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) - 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.

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.

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).
- `office_rooms`: Stores information about office rooms (id, room_number, name, floor_id, created_at).
- `seats`: Stores information about seats (id, seat_number, room_id, employee_id, created_at).
- `employees`: Stores information about employees (id, full_name, occupation, created_at).

4. Key Technologies and Libraries

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

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.

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.

7. Development Environment

The project uses VSCode Dev Containers 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.

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.

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. 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)

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

11. 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.

- Contribute to the project by fixing bugs, adding new features, or improving the documentation.