# Estate Management Website

## Object-Oriented Programming - Project Report

Nguyen Dinh An - 20226007

Nguyen Huu Hoang Hai Anh - 20226010

Tran Anh Dung - 20226031

Le Ngoc Quang Hung - 20226069

2024-06-06

# Contents

# 1. Introduction

## 1.1. Background

Estate management systems are crucial tools for managing properties, whether residential, commercial, or industrial. These systems help streamline operations such as property listings, tenant management, lease agreements, maintenance requests, and financial transactions. With the increasing digital transformation across industries, a robust and efficient back-end for an estate management website is essential to handle the complex workflows and large datasets involved in property management.

## 1.2. Objective

The primary objective of this project is to develop a back-end system for an estate management website using modern Java-based frameworks. The back-end will provide a comprehensive API to support functionalities such as property listing, user authentication, role-based access control, and data persistence.

## 1.3. Scope

The project will cover the implementation of key features required for an estate management system, such as:

- Staff management (registration, authentication, and role-based access control)
- Property management (listing, updating, and deleting properties)

This project will not cover the front-end implementation, real-time data processing, or advanced analytics features, which could be considered in future extensions of the project.

## 1.4. Technology Used

To build a robust and efficient back-end system, aside from Java, the following technologies will be employed:

- Spring Boot: For creating a stand-alone, production-grade Spring-based application with minimal configuration.
- Spring MVC: For handling web requests and building RESTful APIs.
- Spring Data JPA: For data persistence and ORM capabilities, simplifying database interactions.
- Spring Security: For implementing security features such as authentication and authorization.

# 2. Use Case

## 2.1. Actors

This section discusses the actors of the system

### 2.1.1. Manager

Managers are responsible for managing staff and properties, oversee the system operations

