Capstone Project Report

Payroll Management System

Author: Gudala Lalitha Maheswari

Batch 3 – Java Full stack React

Date: September 4, 2025

Abstract

This document provides a comprehensive overview of the design, development, testing, and implementation of the Payroll Management System, a capstone project aimed at resolving the inefficiencies inherent in manual payroll processing. The project delivers a secure, role-based, full-stack web application that serves as a centralized platform for managing all aspects of employee compensation. The system features a dual-dashboard architecture: a powerful administrative portal for managing the employee lifecycle, processing monthly payroll, and viewing analytics, and a streamlined self-service portal for employees to manage their profiles, submit and track leave requests, and access their payslips.

Developed using a modern technology stack featuring a robust Spring Boot backend and a dynamic React frontend, the application automates complex salary calculations, incorporates business logic for bonuses and leave-based deductions, and ensures data integrity through a comprehensive, multi-layered testing strategy. The final product is a scalable, secure, and efficient solution designed to meet the payroll needs of small to medium-sized businesses, significantly reducing administrative overhead while enhancing transparency and employee satisfaction.

1. Introduction

1.1. Project Overview

The Payroll Management System is a full-stack web application engineered to automate the entire lifecycle of employee payroll. In the modern business environment, particularly for small to medium-sized businesses (SMBs), manual payroll management using spreadsheets or paper-based systems is not only inefficient but also a significant source of operational risk. This project was conceived to address these challenges by providing a digital, centralized platform for all payroll-related activities.

The system acts as a single source of truth for both HR administrators and employees. It empowers administrators with the tools to manage company data and execute complex payroll runs with confidence, while providing employees with direct, transparent access to their compensation and leave information. By automating repetitive tasks and enforcing business rules, the application aims to foster transparency, reduce administrative burden, and allow HR personnel to focus on more strategic initiatives.

1.2. Problem Statement

Managing employee payroll manually is a process fraught with challenges that can negatively impact an organization's financial health and employee morale. The core problems addressed by this project include:

High Risk of Human Error: Manual data entry and calculations are highly susceptible to errors, which can lead to incorrect salary payments, tax miscalculations, and compliance issues.

Significant Time Consumption: The process of calculating salaries, applying deductions, and generating payslips for each employee is incredibly time-consuming, especially as an organization grows.

Lack of Transparency: In a manual system, employees have limited visibility into how their salary is calculated, the status of their leave requests, or their historical payment records. This can lead to mistrust and an increased number of queries to the HR department.

Security and Compliance Risks: Storing sensitive employee and financial data in insecure formats like spreadsheets poses a significant security risk. Furthermore, manual systems make it difficult to ensure compliance with labor and tax regulations.

Poor Scalability: Manual processes do not scale. As a company hires more employees, the administrative burden of manual payroll increases exponentially, becoming a major bottleneck.

This project directly tackles these issues by creating an automated, secure, and transparent system.

1.3. Objectives & Scope

Objectives

The primary objectives of this project were to:

Automate Core Payroll Functions: To design and implement an automated monthly payroll calculation process that accurately incorporates variable factors like bonuses and deductions for unpaid leave.

Implement Strict Role-Based Access: To provide two distinct user experiences—a comprehensive portal for Administrators and a focused self-service portal for Employees—with secure, role-based access control.

Ensure System Security: To secure all API communication using JSON Web Token (JWT) authentication and to store sensitive data, such as passwords, using industry-standard hashing techniques.

Deliver an Intuitive User Experience: To create a professional, modern, and user-friendly interface that simplifies complex tasks and adheres to established UX principles.

Scope

In-Scope Features:

Admin Role: Full Create, Read, Update, Delete (CRUD) management of employees, departments, and job roles; a workflow-driven payroll processing system (Create, Process, Lock); management and approval/rejection of leave requests; and an analytics dashboard for reporting on department costs and leave trends.

Employee Role: A self-service portal for viewing and updating personal profile information (address, phone), applying for leave and tracking its status in real-time, and viewing detailed historical payslips.

