HR Desk project

Stefano

22 Maggio 2025

Sommario

Report su progetto Ruby sozinfo - Sistema di gestione ufficio per la gestione delle risorse umane, permessi, ferie e documenti aziendali

Indice

Codebase Explanation: Office Management System

This document provides a comprehensive overview of the Office Management System codebase. It is designed to help new developers quickly understand the project's architecture, key technologies, and how different components interact.

1. Project Architecture

The Office Management System follows a layered architecture, which promotes separation of concerns and maintainability. The main layers are:

- Resource Layer (API): Exposes RESTful endpoints for interacting with the system.
- Model Layer: Defines the data structures and entities used in the application.
- Persistence Layer: Handles the storage and retrieval of data from the database.
- Configuration Layer: Configures the application and its dependencies.

  Note: This project is primarily designed for tutoring purposes and deliberately takes some shortcuts that would normally be avoided in production applications. These simplifications are meant to keep the codebase compact and focused on key concepts rather than architectural purity.

  Simplified architecture includes: - Model and persistence layers are fused together (entities serve both as data models and persistence objects) - No separate DTOs (using entities directly in the API layer, except for specific cases like stats) - No Lombok to reduce boilerplate code (manual getters/setters used throughout) - Minimal abstraction layers (direct Hibernate session usage in resources)

2. REST API

The application exposes a REST API using Jersey (JAX-RS). The API endpoints are defined in the `com.officemanagement.resource` package.

Key Concepts:

- JAX-RS Annotations: Used to define API endpoints, HTTP methods, request/response formats, and parameters.
- Resource Classes: Handle incoming requests, interact with the persistence layer, and return responses.
- Input Validation: Implemented in resource classes to ensure data integrity.

Entity Instead of DTO Pattern:

Unlike production applications, this tutoring project uses entities directly in the API, which has some drawbacks: - Potential Hibernate stale data issues: Updates might not be immediately reflected without explicit reloading - Serialization challenges: Risk of circular references in bidirectional relationships - Tight coupling: Changes to database structure directly impact API contracts - Over-exposure: Internal model details are exposed in the API

The resource classes mitigate these issues in some cases by manually constructing response objects or using dedicated DTOs (as seen in the StatsResource).

Available Resources:

The API provides the following resource endpoints:

- FloorResource (`/api/floors`): Floor management operations
- RoomResource (`/api/rooms`): Room management and geometry updates
- SeatResource (`/api/seats`): Seat management and assignments
- EmployeeResource (`/api/employees`): Employee management with search and pagination
- StatsResource (`/api/stats`): System statistics (total counts)

Example Endpoints:

- `GET /api/floors`: Retrieves all floors.
- `GET /api/floors/{id}`: Retrieves a specific floor by ID.
- `POST /api/floors`: Creates a new floor.
- `PUT /api/floors/{id}`: Updates an existing floor.
- `DELETE /api/floors/{id}`: Deletes a floor.
- `PATCH /api/rooms/{id}/geometry`: Updates room geometry including positions of contained seats.
- `GET /api/employees?search={query}&page={page}&size={size}`: Search employees with pagination.
- `GET /api/stats`: Retrieve system statistics.

Refer to `docs/api-tests.http` for a complete list of endpoints and example requests.

3. Persistence Layer

The persistence layer is responsible for interacting with the PostgreSQL database. It uses Hibernate as the Object-Relational Mapping (ORM) framework.

Key Components:

- Hibernate Configuration (`src/main/resources/hibernate.cfg.xml`): Specifies the database connection details, dialect, and mapping files.
- Entity Classes (`com.officemanagement.model`): Represent the database tables and are annotated with JPA annotations (e.g., `@Entity`, `@Table`, `@Id`, `@GeneratedValue`, `@Column`).
- HibernateUtil (`com.officemanagement.util.HibernateUtil`): Provides a utility class for obtaining the `SessionFactory` and managing database sessions.
- Hibernate Context Listener (`com.officemanagement.config.HibernateContextListener`): Initializes and shuts down the Hibernate `SessionFactory` during application startup and shutdown.
- HikariCP (`src/main/resources/hikari.properties`): Provides connection pooling for efficient database access.

Direct Session Usage:

For simplicity, this project doesn't use repositories or DAOs. Resource classes directly manage Hibernate sessions:

```
try (Session session = sessionFactory.openSession()) {
    // Perform database operations
    Floor floor = session.get(Floor.class, id);
    // ...
}
```

This approach keeps the code simple but sacrifices some separation of concerns.