### 2.1.2. Staff

Staffs register on the platform, views and search for properties

## 2.2. Use Case Diagram

Here is the Use Case diagram illustrating the functionalities of the system

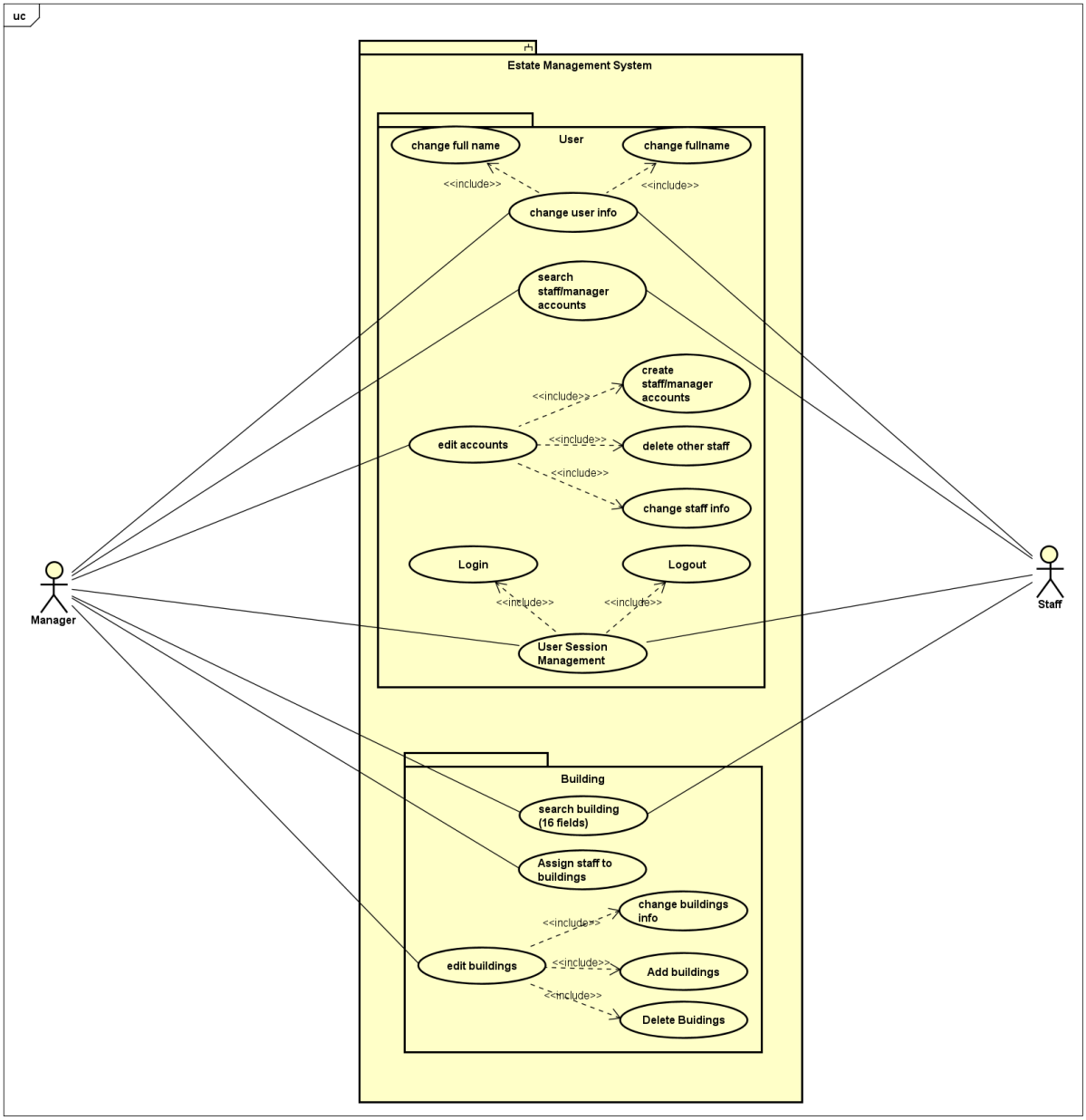



Figure 1: Use Case Diagram

As seen from the figure, the Manager has full access to the system's functionalities, while the staff has limited access.

# 3. Design

## 3.1. Architecture

The estate management back-end is built using a three-layer architecture: Presentation Layer, Business Logic Layer, and Data Access Layer. This architecture promotes separation of concerns, making the system easier to maintain and extend.