Out-of-Scope Features:

Third-Party Tax Integration: Direct integration with government tax filing systems was deemed out of scope due to its complexity and focus on a specific geographic region.

Biometric Attendance: Integration with biometric hardware for attendance tracking was excluded to keep the project focused on core payroll calculation.

Mobile Application: A dedicated mobile application was considered a future enhancement, with the current focus being on a comprehensive desktop web experience.

1.4. Target Audience

The primary target audience for this system is Small to Medium-sized Businesses (SMBs). These organizations often have a small HR team and are at a stage where manual payroll

systems become unmanageable but a full-scale enterprise resource planning (ERP) system is too costly and complex. The system is designed for two main user personas:

The HR Administrator: This user needs a powerful, efficient tool to manage the entire employee lifecycle and process payroll accurately and on time.

The Employee: This user needs a simple, transparent, and accessible way to view their salary information and manage their leave requests without needing to contact HR for every query.

2. Requirement Analysis

2.1. Functional Requirements

The system's capabilities are defined by a set of functional requirements, categorized by user role and system-wide security.

Administrator Role:

Full Management: Admins shall be able to perform complete CRUD (Create, Read, Update, Delete) operations on employee records, organizational departments, and job roles.

Compensation: They shall be able to assign and view the historical salary structures for any employee.

Payroll Lifecycle: Admins are responsible for the entire payroll workflow, including creating a new run for a specific month, processing it to calculate all salaries, and locking the run to finalize it.

Leave Approval: They shall be able to view all employee leave requests, filter them by status, and approve or reject pending requests.

Reporting: Admins have access to reports on department-wise salary costs and trends in leave usage.

Employee Role:

Self-Service: Employees shall be able to view and update their own personal profile information.

Leave Management: They are able to apply for leave and view the real-time status of all their requests.

Payslip Access: Employees shall be able to view their detailed monthly payslips, but only after an administrator has locked the corresponding payroll run.

Security:

Authentication: The system shall provide a login endpoint for all users to authenticate with a username and password.

Authorization: The system shall issue a JSON Web Token (JWT) upon successful login and require a valid JWT to protect all subsequent API requests.

2.2. Non-Functional Requirements

Performance: The application is designed to be highly responsive. Standard API GET requests for data retrieval must have an average response time of under 500 milliseconds to ensure a fluid user experience. The frontend application load time should be under 3 seconds on a standard broadband connection.

Security: Security is a cornerstone of the system. All user passwords must be securely hashed using the industry-standard BCrypt algorithm before being persisted to the database. The system must prevent common web vulnerabilities such as Cross-Site Scripting (XSS) and SQL Injection.

Usability: The user interface must be intuitive and easy to navigate for non-technical users. It must adhere to the UX principles defined in the project scope, including:

Consistency: Uniform design elements across the application.

Clarity & Simplicity: A clean, uncluttered interface that prioritizes important information.

Feedback & Response: Clear confirmation messages, loading indicators, and toast notifications for all user actions.

Error Prevention: Robust client-side and server-side validation to guide users and prevent invalid data submission.

Scalability: The backend is built on a modular Spring Boot architecture, and the database is normalized, allowing the system to handle a growing number of employees and historical data without significant performance degradation.

2.3. Technology Stack and Justification

Tier	Technology	Justification
Backend	Java 21 & Spring Boot 3.x	Chosen for its robustness, performance, and extensive ecosystem. Spring Boot accelerates development, while Spring Security provides a powerful framework for implementing enterprise-grade security.
Database	MySQL 8.x	A reliable, open-source relational database that is well-supported and ideal for structured data like employee and payroll records.
ORM	Spring Data JPA / Hibernate	Simplifies database interactions, reduces boilerplate JDBC code, and provides a layer of database independence.
Frontend	React 18 (with Vite)	A modern, component-based JavaScript library for building fast and complex single-page applications. Vite provides an extremely fast development server and optimized build process.
UI Framework	Bootstrap 5	A comprehensive and responsive UI toolkit that provides a solid foundation for building professional-looking components quickly, ensuring a consistent design.
Form Mgmt	Formik & Yup	A powerful combination for managing form state and validation in React, reducing complexity and improving user experience through real-time feedback.