Database Schema:

The database schema is defined in `.devcontainer/schema.sql`. It includes tables for:

- `floors`: Stores information about floors (id, floor_number, name, created_at).
- `floor_planimetry`: Stores SVG floor plan data separately for performance (floor_id, planimetry, last_updated).
- `office_rooms`: Stores information about office rooms (id, room_number, name, floor_id, x, y, width, height, created_at).

- `seats`: Stores information about seats (id, seat_number, room_id, employee_id, x, y, width, height, rotation, created_at).
- `employees`: Stores information about employees (id, full_name, occupation, created_at).
- `employee_seat_assignments`: Many-to-many relationship table for employee-seat assignments.

Entity Models:

The project includes the following entity classes:

- Floor: Represents office floors with one-to-many relationship to rooms and one-to-one relationship to floor planimetry.
- FloorPlanimetry: Separate entity for storing SVG floor plan data, optimizing performance by separating large text data.
- OfficeRoom: Represents rooms within floors, includes position and dimension data for visualization.
- Seat: Represents individual seats within rooms, includes position, dimension, and rotation data for floor plan visualization.
- Employee: Represents employees with basic profile information.

4. Key Technologies and Libraries

- Java 11: The primary programming language.
- Jersey (JAX-RS) 2.34: Framework for building RESTful APIs.
- Hibernate 5.6: ORM framework for mapping Java objects to database tables.
- PostgreSQL: Relational database used for data storage.
- HikariCP 6.2.1: High-performance JDBC connection pooling library.
- Maven: Build automation tool for managing dependencies and building the application.
- JUnit Jupiter 5.8.2: Modern testing framework for writing and running unit tests.
- Mockito 4.2.0: Mocking framework for creating mock objects in unit tests.
- REST Assured 5.3.0: Library for testing REST APIs.
- Jackson 2.13.0: Library for JSON serialization and deserialization with JSR310 support.
- Dev Containers: Provides consistent development environment using Docker.
- Devbox: Modern development environment management.

5. Configuration

Jersey Configuration:

- `com.officemanagement.config.JerseyConfig`: Configures Jersey to scan the `com.officemanagement.resource` package for resource classes and registers Jackson for JSON serialization.

Hibernate Configuration:

- `src/main/resources/hibernate.cfg.xml`: Configures the database connection, dialect, and mapping files.

Connection Pooling:

- `src/main/resources/hikari.properties`: Configures HikariCP connection pool settings for optimal database performance.

Logging Configuration:

- `src/main/resources/logback.xml`: Configures the logging framework (Logback) to output logs to the console and a file.

6. Testing

The project includes unit tests and integration tests to ensure the quality and stability of the codebase.

- Unit Tests: Located in `src/test/java`. Test individual components and classes in isolation using Mockito for mocking dependencies.
- Integration Tests: Located in `src/test/java/com/officemanagement/resource`. Test the REST API endpoints using REST Assured and the Jersey test framework.
- Test Database: H2 in-memory database used for testing to avoid dependencies on external PostgreSQL instance.

7. Development Environment

The project supports multiple development environment options:

VSCode Dev Containers

The project uses VSCode's Dev Containers feature to provide a consistent and reproducible development environment. The Dev Container configuration includes:

- A `Dockerfile` that defines the base image and installs the necessary dependencies.
- A `devcontainer.json` file that configures VSCode to use the Dev Container.
- A `docker-compose.yml` file that defines the services (e.g., the application and the database) that are included in the Dev Container.

Devbox Environment

The project also supports Devbox for development environment management:

- `devbox.json`: Defines the development environment with specific tool versions.
- `devbox.lock`: Locks specific versions for reproducible environments.

8. Build Process

The project uses Maven for building and packaging the application. The build process is defined in the `pom.xml` file.

Key Maven Goals:

- `mvn clean`: Cleans the project by deleting the `target` directory.
- `mvn compile`: Compiles the Java source code.
- `mvn test`: Runs the unit tests.
- `mvn package`: Packages the application into a WAR file.
- `mvn install`: Installs the WAR file into the local Maven repository.
- `mvn cargo:run`: Starts the application using the embedded Tomcat server.

Cargo Plugin:

The project uses the Cargo Maven plugin to run the application with an embedded Tomcat 9 server for development and testing purposes.

9. Important Packages

- `com.officemanagement.config`: Contains configuration classes for Jersey, Hibernate, and other components.
- `com.officemanagement.filter`: Contains servlet filters (e.g., `CORSFilter`) for handling cross-origin requests.
- `com.officemanagement.model`: Contains the entity classes that represent the database tables.
- `com.officemanagement.resource`: Contains the resource classes that define the REST API endpoints.
- `com.officemanagement.util`: Contains utility classes (e.g., `HibernateUtil`) for managing Hibernate sessions.