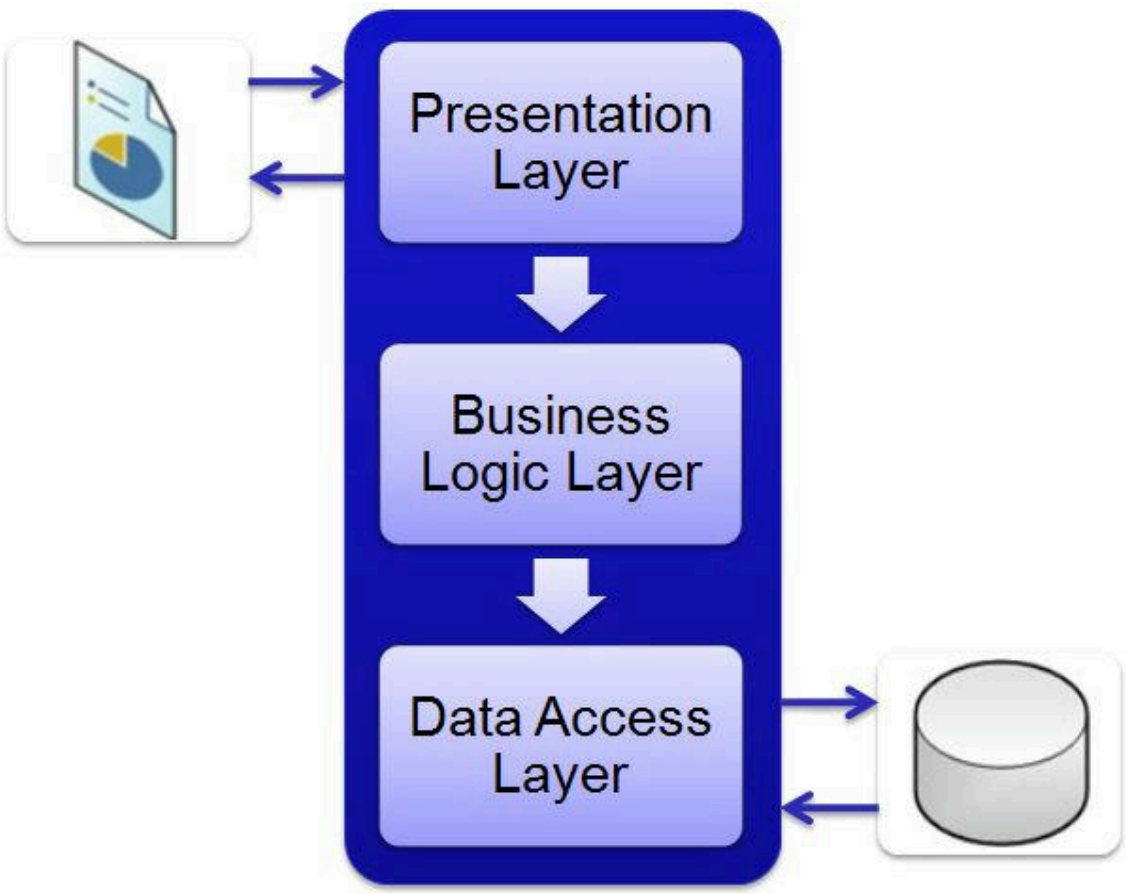



Figure 2: 3-layer architecture

Here is the detail for each layer:

- Presentation Layer: User requests are received and appropriate responses are returned. With Server-Side Rendering (SSR) for simplicity, responses may include data or web pages.
- Business Logic Layer: The business logic of the estate company is handled. This “middle layer” serves as an intermediary, facilitating communication between the user and the database.
- Data Access Layer: The database is queried, and SQL query results are transformed into Java objects.

## 3.2. Entity-Relationship Diagram

Though not exactly within the scope of this project, the classes that hold data are derived from the Entity-Relationship Diagram and must be discussed.

The main entities in the system are:

- User (either with the role of Staff or Manager)
- Building (also referred to as Property)

The following relationships exist for the User entity:

- Role: Users can have many Roles (Manager, Staff, or both), forming a Many-to-Many relationship.
- Building: Users can be assigned to many Buildings, and Buildings can have many Users managing them, also forming a Many-to-Many relationship.

The following relationships exist for the Building entity:

- Building Type: Buildings can have many Types (`NOI_THAT`, `TANG_TRET`, `NGUYEN_CAN`), and many Buildings can share the same Types, forming a Many-to-Many relationship.
- District: A Building can only be in one District, and a District can contain many Buildings, forming a Many-to-One relationship.
- Rent Area: A Building can have many Rent Areas, forming a Many-to-One relationship.