3. Planning

3.1. Methodology

An Agile-inspired, modular development approach was adopted for this project. The work was systematically broken down into distinct backend and frontend phases. Each phase was further subdivided into feature-based modules (e.g., Authentication, Employee Management, Payroll Processing). This modular strategy acted as a series of mini-sprints, allowing for an iterative process where each component was designed, developed, and tested before being integrated into the larger system. This approach ensured a high-quality, robust final product and allowed for flexibility in development.

3.2. Project Timeline

The project was executed over a condensed and highly focused 3-day sprint. Each day was dedicated to a specific, high-level phase of the development lifecycle, ensuring a clear focus and measurable progress from initial design to final delivery.

Day 1:

Frontend and Backend Core:

This day was dedicated to project planning and building the entire backend infrastructure. Key deliverables included finalizing the 8-table database schema, defining all 35+ API endpoints, scaffolding the Spring Boot project, implementing the schema via Flyway, developing all JPA entities, and implementing the core JWT security layer. On the frontend, the initial project setup and folder structure were created.

Day 2:

Full-Stack Feature Implementation

This day focused on implementing the core business logic and building the complete user interface. On the backend, this involved developing the complex service logic for the Payroll (salary calculation) and Leave (balance checks) modules, along with all other CRUD services. On the frontend, the Vite React project was fully set up, and both the Admin and Employee Dashboards were developed and integrated with the backend APIs.

Day 3:

Testing, Validation and Finalization

The final day was dedicated to ensuring quality and preparing the project for handover. The backend was thoroughly tested with a comprehensive suite of Unit Tests (Mockito) and Integration Tests (MockMvc). The frontend was validated with client-side form validation (Formik & Yup) and manual end-to-end testing of the primary user flows. The project was prepared for local execution, and all documentation, including the final README.md, was completed.

4. System Design

4.1. System Architecture

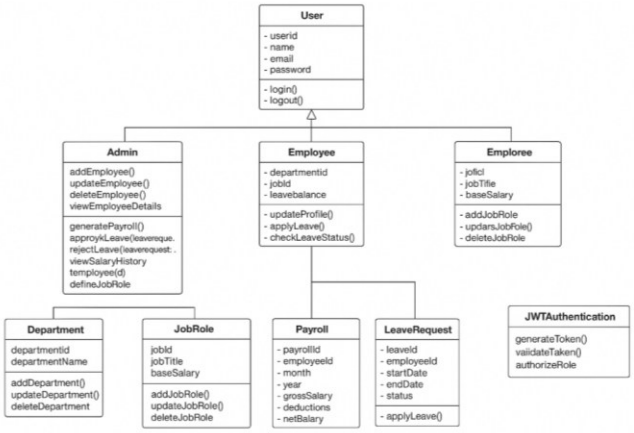
The system employs a modern, decoupled client-server architecture. This architectural pattern separates the user interface (client) from the business logic and data storage (server), allowing for independent development, deployment, and scaling of each part.

Client (Frontend): A dynamic Single-Page Application (SPA) built with React. It is responsible for rendering the user interface and managing all user interactions. It runs entirely in the user's web browser and communicates with the backend via REST API calls.

Server (Backend): A powerful RESTful API built with Spring Boot. It is responsible for all business logic, data processing, and security enforcement. It acts as an intermediary between the client and the database.

Database: A MySQL relational database that serves as the persistent storage layer for all application data.

UML:



This design ensures a clean separation of concerns, making the system more maintainable and flexible for future enhancements.

4.2. Backend Design

API Design

The backend exposes a versioned REST API, with the base URL /api/v1. The API adheres to standard REST principles, utilizing appropriate HTTP methods (GET, POST, PUT, PATCH, DELETE) for different operations and returning standard HTTP status codes to indicate the outcome of a request. The API is stateless; every request from the client must contain all the information needed to be understood, which is achieved by including a JWT in the Authorization header.