10. Special Features

Floor Plan Management

The system includes sophisticated floor plan management capabilities:

- SVG Floor Plans: Stored in the `floor_planimetry` table as TEXT data.
- Sample Floor Plans: Pre-loaded SVG floor plans available in `.devcontainer/svg_floor_plans/`.
- Floor Plan Loading: Automated scripts for loading SVG floor plans into the database.
- Geometry Updates: API endpoints for updating room and seat positions/dimensions on floor plans.

Pagination and Search

The system provides advanced query capabilities:

- Employee Search: Full-text search across employee names and occupations.
- Pagination: Configurable page size and navigation for large data sets.
- Query Parameters: Flexible filtering and sorting options.

Statistics API

Dedicated statistics endpoint providing system-wide metrics:

- Total employee count
- Total floor count

- Total office count
- Total seat count

11. Authentication and Authorization Implementation Guide

> Important: The current system lacks authentication and authorization. This section provides comprehensive guidance for students who need to implement these security features for the first time.

11.1 Authentication vs Authorization

Before diving into implementation, it's crucial to understand the difference:

- Authentication: Verifying WHO the user is (login process)
- Authorization: Determining WHAT the authenticated user can do (permissions)

11.2 JWT (JSON Web Tokens) Fundamentals

What is JWT? JWT is a compact, URL-safe means of representing claims between two parties. It's a standard (RFC 7519) for securely transmitting information between parties as a JSON object.

JWT Structure A JWT consists of three parts separated by dots (.):

```
header.payload.signature
```

Example JWT:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4f
```

1. Header:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

2. Payload (Claims):

```
{
  "sub": "1234567890",    // Subject (user ID)
  "name": "John Doe",     // Custom claim
  "iat": 1516239022,      // Issued at
  "exp": 1516242622,      // Expiration time
  "roles": ["ADMIN", "USER"] // Custom roles claim
}
```

3. Signature:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret
)
```

JWT Benefits for this Project:

- Stateless: No need to store sessions on the server
- Scalable: Works well with microservices
- Cross-domain: Can be used across different domains
- Self-contained: Contains all necessary information

11.3 Database Schema Extensions

To implement authentication, you'll need to extend the database schema:

```sql
-- User authentication table
CREATE TABLE users (
    id BIGINT DEFAULT nextval('user_seq') PRIMARY KEY,
    username VARCHAR(255) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    salt VARCHAR(255) NOT NULL,
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    last_login TIMESTAMP,
    failed_login_attempts INTEGER DEFAULT 0,
    locked_until TIMESTAMP
);

-- Roles table
CREATE TABLE roles (
    id BIGINT DEFAULT nextval('role_seq') PRIMARY KEY,
    name VARCHAR(50) UNIQUE NOT NULL,
    description VARCHAR(255),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- User-role relationships (many-to-many)
CREATE TABLE user_roles (
    user_id BIGINT REFERENCES users(id) ON DELETE CASCADE,
    role_id BIGINT REFERENCES roles(id) ON DELETE CASCADE,
    assigned_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (user_id, role_id)
);

-- Link users to employees (optional)
CREATE TABLE user_employee_mapping (
    user_id BIGINT REFERENCES users(id) ON DELETE CASCADE,
    employee_id BIGINT REFERENCES employees(id) ON DELETE CASCADE,
    PRIMARY KEY (user_id, employee_id)
);

-- Insert default roles
INSERT INTO roles (name, description) VALUES
```

```
('ADMIN', 'Full system access'),
('MANAGER', 'Floor and room management'),
('EMPLOYEE', 'Basic access to view and update own information'),
('VIEWER', 'Read-only access');
```

