → Redirect to dashboard.
- Attach JWT in `Authorization: Bearer <token>` header for API requests.
- Handle 401/403 errors with logout and redirection.

Extended API Guidelines

Base URL: `/api/v1`

APIs secured with JWT.

Swagger UI integrated for interactive API documentation at:

- http://localhost:8080/swagger-ui/

Swagger API Examples

Authentication

POST /api/v1/auth/login
Request:

```
{
  "username": "john.doe",
  "password": "secret123"
}
```

Response:

```
{
  "accessToken": "eyJhbGciOiJIUzI1...",
  "user": {
    "id": 1,
    "username": "john.doe",
    "role": "EMPLOYEE"
  }
}
```

Module	Endpoint	Method	Access	Description
Auth & Users	/auth/login	POST	Public	Authenticates a user and returns a JWT with an access token and user details.
	/users/me	GET	All (Auth)	Retrieves the details of the currently logged-in user.
	/users	POST	Admin	Creates a new user with a specified role (Admin or Employee).
	/users/:id/status	PATCH	Admin	Activates or deactivates a user account.
Employees	/employees	GET	Admin	Lists employees with optional filters.
	/employees	POST	Admin	Creates a new employee record.
	/employees/:id	GET	Admin / Self	Retrieves the profile of a specific employee.
	/employees/:id	PUT	Admin	Updates the details of an employee.
	/employees/:id/ salary-structures	GET	Admin	Retrieves the salary structure(s) of a specific employee.
	/employees/:id/ salary-structures	POST	Admin	Assigns a new salary structure to an employee.
Departments & Jobs	/departments	GET	Admin	Retrieves all departments.

	/departments	POST	Admin	Creates a new department.
	/departments/:id	PUT	Admin	Updates a department's details.
	/departments/:id	DELETE	Admin	Deletes a department.
	/jobs	GET	Admin	Retrieves all job roles.
	/jobs	POST	Admin	Creates a new job role.
	/jobs/:id	PUT	Admin	Updates a job role.
	/jobs/:id	DELETE	Admin	Deletes a job role.
Payroll	/payroll/runs	POST	Admin	Creates a new payroll run for a specified year and month.
	/payroll/runs/:id/process	POST	Admin	Processes a specific payroll run, calculating all salaries.
	/payroll/runs/:id/lock	POST	Admin	Locks a payroll run to prevent further changes.
	/payroll/runs/:id/items	GET	Admin	Retrieves the detailed payroll items for a specific run.
	/payroll/my/:year/:month	GET	Employee	Retrieves an employee's own net pay details for a given period.
Reports	/reports/payroll-summary	GET	Admin	Generates a summary report of payroll for a given period.
	/reports/department-cost	GET	Admin	Generates a report on department costs for a given period.

Database Guidelines (Conceptual)

- **Normalization**: At least 3rd Normal Form (separate users, employees, payroll).
- **User-Employee Mapping**: One-to-one mapping (each employee has a user account).
- **Department & Job Mapping**: One department → Many employees, One job → Many employees.
- **Salary Structure**: Historical tracking with `effective_from` and `effective_to`.
- **Leave Management**: Separate leave types, balances, and requests.
- **Payroll**: Payroll runs are created per month; payroll items store employee-specific salary.

Entities and Relationships

1. User (Table:users)

- `user_id (PK)`
- `username`
- `password`
- `email`
- `role` (Admin / Employee)

2. Employee (Table: employees)

- `employee_id (PK)`
- `user_id (FK → User.user_id)`
- `first_name`
- `last_name`
- `dob`
- `phone`
- `address`
- `designation`
- `department`
- `salary`

3. Payroll (Table: payroll)

- `payroll_id (PK)`
- `employee_id (FK → Employee.employee_id)`
- `basic_salary`
- `deductions`
- `bonus`
- `net_salary`
- `pay_date`

4. Leave (Table: leave)

- `leave_id (PK)`
- `employee_id (FK → Employee.employee_id)`
- `start_date`
- `end_date`
- `leave_type` (e.g., Sick, Casual, Paid)
- `status` (Pending / Approved / Rejected)

UX guidelines table for a Payroll Management System

UX Guideline	UX Principle	Implementation in Payroll Management System
Consistency	Maintain uniformity in design and interactions	Use consistent color schemes, fonts, button styles, table layouts, and form designs across all modules (Employee, Payroll, Leave, Departments).
Clarity & Simplicity	Reduce cognitive load, make tasks understandable	Keep payroll forms simple with clear labels, avoid clutter on dashboards, and highlight important info like salary, leave status, and deductions.
Feedback & Response	Keep users informed about actions	Provide confirmation messages for actions like "Salary processed," "Leave approved," or "Employee added," and show loading indicators during processing.
Error Prevention & Handling	Minimize errors and guide users	Validate inputs on forms (e.g., numeric fields for salary), show clear error messages, and prevent actions that could corrupt data.

Expected Outcomes

By implementing this Payroll Management System:

1. **Efficiency & Accuracy**
    - Automated payroll reduces errors in salary calculation.
    - Attendance and leave directly impact payroll.

2. **Role-Based Access**
    - Admins manage all employees, payroll, and leave approvals.
    - Employees access only their own data (profile, salary, leave).
3. **Transparency**
    - Employees can view salary slips and leave status anytime.
    - Admins can generate payroll and department cost reports.
4. **Security**
    - JWT ensures secure login and API communication.
    - Passwords stored using encryption (BCrypt).
5. **Scalability**
    - Modular design allows future integration with tax systems, biometric attendance, or mobile apps.
6. **Reporting & Analytics**
    - Salary history per employee.
    - Department-wise cost tracking.
    - Leave usage trends.

General Guidelines

1. **Project Structure**
    - Maintain a consistent folder structure for frontend, backend, and documentation.
    - Separate configuration files, source code, and assets for clarity.
2. **Coding Standards**
    - Use meaningful variable, function, and class names.
    - Maintain consistent indentation, spacing, and commenting.
3. **Version Control with Git**
    - **Repository Setup:** Initialize a Git repository at the project root (`git init`).
    - **Branching Strategy:**
        - Use `main` (or `master`) for production-ready code.

        - Create feature branches (`feature/<feature-name>`) for new features.

4. **Documentation**
    - Keep API specs, UI/UX guidelines, and system diagrams updated.
    - Include README with project setup, dependencies, and instructions.
5. **Testing & Validation**
    - Test features before committing.
    - Document test cases and outcomes for major modules.