For developer convenience and clear documentation, the API is integrated with Swagger UI, providing an interactive interface to explore and test all available endpoints.

Example API Interaction (Create Employee):

Request: POST /api/v1/employees

Headers:

Authorization: Bearer <jwt_token>

Body:

{

“username:”new.employee”,

“password”:”strongPassword123”,

“email”:”new@example.com”,

“firstName”:”New”,

“lastName”:”Employee”

}

Success Response: 201 Created with the newly created employee object.

Error Response: 400 Bad Request if validation fails, or 403 Forbidden if the user is not an Admin.

Database Schema

The database is designed in 3rd Normal Form to ensure data integrity, eliminate redundancy, and improve scalability. It consists of the following 8 tables.

Table Definitions:

users: Stores authentication credentials and roles.

id (PK), username (UNIQUE), email (UNIQUE), password (Hashed), role.

departments: Stores organizational departments.

id (PK), name (UNIQUE).

job_roles: Stores job titles and their default base salaries.

id (PK), title (UNIQUE), base_salary.

employees: Stores detailed employee information.

id (PK), user_id (FK, UNIQUE), department_id (FK), job_role_id (FK), first_name, last_name, date_of_birth, hire_date, leave_balance. This table has a strict one-to-one relationship with users.

salary_structures: A key table that provides historical tracking of an employee's compensation.

id (PK), employee_id (FK), base_salary, bonus_details (JSON), effective_from (DATE), effective_to (DATE). This allows for scheduling future salary changes and maintaining a perfect audit trail.

leave_requests: Tracks all employee leave applications.

id (PK), employee_id (FK), leave_type, start_date, end_date, status, reason.

payroll_runs: Stores a record for each monthly payroll cycle.

id (PK), year, month, status (DRAFT, PROCESSED, LOCKED), processed_at, locked_at. This table is the master record for each payroll event.

payroll_items: Stores the detailed, calculated salary slip for each employee within a specific run.

id (PK), run_id (FK), employee_id (FK), base_salary, bonus, deductions, net_salary, pay_date.

4.3. Frontend Design

Component Architecture

The React application is structured logically to promote reusability, maintainability, and a clear separation of concerns.

pages/: Contains top-level components that represent a full page or view (e.g., EmployeeManagementPage.jsx). These components are responsible for fetching data and composing the page layout from smaller components.

components/: Contains reusable UI pieces. This is further divided into:

common/: Generic, application-wide components like custom buttons, modals, and loaders.

admin/ & employee/: More complex components specific to a single user role's dashboard.

layouts/: Contains wrapper components (AdminLayout.jsx, EmployeeLayout.jsx) that provide the consistent sidebar and header structure for different sections of the application.

context/: Manages global state, with AuthContext.jsx being the most critical for handling user authentication.

routes/: Defines the application's navigation structure and protected routes.

Component Tree :