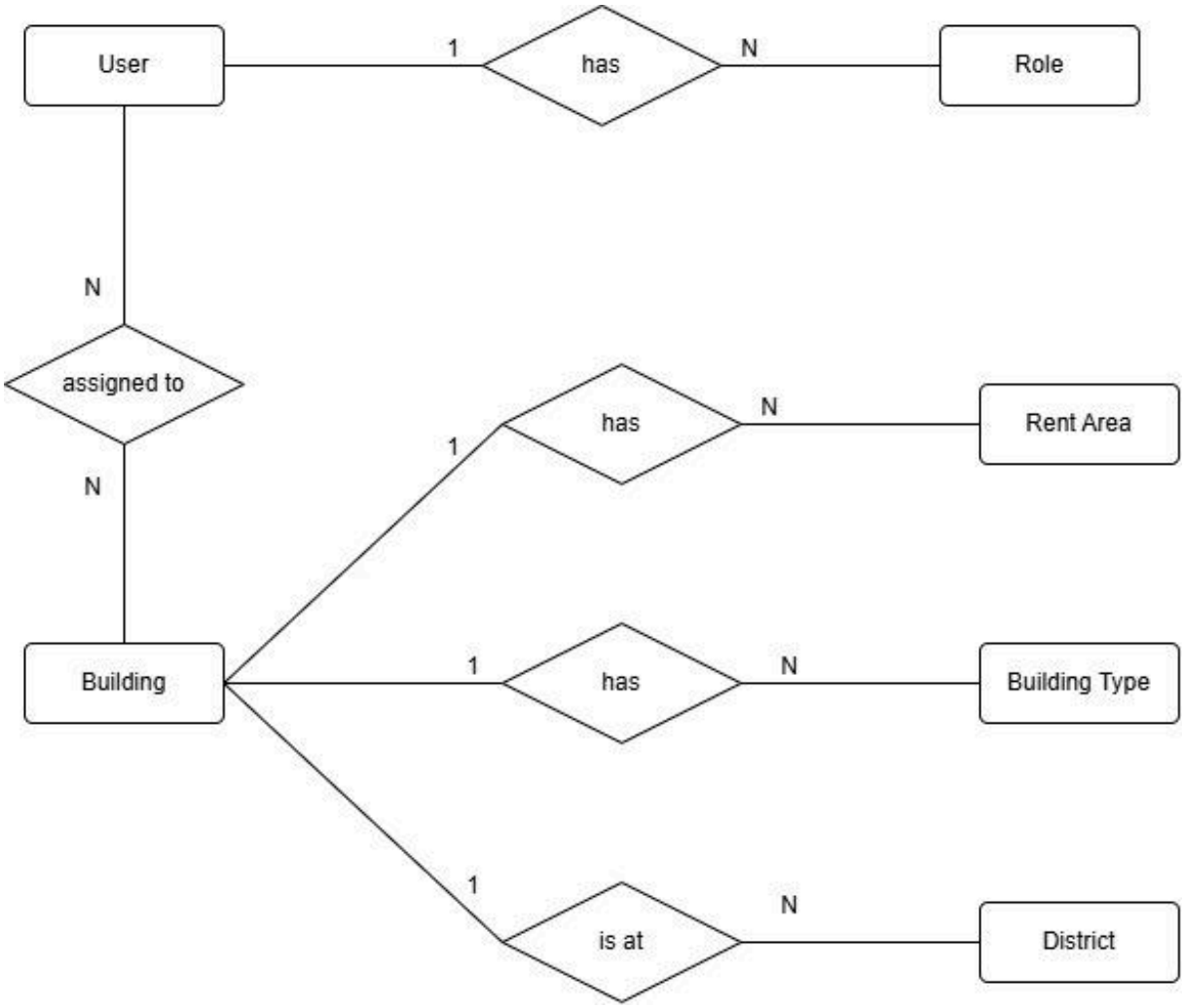



Figure 3: Entity-Relationship Diagram

# 4. Class Design

## 4.1. Application Flow

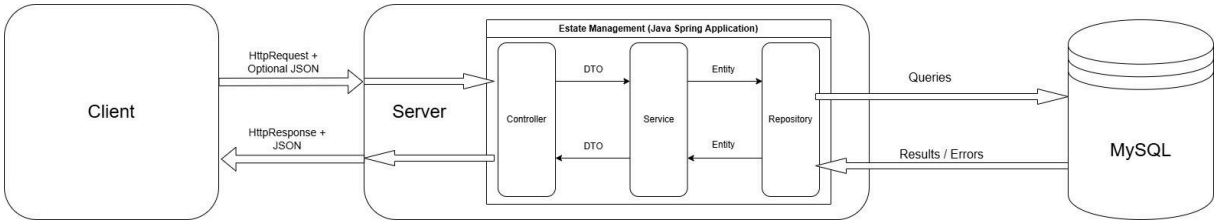



Figure 4: Application flow

Presentation Layer = Controller, Business Logic Layer = Service, Data Access Layer = Repository

The application begins with a user making a request, where the request data is formatted as JSON and sent to the server. The server then routes the request to the appropriate Request Handler in the Presentation Layer. The JSON data is mapped to a Data Transfer Object (DTO) for further processing in the Business Logic Layer.

**Request Handling and Processing**

In the Business Logic Layer, various operations are performed on the DTO to create a new DTO, which will be returned to the Presentation Layer for response preparation.

**Data Retrieval**

When a user requests data from the database, the Business Logic Layer interacts with the Data Access Layer to fetch the data as Entity objects. These Entities, which mirror the structure of the corresponding SQL tables, are then converted to DTOs by the Business Logic Layer. The converted DTOs are sent to the Presentation Layer, which transforms them back into JSON format for the user response.

**Data Persistence**

When users submit data to be persisted, the Business Logic Layer converts the DTO to an Entity and sends it to the Data Access Layer. This layer performs SQL queries to persist the Entity in the database.

**Data Access Layer Responsibilities**

The Data Access Layer connects to the MySQL database and executes SQL queries using the Entity objects. It does not handle any conversion between Entities and DTOs; it strictly receives and processes the converted Entity objects provided by the Business Logic Layer.