11.4 New Entity Classes

User Entity

```
package com.officemanagement.model;

import javax.persistence.*;
import java.time.LocalDateTime;
import java.util.HashSet;
import java.util.Set;

@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "user_seq")
    @SequenceGenerator(name = "user_seq", sequenceName = "user_seq", allocationSize = 1)
    private Long id;

    @Column(unique = true, nullable = false)
    private String username;

    @Column(unique = true, nullable = false)
    private String email;

    @Column(name = "password_hash", nullable = false)
    private String passwordHash;

    @Column(nullable = false)
    private String salt;

    @Column(name = "is_active")
    private Boolean isActive = true;

    @Column(name = "created_at")
    private LocalDateTime createdAt;

    @Column(name = "last_login")
    private LocalDateTime lastLogin;

    @Column(name = "failed_login_attempts")
    private Integer failedLoginAttempts = 0;

    @Column(name = "locked_until")
    private LocalDateTime lockedUntil;

    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(
        name = "user_roles",
        joinColumns = @JoinColumn(name = "user_id"),
```

```
        inverseJoinColumns = @JoinColumn(name = "role_id")
    )
    private Set<Role> roles = new HashSet<>();

    @OneToOne
    @JoinTable(
        name = "user_employee_mapping",
        joinColumns = @JoinColumn(name = "user_id"),
        inverseJoinColumns = @JoinColumn(name = "employee_id")
    )
    private Employee employee;

    // Constructors, getters, and setters
    public User() {}

    public User(String username, String email) {
        this.username = username;
        this.email = email;
        this.createdAt = LocalDateTime.now();
    }

    // ... getters and setters for all fields
}
```

Role Entity

```
package com.officemanagement.model;

import javax.persistence.*;
import java.time.LocalDateTime;

@Entity
@Table(name = "roles")
public class Role {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "role_seq")
    @SequenceGenerator(name = "role_seq", sequenceName = "role_seq", allocationSize = 1)
    private Long id;

    @Column(unique = true, nullable = false)
    private String name;

    private String description;

    @Column(name = "created_at")
    private LocalDateTime createdAt;

    // Constructors, getters, and setters
    public Role() {}

    public Role(String name, String description) {
        this.name = name;
        this.description = description;
        this.createdAt = LocalDateTime.now();
    }
```

```
    // ... getters and setters
}
```

11.5 Security Utilities

Password Security

```
package com.officemanagement.util;

import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;
import java.security.spec.InvalidKeySpecException;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;
import java.util.Base64;

public class PasswordUtil {
    private static final int SALT_LENGTH = 32;
    private static final int HASH_LENGTH = 64;
    private static final int ITERATIONS = 100000; // OWASP recommended minimum

    /**
     * Generates a random salt for password hashing
     */
    public static String generateSalt() {
        SecureRandom random = new SecureRandom();
        byte[] salt = new byte[SALT_LENGTH];
        random.nextBytes(salt);
        return Base64.getEncoder().encodeToString(salt);
    }

    /**
     * Hashes a password with the given salt using PBKDF2
     */
    public static String hashPassword(String password, String salt)
            throws NoSuchAlgorithmException, InvalidKeySpecException {
        byte[] saltBytes = Base64.getDecoder().decode(salt);

        PBEKeySpec spec = new PBEKeySpec(
            password.toCharArray(),
            saltBytes,
            ITERATIONS,
            HASH_LENGTH * 8
        );

        SecretKeyFactory factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
        byte[] hash = factory.generateSecret(spec).getEncoded();

        return Base64.getEncoder().encodeToString(hash);
    }

    /**
     * Verifies a password against its hash
     */
```

```
    public static boolean verifyPassword(String password, String salt, String expectedHash)
            throws NoSuchAlgorithmException, InvalidKeySpecException {
        String actualHash = hashPassword(password, salt);
        return actualHash.equals(expectedHash);
    }
}
```

JWT Utility

```
package com.officemanagement.util;

import io.jsonwebtoken.*;
import io.jsonwebtoken.security.Keys;
import java.security.Key;
import java.util.Date;
import java.util.List;
import java.util.stream.Collectors;
import com.officemanagement.model.User;
import com.officemanagement.model.Role;

public class JwtUtil {
    // In production, load this from environment variables or secure configuration
    private static final String SECRET_KEY = "MySecretKeyThatIsAtLeast256BitsLongForHS256Algorithm";
    private static final Key key = Keys.hmacShaKeyFor(SECRET_KEY.getBytes());

    // Token validity: 24 hours
    private static final long EXPIRATION_TIME = 86400000;

    /**
     * Generate JWT token for authenticated user
     */
    public static String generateToken(User user) {
        List<String> roles = user.getRoles().stream()
            .map(Role::getName)
            .collect(Collectors.toList());

        return Jwts.builder()
            .setSubject(user.getId().toString())
            .claim("username", user.getUsername())
            .claim("email", user.getEmail())
            .claim("roles", roles)
            .setIssuedAt(new Date())
            .setExpiration(new Date(System.currentTimeMillis() + EXPIRATION_TIME))
            .signWith(key, SignatureAlgorithm.HS256)
            .compact();
    }

    /**
     * Extract username from JWT token
     */
    public static String getUsernameFromToken(String token) {
        Claims claims = getClaimsFromToken(token);
        return claims.get("username", String.class);
    }
```

```
    /**
     * Extract user ID from JWT token
     */
    public static Long getUserIdFromToken(String token) {
        Claims claims = getClaimsFromToken(token);
        return Long.parseLong(claims.getSubject());
    }

    /**
     * Extract roles from JWT token
     */
    @SuppressWarnings("unchecked")
    public static List<String> getRolesFromToken(String token) {
        Claims claims = getClaimsFromToken(token);
        return claims.get("roles", List.class);
    }

    /**
     * Check if token is expired
     */
    public static boolean isTokenExpired(String token) {
        try {
            Claims claims = getClaimsFromToken(token);
            return claims.getExpiration().before(new Date());
        } catch (JwtException e) {
            return true;
        }
    }

    /**
     * Validate JWT token
     */
    public static boolean validateToken(String token) {
        try {
            getClaimsFromToken(token);
            return !isTokenExpired(token);
        } catch (JwtException | IllegalArgumentException e) {
            return false;
        }
    }

    private static Claims getClaimsFromToken(String token) {
        return Jwts.parserBuilder()
            .setSigningKey(key)
            .build()
            .parseClaimsJws(token)
            .getBody();
    }
}
```