frontend/
│── node_modules/
│── public/
│── src/
│ │── api/
│ │ ├── apiService.js

```
│   │   └── index.js
│   │
│   ├── assets/
│   │
│   ├── components/
│   │   ├── admin/
│   │   │   └── EmployeeDetailTabs/
│   │   │       ├── LeaveHistoryTab.jsx
│   │   │       ├── PayslipsTab.jsx
│   │   │       ├── ProfileTab.jsx
│   │   │       └── SalaryStructureTab.jsx
│   │   │
│   │   ├── styles/
│   │   ├── AddEmployeeModal.jsx
│   │   └── EmployeeDetailPane.jsx
│   │
│   │   ├── common/
│   │   │   ├── styles/
│   │   │   ├── ConfirmationModal.jsx
│   │   │   ├── LoadingSpinner.jsx
│   │   │   ├── SkeletonLoader.jsx
│   │   │   ├── ToastNotification.jsx
│   │   │   └── index.js
│   │   │
│   ├── context/
│   │   ├── AuthContext.jsx
│   │   └── index.js
│   │
│   ├── layouts/
│   │   ├── styles/
│   │   ├── AdminLayout.jsx
│   │   └── EmployeeLayout.jsx
│   │
│   ├── pages/
```

```
│   │   ├── admin/
│   │   │   ├── styles/
│   │   │   ├── AdminDashboard.jsx
│   │   │   ├── EmployeeManagementPage.jsx
│   │   │   ├── LeaveManagementAdminPage.jsx
│   │   │   ├── OrgSettingsPage.jsx
│   │   │   ├── PayrollManagementPage.jsx
│   │   │   └── ReportsPage.jsx
│   │   │
│   │   ├── employee/
│   │   │   ├── styles/
│   │   │   ├── EmployeeDashboard.jsx
│   │   │   ├── LeaveRequestPage.jsx
│   │   │   ├── PayslipsPage.jsx
│   │   │   ├── ProfilePage.jsx
│   │   │   └── SalaryStructurePage.jsx
│   │   │
│   │   ├── styles/
│   │   └── LoginPage.jsx
│   │
│   ├── routes/
│   │   ├── AppRoutes.jsx
│   │   ├── index.js
│   │   └── ProtectedRoute.jsx
│   │
│   ├── validation/
│   │   └── employeeSchema.js
│   │
│   ├── App.css
│   ├── App.jsx
│   ├── index.css
│   ├── main.jsx
│
├── eslint.config.js
```

```
├── index.html
├── package-lock.json
├── package.json
├── README.md
└──vite.config.js
```

State Management

Global application state, specifically the user's authentication status (JWT token, role, and username), is managed using React's Context API. An AuthContext is created to provide this data to any component in the application that needs it. This avoids "prop drilling" and provides a clean, centralized way to handle login, logout, and user session persistence. For component-level state, standard React Hooks (useState, useEffect) are used.

UI/UX Design

The user interface and experience are built on a "Muted & Professional" design philosophy, aimed at creating an application that is both aesthetically pleasing and highly functional.

Color Palette: A professional grayscale palette is used for the core structure, providing a clean and calming foundation. This is paired with a sophisticated, muted Indigo (#4C51BF) as the primary action color for all interactive elements like buttons and links, guiding the user's attention to key actions.

Typography: The "Poppins" font family is used throughout the application for its excellent readability and modern aesthetic. A clear typographic hierarchy is maintained for headings, subheadings, and body text.

Rich UX Patterns: The application employs advanced UX patterns to enhance efficiency and usability:

Master-Detail Pane View: For employee management, clicking an employee in the main table slides out a detail pane from the right. This allows the administrator to view and edit employee details without losing the context of the main list.

Inline Editing: For managing simple lists like Departments and Jobs, administrators can add and edit items directly within the table, making the process incredibly fast and fluid.

Animations & Feedback: Subtle CSS animations are used for page transitions and modal pop-ups to create a smooth experience. Skeleton Loaders are used when fetching data to give the user a visual indication that content is loading, which is a significant improvement over traditional spinners. All actions are confirmed with Toast Notifications.

5. Implementation

This section details the implementation of the most complex and critical modules of the system, highlighting key code structures and algorithms.

5.1. Backend Implementation

Security with Spring Security and JWT

The security layer is the foundation of the backend. It was implemented using Spring Security to protect all endpoints and manage authentication and authorization.

JWT Generation: Upon successful login via the AuthController, a JwtTokenProvider service generates a signed JWT. This token contains essential claims, including the username (sub) and the user's role (roles).

JWT Filter Chain: A custom JwtAuthenticationFilter is configured to execute for every incoming request. This filter extracts the JWT from the Authorization: Bearer <token> header, validates its signature and expiration, and if valid, sets the user's authentication context in the SecurityContextHolder.

Role-Based Access Control (RBAC): Access to controller methods is restricted at the method level using the @PreAuthorize annotation. This provides granular control, ensuring that only users with the appropriate role (e.g., hasRole('ADMIN')) can access specific endpoints.

