Project Overview

You are assisting with a Spring Boot web application using Thymeleaf for Employee Management System (EMS). This system manages employee profile, attendence, leave request, and admin module for employee attendence and leave management.

Technology Stack

- **Backend**: Spring Boot 3.2+, Spring MVC, Spring Data JPA
- **View Layer**: Thyemelaf, Bootstrap 5
- **Database**: H2 In Memory Database
- **Testing**: JUnit 5, Mockito, TestContainers
- **Build Tool**: Gradle-8.14
- **Logging**: SLF4J with Log4j2
- **Java** : 17
- **Hibernate**

Architecture Patterns & Code Examples

1. Layered Architecture Pattern

```
// Controller Layer

@Controller

@RequestMapping("/employees")

public class EmployeeController {



   @Autowired

   private EmployeeService employeeService;



   @GetMapping("/{id}")

   public String getEmployee(@PathVariable Long id, Model model) {

      // Implementation follows MVC pattern
   }
}


// Service Layer (Business Logic)
@Service
@Transactional
public class EmployeeServiceImpl implements EmployeeService {


   @Autowired
   private EmployeeRepository employeeRepository;


   // Business logic implementation
}


// Repository Layer (Data Access)
@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long> {
   // Custom query methods
}
```

2. Dependency Injection Pattern

```
@Component
public class EmployeeValidator {
    @Autowired
    private ValidationRuleService validationRuleService;


    @Autowired
    private AuditService auditService;


    // Constructor injection preferred
    public EmployeeValidator(ValidationRuleService validationRuleService,
                            AuditService auditService) {
        this.validationRuleService = validationRuleService;
        this.auditService = auditService;
    }
}
```

3. Repository Pattern with Custom Queries

```
@Repository
public interface EmployeetRepository extends JpaRepository<Employee, Long> {


    @Query("SELECT e FROM Employee e WHERE e.email = :email")
    List<Employee> findEmployeeByEmail(@Param("email") String email);



    @Modifying

    @Query("UPDATE Employee e SET e.name = :name WHERE e.id = :id")

    int updateEmployeeName(@Param("id") Long id, @Param("name") String name);

}
```

Error Handling Patterns

1. Global Exception Handler

```
@ControllerAdvice

public class GlobalExceptionHandler {



    private static final Logger logger =
LoggerFactory.getLogger(GlobalExceptionHandler.class);



    @ExceptionHandler(EmployeeNotFoundException.class)

    public ModelAndView handleProductNotFound(EmployeeNotFoundException ex,
HttpServletRequest request) {

        logger.warn("Employeet not found: {} for request: {}", ex.getMessage(),
request.getRequestURL());



        ModelAndView mav = new ModelAndView("error/employee-not-found");

        mav.addObject("errorMessage", ex.getMessage());

        mav.addObject("timestamp", LocalDateTime.now());
        return mav;
    }



    @ExceptionHandler(ValidationException.class)
    public ModelAndView handleValidationError(ValidationException ex) {
        logger.error("Validation error: {}", ex.getMessage());



        ModelAndView mav = new ModelAndView("error/validation-error");
        mav.addObject("errors", ex.getErrors());
        return mav;
    }
}
```

2. Custom Exception Classes

```
@ResponseStatus(HttpStatus.NOT_FOUND)
public class EmployeeNotFoundException extends RuntimeException {
    public EmployeeNotFoundException(Long employeetId) {
        super("Employee not found with ID: " + employeetId);
    }
}



@ResponseStatus(HttpStatus.BAD_REQUEST)
public class ValidationException extends RuntimeException {
    private final List<String> errors;



    public ValidationException(List<String> errors) {

        super("Validation failed");

        this.errors = errors;

    }



    public List<String> getErrors() {

        return errors;

    }

}
```

Logging Patterns

1. Service Layer Logging

```
@Service

public class EmployeeService {



    private static final Logger logger = LoggerFactory.getLogger(EmployeeService.class);

```

```
@Transactional

public Employee updateEmployeeName(Long employeeId, String employeeName) {

  logger.info("Starting update employee name : employeeId={}, name={}",

        employeeId, employeeName);



  try {

    Employee employee = employeeRepository.findById(employeeId)

      .orElseThrow(() -> new EmployeeNotFoundException(employeeId));



    logger.debug("Current employee name: {}", employee.getName());



    // Business logic here

    employee.setName(employeeName);

    employee.setLastModified(LocalDateTime.now());



    Product savedEmployee = employeeRepository.save(employee);



    logger.info("Successfullyupdated employee name: employeeId={}, name={}",

          employeeId, employeeName);



    return savedEmployee;
```

```
        } catch (Exception ex) {
            logger.error("Failed to update employee name: employeeId={}, name={}, error={}",
                        employeeId, name, ex.getMessage(), ex);
            throw ex;
        }
    }
}
```

2. Audit Logging Pattern

```
@Component
public class AuditLogger {


    private static final Logger auditLogger = LoggerFactory.getLogger("AUDIT");


    public void logEmployeeChange(String username, String action, Long employeeId, String
details) {
        auditLogger.info("USER={} ACTION={} EMPLOYEE_ID={} DETAILS={} TIMESTAMP={}",
                        username, action, productId, details, LocalDateTime.now());
    }
}
```

Coding Standards

Naming Conventions

- **Classes**: PascalCase (EmployeeService, AdminService)
- **Methods**: camelCase (getEmployee, updateEmployeeName)
- **Variables**: camelCase (employeeList, employeeName)
- **Constants**: UPPER_SNAKE_CASE (MAX_EMPLOYEE_NAME_LENGTH)
- **Thyemeleaf Files**: kebab-case (employee-details.html, employee-profile.html)

Documentation Requirements

- **JavaDoc**: All public methods and classes
- **Comments**: Complex business logic and algorithms
- **README**: Setup instructions and API documentation

Validation Standards

- **Bean Validation**: Use @Valid, @NotNull, @Size annotations
- **Custom Validators**: For business-specific rules
- **Input Sanitization**: Prevent XSS and SQL injection

Testing Requirements

- **Unit Tests**: 85% coverage minimum for service layer
- **Integration Tests**: All controller endpoints
- **Repository Tests**: Custom query methods
- **JSP Tests**: Selenium for critical user flows

Security Requirements

- **Authentication**: Spring Security with role-based access
- **Authorization**: Method-level security annotations
- **CSRF Protection**: Enabled for state-changing operations
- **Input Validation**: Server-side validation for all forms

Project Structure:

```
Employee_Management_System/
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com/
│   │   │       └── bjet/
│   │   │           └── ems/
│   │   │               ├── EmsApplication.java
│   │   ├── resources/
│   │   │   └── application.properties
│   │   └── webapp/
│   │       └── WEB-INF/
│   └── test/
│       └── java/
│           └── com/
│               └── bjet/
│                   └── Employee_Management_System/
│                       └── TestEmsApplication.java
└── build.gradle
```

Project Name: Employee_Management_System

Project Package/artifact: com.bjet.ems

Main Class: EmsApplication.java

Main Test Class: TestEmsApplication.java