11.6 Authentication Resource

```
package com.officemanagement.resource;

import javax.ws.rs.*;
```

```
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.query.Query;
import com.officemanagement.util.HibernateUtil;
import com.officemanagement.util.PasswordUtil;
import com.officemanagement.util.JwtUtil;
import com.officemanagement.model.User;
import com.fasterxml.jackson.annotation.JsonProperty;
import java.time.LocalDateTime;

@Path("/auth")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class AuthResource {

    private final SessionFactory sessionFactory;

    public AuthResource() {
        this.sessionFactory = HibernateUtil.getSessionFactory();
    }

    // DTOs for requests and responses
    public static class LoginRequest {
        @JsonProperty("username")
        private String username;

        @JsonProperty("password")
        private String password;

        // Constructors, getters, setters
        public LoginRequest() {}

        public String getUsername() { return username; }
        public void setUsername(String username) { this.username = username; }
        public String getPassword() { return password; }
        public void setPassword(String password) { this.password = password; }
    }

    public static class LoginResponse {
        @JsonProperty("token")
        private final String token;

        @JsonProperty("user")
        private final UserInfo user;

        public LoginResponse(String token, UserInfo user) {
            this.token = token;
            this.user = user;
        }
    }

    public static class UserInfo {
```

```
    @JsonProperty("id")
    private final Long id;

    @JsonProperty("username")
    private final String username;

    @JsonProperty("email")
    private final String email;

    @JsonProperty("roles")
    private final java.util.List<String> roles;

    public UserInfo(User user) {
        this.id = user.getId();
        this.username = user.getUsername();
        this.email = user.getEmail();
        this.roles = user.getRoles().stream()
            .map(role -> role.getName())
            .collect(java.util.stream.Collectors.toList());
    }
}

@POST
@Path("/login")
public Response login(LoginRequest loginRequest) {
    if (loginRequest.getUsername() == null || loginRequest.getPassword() == null) {
        return Response.status(Response.Status.BAD_REQUEST)
            .entity(new ErrorResponse("Username and password are required"))
            .build();
    }

    Session session = null;
    try {
        session = sessionFactory.openSession();

        // Find user by username or email
        Query<User> query = session.createQuery(
            "FROM User u WHERE u.username = :identifier OR u.email = :identifier",
            User.class
        );
        query.setParameter("identifier", loginRequest.getUsername());
        User user = query.uniqueResult();

        if (user == null) {
            return Response.status(Response.Status.UNAUTHORIZED)
                .entity(new ErrorResponse("Invalid credentials"))
                .build();
        }

        // Check if account is locked
        if (user.getLockedUntil() != null && user.getLockedUntil().isAfter(LocalDateTime.now())) {
            return Response.status(Response.Status.LOCKED)
                .entity(new ErrorResponse("Account is temporarily locked"))
                .build();
```

```
        }

        // Check if account is active
        if (!user.getIsActive()) {
            return Response.status(Response.Status.FORBIDDEN)
                .entity(new ErrorResponse("Account is disabled"))
                .build();
        }

        // Verify password
        try {
            if (!PasswordUtil.verifyPassword(
                loginRequest.getPassword(),
                user.getSalt(),
                user.getPasswordHash()
            )) {
                handleFailedLogin(session, user);
                return Response.status(Response.Status.UNAUTHORIZED)
                    .entity(new ErrorResponse("Invalid credentials"))
                    .build();
            }
        } catch (Exception e) {
            return Response.status(Response.Status.INTERNAL_SERVER_ERROR)
                .entity(new ErrorResponse("Authentication error"))
                .build();
        }

        // Successful login
        handleSuccessfulLogin(session, user);
        String token = JwtUtil.generateToken(user);

        return Response.ok(new LoginResponse(token, new UserInfo(user))).build();

    } catch (Exception e) {
        return Response.status(Response.Status.INTERNAL_SERVER_ERROR)
            .entity(new ErrorResponse("Login failed: " + e.getMessage()))
            .build();
    } finally {
        if (session != null && session.isOpen()) {
            session.close();
        }
    }
}

@POST
@Path("/validate")
@Consumes(MediaType.TEXT_PLAIN)
public Response validateToken(String token) {
    if (JwtUtil.validateToken(token)) {
        return Response.ok(new SuccessResponse("Token is valid")).build();
    } else {
        return Response.status(Response.Status.UNAUTHORIZED)
            .entity(new ErrorResponse("Invalid or expired token"))
            .build();
```

```
    }
}

private void handleFailedLogin(Session session, User user) {
    try {
        session.beginTransaction();
        user.setFailedLoginAttempts(user.getFailedLoginAttempts() + 1);

        // Lock account after 5 failed attempts for 30 minutes
        if (user.getFailedLoginAttempts() >= 5) {
            user.setLockedUntil(LocalDateTime.now().plusMinutes(30));
        }

        session.update(user);
        session.getTransaction().commit();
    } catch (Exception e) {
        if (session.getTransaction() != null) {
            session.getTransaction().rollback();
        }
    }
}

private void handleSuccessfulLogin(Session session, User user) {
    try {
        session.beginTransaction();
        user.setLastLogin(LocalDateTime.now());
        user.setFailedLoginAttempts(0);
        user.setLockedUntil(null);
        session.update(user);
        session.getTransaction().commit();
    } catch (Exception e) {
        if (session.getTransaction() != null) {
            session.getTransaction().rollback();
        }
    }
}

// Helper classes for responses
private static class ErrorResponse {
    @JsonProperty("message")
    private final String message;

    public ErrorResponse(String message) {
        this.message = message;
    }
}

private static class SuccessResponse {
    @JsonProperty("message")
    private final String message;

    public SuccessResponse(String message) {
        this.message = message;
    }
```

```
    }
}
```