Securing a Controller Method

```
@RestController
@RequestMapping("/api/v1/employees")
public class EmployeeController{

@GetMapping

@PreAuthorrize(“hasRole(‘ADMIN’)”)

Public ResponseEntity<List<EmployeeResponseDto>> getAllEmployees() {

//implementation

}

}
```

Payroll Processing Logic

The core business logic resides in the PayrollServiceImpl. The processPayrollRun method implements a state machine and a complex calculation algorithm.

Algorithm Pseudo-code:

function processPayrollRun(runId):

1. Find payrollRun by runId. If not found, throw error.
2. If run.status is ‘LOCKED’ throw error.
3. If run.status is ‘PROCESSED’, delete all existing payrollItems for this run.
4. Get all active employees.
5. For each employee:
    a. Find their active salaryStructure for the run’s month.
    b. If no structure found, log warning and continue to next employee.
    c. Calculate bonus based on bonus_details in the structure.
    d. Find all approved, unpaid leave requests within the month.
    e. Calculate lossOfPayDeduction = (baseSalary / daysInMonth) * unpaidLeaveDays.
    f. netSalary = (baseSalary + bonus) – lossOfPayDeduction.
    g. Create and save a new payrollItem with all calculated values.
6. Update payrollRun status to ‘PROCESSED’ and set processed at timestamp.
7. Save and update payRun

This logic is executed within a single database transaction (@Transactional) to ensure data consistency.

5.2. Frontend Implementation

Role-Based Routing

To enforce a strict separation between the Admin and Employee dashboards, a custom ProtectedRoute component was created. This component acts as a gatekeeper for sensitive routes.

Example: **ProtectedRoute.jsx Usage in AppRoutes.jsx**

```
Import { Routes, Route } from 'react-router-dom';
import ProtectedRoute from './ProtectedRoute';
import AdminLayout from '../layouts/AdminLayout';
import EmployeeManagementPage from '../pages/admin/EmployeeManagementPage';
```

```
import LoginPage from '../pages/LoginPage';

const AppRoutes = () => {
  return (
    <Routes>
      <Route path="/login" element={<LoginPage />} />

      {/* Admin Routes */}
      <Route
        path="/admin/employees"
        element={
          <ProtectedRoute role="ADMIN">
            <AdminLayout>
              <EmployeeManagementPage />
            </Layout>
          </ProtectedRoute>
        }
      />
      {/* ... other admin and employee routes */}
    </Routes>
  );
};
```

This structure ensures that only an authenticated user with the 'ADMIN' role can ever access the EmployeeManagementPage.

Form Management & Validation with Formik and Yup

All forms in the application were built using Formik for state management and Yup for schema-based validation. This pattern significantly cleans up form logic and improves the user experience by providing instant, inline feedback.

Example: Yup Validation Schema (leaveSchema.js)

```
import * as Yup from 'yup';

export const leaveSchema = Yup.object().shape({

  leaveType: Yup.string()
```

```
    .required('Leave type is required'),
  startDate: Yup.date()
    .required('Start date is required'),
  endDate: Yup.date()
    .required('End date is required')
    .min(Yup.ref('startDate'), 'End date cannot be before the start date'),
  reason: Yup.string()
    .required('A reason for the leave is required')
    .min(10, 'Reason must be at least 10 characters long'),
});
```

Example: Using the Schema in a Formik Form

```
import { Formik, Form, Field, ErrorMessage } from 'formik';
import { leaveSchema } from '../../validation/leaveSchema';
const LeaveForm = ({ handleSubmit }) => (
  <Formik
    initialValues={{ leaveType: '', startDate: '', endDate: '', reason: '' }}
    validationSchema={leaveSchema}
    onSubmit={handleSubmit}
  >
    {({ isSubmitting }) => (
      <Form>
        {/* ... form fields with Field and ErrorMessage components ... */}
        <button type="submit" disabled={isSubmitting}>
          {isSubmitting ? 'Submitting...' : 'Submit Request'}
        </button>
      </Form>
    )}
```

```
  </Formik>
);
```

This ensures that the handleSubmit function is only called when the form data is valid according to the schema, fulfilling the "Error Prevention" UX principle.

6. Testing & Validation

A multi-layered testing strategy was implemented to ensure the quality, correctness, and reliability of the backend application.

6.1. Backend Testing

Unit Testing with Mockito

The service layer, containing the most critical business logic, was extensively tested using Mockito. The goal of unit testing is to test a single class in complete isolation from its dependencies (like the database). By mocking the repository layer, we could create controlled, predictable scenarios to validate our logic.

Key Unit Test Scenarios:

Service	Test Case	Purpose
PayrollService	processPayrollRun_withBonus	Verifies that a percentage-based bonus is correctly calculated and added to the net salary.
PayrollService	processPayrollRun_withUnpaidLeave	Verifies that loss-of-pay deductions are correctly calculated for approved, unpaid leave.

PayrollService	processPayrollRun_whenRunIsLocked	Verifies that a BadRequestException is thrown when attempting to process a locked payroll run.
LeaveRequestService	updateLeaveStatus_deductsPaidLeave	Verifies that approving a 'PAID' leave request correctly deducts the days from the employee's leaveBalance.
LeaveRequestService	applyForLeave_insufficientBalance	Verifies that a BadRequestException is thrown when an employee applies for more 'PAID' leave than they have available.

Example: Mockito Test Method

```
@Test
void testUpdateLeaveStatus_deductsPaidLeave_whenLeaveIsPaid() {
    // Given
    Employee employee = new Employee();
    employee.setLeaveBalance(10);
    LeaveRequest leaveRequest = new LeaveRequest();
    leaveRequest.setLeaveType("PAID");
    leaveRequest.setDays(5); // Assuming days are pre-calculated
    leaveRequest.setEmployee(employee);
    when(leaveRequestRepository.findById(any())).thenReturn(Optional.of(leaveRequest));
```

```
    // When
    leaveRequestService.updateLeaveStatus(1L, "APPROVED");
    // Then
    ArgumentCaptor<Employee> employeeCaptor =
ArgumentCaptor.forClass(Employee.class);
    verify(employeeRepository).save(employeeCaptor.capture());
    assertThat(employeeCaptor.getValue().getLeaveBalance()).isEqualTo(5);
}
```

Integration Testing with MockMvc

To test the full HTTP request-response cycle, including the controller and security layers, Integration Tests were written using Spring's MockMvc. These tests simulate real API calls and verify that our security rules and endpoint configurations are working correctly.

Key Integration Test Scenarios:

Authentication: Test the /api/v1/auth/login endpoint for both successful and failed login attempts.

Authorization: Test that a user with an 'EMPLOYEE' role receives a 403 Forbidden error when trying to access an admin-only endpoint like GET /api/v1/employees.

Validation: Test that sending a request with invalid data (e.g., a blank department name) to a POST endpoint correctly returns a 400 Bad Request status with validation error messages.

Example: MockMvc Security Test Method

```
@Test
@WithMockUser(roles = "EMPLOYEE")
void testGetAllEmployees_failsWith403_forEmployeeRole() throws Exception {
    // When & Then
    mockMvc.perform(get("/api/v1/employees")
            .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isForbidden());
}
```

6.2. Frontend Validation

Client-side validation is a crucial part of the "Error Prevention" UX principle. All user input forms in the React application were implemented with Yup for schema-based validation. This provides immediate, user-friendly feedback on invalid inputs (e.g., an incorrectly formatted email, a required field left blank) and ensures that the data sent to the backend API is well-formed and syntactically correct, reducing unnecessary server-side errors.

7. Local Development & Setup

7.1. Environment Setup

The Payroll Management System is designed to run in a standard local development environment. This requires the manual setup of the core components: the database, the backend server, and the frontend server. The following software is required:

Java Development Kit (JDK) 21

Apache Maven 3.6

MySQL Server 8.x

A database management tool (MySQL Workbench)

Node.js 18.x

7.2. Database Setup

The application requires a local instance of a MySQL server.