11.7 JWT Authentication Filter

```
package com.officemanagement.filter;

import javax.annotation.Priority;
import javax.ws.rs.Priorities;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.core.HttpHeaders;
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.Provider;
import com.officemanagement.util.JwtUtil;
import java.io.IOException;
import java.util.List;

@Provider
@Priority(Priorities.AUTHENTICATION)
public class JwtAuthenticationFilter implements ContainerRequestFilter {

    private static final String BEARER_PREFIX = "Bearer ";

    @Override
    public void filter(ContainerRequestContext requestContext) throws IOException {
        String path = requestContext.getUriInfo().getPath();

        // Skip authentication for login endpoint and health checks
        if (path.equals("auth/login") || path.equals("health")) {
            return;
        }

        String authorizationHeader = requestContext.getHeaderString(HttpHeaders.AUTHORIZATION);

        if (authorizationHeader == null || !authorizationHeader.startsWith(BEARER_PREFIX)) {
            abortWithUnauthorized(requestContext, "Missing or invalid Authorization header");
            return;
        }

        String token = authorizationHeader.substring(BEARER_PREFIX.length()).trim();

        if (!JwtUtil.validateToken(token)) {
            abortWithUnauthorized(requestContext, "Invalid or expired token");
            return;
        }

        // Add user information to request context for use in resources
        try {
            Long userId = JwtUtil.getUserIdFromToken(token);
            String username = JwtUtil.getUsernameFromToken(token);
            List<String> roles = JwtUtil.getRolesFromToken(token);

            requestContext.setProperty("userId", userId);
            requestContext.setProperty("username", username);
```

```
            requestContext.setProperty("roles", roles);
        } catch (Exception e) {
            abortWithUnauthorized(requestContext, "Token processing error");
        }
    }

    private void abortWithUnauthorized(ContainerRequestContext requestContext, String message) {
        Response response = Response.status(Response.Status.UNAUTHORIZED)
            .entity("{\"message\": \"" + message + "\"}")
            .build();
        requestContext.abortWith(response);
    }
}
```

11.8 Role-Based Authorization

Authorization Annotations

```
package com.officemanagement.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface RequireRole {
    String[] value();
}
```

Authorization Filter

```
package com.officemanagement.filter;

import javax.annotation.Priority;
import javax.ws.rs.Priorities;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.container.ResourceInfo;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.Response;
import javax.ws.rs.ext.Provider;
import com.officemanagement.annotation.RequireRole;
import java.io.IOException;
import java.lang.reflect.Method;
import java.util.Arrays;
import java.util.List;

@Provider
@Priority(Priorities.AUTHORIZATION)
public class RoleAuthorizationFilter implements ContainerRequestFilter {

    @Context
    private ResourceInfo resourceInfo;
```

```
    @Override
    public void filter(ContainerRequestContext requestContext) throws IOException {
        Method method = resourceInfo.getResourceMethod();
        Class<?> resourceClass = resourceInfo.getResourceClass();

        RequireRole methodAnnotation = method.getAnnotation(RequireRole.class);
        RequireRole classAnnotation = resourceClass.getAnnotation(RequireRole.class);

        if (methodAnnotation == null && classAnnotation == null) {
            return; // No role requirements
        }

        @SuppressWarnings("unchecked")
        List<String> userRoles = (List<String>) requestContext.getProperty("roles");

        if (userRoles == null || userRoles.isEmpty()) {
            abortWithForbidden(requestContext, "No roles assigned to user");
            return;
        }

        String[] requiredRoles = methodAnnotation != null ?
            methodAnnotation.value() : classAnnotation.value();

        boolean hasRequiredRole = Arrays.stream(requiredRoles)
            .anyMatch(userRoles::contains);

        if (!hasRequiredRole) {
            abortWithForbidden(requestContext, "Insufficient privileges");
        }
    }

    private void abortWithForbidden(ContainerRequestContext requestContext, String message) {
        Response response = Response.status(Response.Status.FORBIDDEN)
            .entity("{\"message\": \"" + message + "\"}")
            .build();
        requestContext.abortWith(response);
    }
}
```

11.9 Updated Resource Classes

Add authorization to existing resources:

```
// Example: Updated FloorResource with role-based access
@Path("/floors")
@RequireRole({"ADMIN", "MANAGER"}) // Class-level: only ADMIN and MANAGER can access floors
public class FloorResource {

    @GET
    @RequireRole({"ADMIN", "MANAGER", "EMPLOYEE", "VIEWER"}) // Method-level: anyone can view
    public Response getAllFloors() {
        // ... existing implementation
    }

    @POST
```

```
    @RequireRole({"ADMIN"}) // Only ADMIN can create floors
    public Response createFloor(Floor floor) {
        // ... existing implementation
    }

    @DELETE
    @Path("/{id}")
    @RequireRole({"ADMIN"}) // Only ADMIN can delete floors
    public Response deleteFloor(@PathParam("id") Long id) {
        // ... existing implementation
    }
}
```

11.10 Maven Dependencies

Add these dependencies to your `pom.xml`:

```
<!-- JWT -->
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>

<!-- BCrypt for password hashing (alternative to PBKDF2) -->
<dependency>
    <groupId>org.mindrot</groupId>
    <artifactId>jbcrypt</artifactId>
    <version>0.4</version>
</dependency>
```

11.11 Configuration Updates

Register Filters in JerseyConfig

```
@ApplicationPath("/api")
public class JerseyConfig extends ResourceConfig {
    public JerseyConfig() {
        packages("com.officemanagement.resource");
        register(JacksonFeature.class);
        register(ObjectMapperContextResolver.class);

        // Register authentication and authorization filters
        register(JwtAuthenticationFilter.class);
```

```
        register(RoleAuthorizationFilter.class);

        // Register CORS filter if needed
        register(CORSFilter.class);
    }
}
```

11.12 Security Best Practices

1. Password Security

- Never store plaintext passwords
- Use strong hashing algorithms (PBKDF2, bcrypt, scrypt, or Argon2)
- Always use a unique salt per password
- Implement minimum password complexity requirements

2. JWT Security

- Secret Management: Store JWT secrets in environment variables, never in code
- Token Expiration: Keep token lifetimes short (1-24 hours)
- Refresh Tokens: Implement refresh token mechanism for longer sessions
- Blacklisting: Consider token blacklisting for logout functionality

3. Account Security

- Rate Limiting: Implement login attempt rate limiting
- Account Locking: Lock accounts after multiple failed attempts
- Session Management: Track user sessions and provide logout functionality