Create Schema: Using a database management tool, connect to your local MySQL instance and create a new database schema. It is recommended to name it payroll_db.
CREATE DATABASE payroll_db;

Schema Initialization: The database tables and initial data are managed by Flyway. No manual table creation is needed. The first time the backend application is run, Flyway will automatically execute the SQL migration scripts located in src/main/resources/db/migration to create all the necessary tables and seed an initial admin user.

7.3. Backend Setup

The backend is a standalone Spring Boot application.

Configuration: Navigate to the src/main/resources/application.properties file. Update the database details to match your local MySQL setup:

spring.datasource.url=jdbc:mysql://localhost:3306/payroll_db
spring.datasource.username=your_mysql_username
spring.datasource.password=your_mysql_password
You must also set a secret key for JWT signing:

app.jwt-secret=YourVeryStrongAndLongSecretKeyHere

Execution: From an IDE: Open the project in a Java IDE like IntelliJ IDEA or Eclipse and run the PayrollManagementSystemApplication main class.

Access: The backend server will start on http://localhost:8080. The Swagger UI for API documentation will be available at http://localhost:8080/swagger-ui/index.html.

7.4. Frontend Setup

The frontend is a standalone React application built with Vite.

Dependencies: Navigate to the frontend project's root directory and install all necessary dependencies by running npm install.

API Configuration: The frontend is pre-configured to send all API requests to http://localhost:8080. This is configured in the Axios instance in src/api/apiService.js and can be changed if needed.

Execution: Start the local development server by running the following command: npm run dev

Access: The frontend application will become accessible in your web browser at http://localhost:5173 (or the next available port, which will be indicated in the terminal).

8. Conclusion

8.1. Project Summary

This project successfully delivered a complete and functional Payroll Management System. It met all its core objectives by automating payroll, providing secure role-based access, and offering a transparent, user-friendly interface. The final application is a robust and scalable solution that effectively solves the challenges of manual payroll processing.

8.2. Future Scope & Enhancements

The modular design of the system provides a strong foundation for several future enhancements that could further increase its value:

Email Notifications: A key enhancement would be to integrate an email service (like Spring Mail). This would enable automated notifications to be sent to employees when their leave requests are approved or rejected, and when a new payslip has been generated and is available for viewing. This would improve communication and reduce the need for employees to manually check for updates.

Advanced Reporting & Data Export: The current reporting module could be expanded to offer more advanced analytics. This could include customizable charts, date range filtering for all reports, and the ability for administrators to export report data (e.g., payroll summaries, department costs) to standard formats like CSV or PDF for offline analysis and record-keeping.

Third-Party Tax Integration: To make the system more powerful, it could be integrated with a third-party API for automated tax calculation. This would ensure that tax deductions are always up-to-date with the latest government regulations, significantly reducing the compliance burden on the HR department.

8.3. Lessons Learned

The development of this full-stack application was a valuable and comprehensive learning experience. Several key lessons were learned throughout the project lifecycle:

The Importance of Upfront Design: The initial phase of thoroughly designing the database schema and defining the complete API contract proved to be invaluable. This blueprint prevented significant rework later in the development process and ensured that both the frontend and backend teams were building towards a common, well-understood goal.

Challenges of State Management: Managing state in a complex single-page application like the React frontend highlighted the importance of a centralized state management strategy.

Using React's Context API for global authentication state proved to be an effective solution for preventing prop-drilling and keeping the application's state predictable.

The Power of a Multi-Layered Testing Strategy: Implementing both unit tests with Mockito and integration tests with MockMvc was critical for ensuring the backend's reliability. Unit tests provided confidence in the correctness of the core business logic (like salary calculation), while integration tests ensured that the security and API layers were functioning as a cohesive whole. This multi-layered approach was essential for building a robust and trustworthy system.