4. HTTPS Only

- Always use HTTPS in production
- Set secure flags on cookies
- Implement HSTS headers

11.13 Testing Authentication

Unit Test Example

```
@ExtendWith(MockitoExtension.class)
class AuthResourceTest {

    @Mock
    private SessionFactory sessionFactory;

    @Mock
    private Session session;

    @Mock
    private Query<User> query;

    private AuthResource authResource;

    @BeforeEach
    void setUp() {
        when(sessionFactory.openSession()).thenReturn(session);
        authResource = new AuthResource();
```

```
        // Use reflection to inject mock sessionFactory
    }

    @Test
    void login_ValidCredentials_ReturnsToken() {
        // Arrange
        User mockUser = createMockUser();
        when(session.createQuery(anyString(), eq(User.class))).thenReturn(query);
        when(query.uniqueResult()).thenReturn(mockUser);

        LoginRequest request = new LoginRequest();
        request.setUsername("testuser");
        request.setPassword("correctpassword");

        // Act
        Response response = authResource.login(request);

        // Assert
        assertEquals(200, response.getStatus());
        assertNotNull(response.getEntity());
    }

    @Test
    void login_InvalidCredentials_ReturnsUnauthorized() {
        // Arrange
        when(session.createQuery(anyString(), eq(User.class))).thenReturn(query);
        when(query.uniqueResult()).thenReturn(null);

        LoginRequest request = new LoginRequest();
        request.setUsername("wronguser");
        request.setPassword("wrongpassword");

        // Act
        Response response = authResource.login(request);

        // Assert
        assertEquals(401, response.getStatus());
    }

    private User createMockUser() {
        // Create and return a mock user with proper password hash
        User user = new User("testuser", "test@example.com");
        user.setId(1L);
        user.setIsActive(true);
        // Set other required fields
        return user;
    }
}
```

11.14 Common Implementation Pitfalls

1. Security Vulnerabilities

- Don't store JWT secrets in code or version control
- Don't use weak password hashing algorithms (MD5, SHA1)

- Don't expose sensitive information in error messages
- Don't forget to validate JWT signatures

2. Performance Issues

- Don't make database calls on every request for token validation
- Don't use overly complex role hierarchies without caching
- Consider caching user roles and permissions

3. User Experience

- Don't lock accounts permanently without recovery mechanism
- Don't make error messages too specific (information disclosure)
- Do provide clear feedback for password requirements

11.15 Implementation Checklist

- ☐ Create database schema for users, roles, and relationships
- ☐ Implement User and Role entities with proper relationships
- ☐ Create password hashing utility with salt generation
- ☐ Implement JWT utility for token generation and validation
- ☐ Create authentication resource with login endpoint
- ☐ Implement JWT authentication filter
- ☐ Create role-based authorization annotations and filter
- ☐ Update existing resources with appropriate role requirements
- ☐ Add JWT dependencies to Maven configuration
- ☐ Register filters in Jersey configuration
- ☐ Write comprehensive tests for authentication logic
- ☐ Implement account security features (locking, rate limiting)
- ☐ Set up proper secret management for production
- ☐ Add API documentation for authentication endpoints
- ☐ Implement logout functionality with token invalidation (optional)
- ☐ Add refresh token mechanism (optional)

This comprehensive guide provides everything needed to implement secure authentication and authorization in the office management system using industry-standard practices and JWT tokens.

12. Educational Purpose

This project is designed primarily as a tutoring tool to demonstrate key Java web application concepts in a straightforward manner. The simplifications chosen allow students to:

- See direct relationships between API endpoints and database operations
- Understand Hibernate entity mappings without complex abstraction layers
- Focus on core REST concepts without getting lost in architectural complexity
- Experience common pitfalls firsthand (like Hibernate lazy loading issues or serialization cycles)
- Learn modern testing practices with JUnit 5 and Mockito

In a production application, you would likely: * Implement proper DTOs for API contracts * Add repository/DAO layer between resources and database * Use Lombok or record classes to reduce boilerplate * Add more validation, error handling, and security features * Implement proper transaction management * Add caching strategies * Implement proper authentication and authorization

13. Next Steps

- Explore the codebase and familiarize yourself with the different packages and classes.
- Run the unit tests and integration tests to verify the functionality of the application.
- Experiment with the REST API endpoints using a tool like curl or Postman, or the provided `docs/api-tests.http` file.
- Try loading different floor plans using the provided SVG files and scripts.

- Contribute to the project by fixing bugs, adding new features, or improving the documentation.
- Experiment with the search and pagination features in the Employee API.
- Test the geometry update functionality for rooms and seats.