This structured flow ensures that each layer of the application has a clear and defined role, enhancing maintainability and scalability.

## 4.2. Data

### 4.2.1. Entity

This part will discuss the creation of the Entity classes

First, the Entity-Relationship Diagram is converted to a database schema

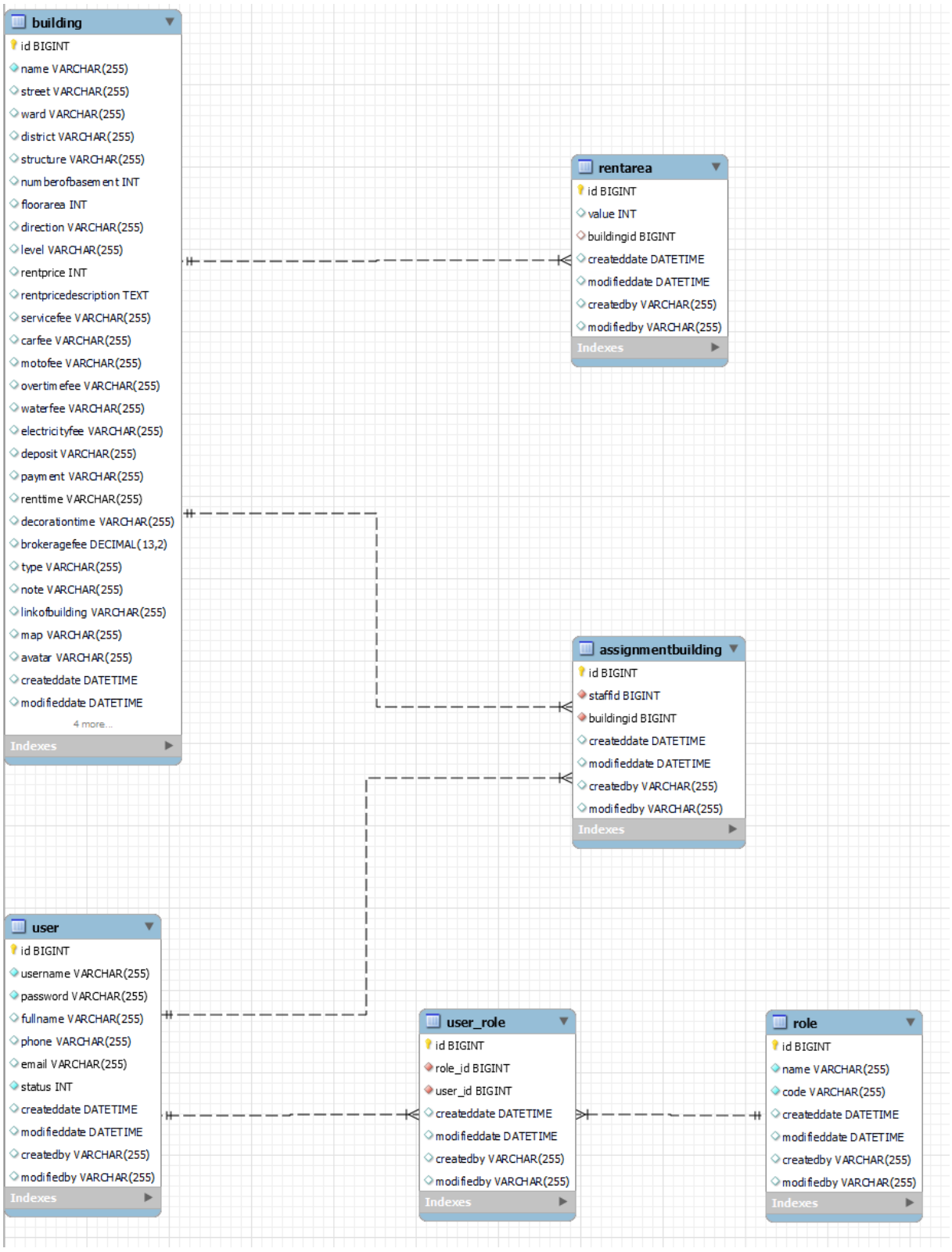



Figure 5: Schema

Then, from this schema, the Entity classes are created

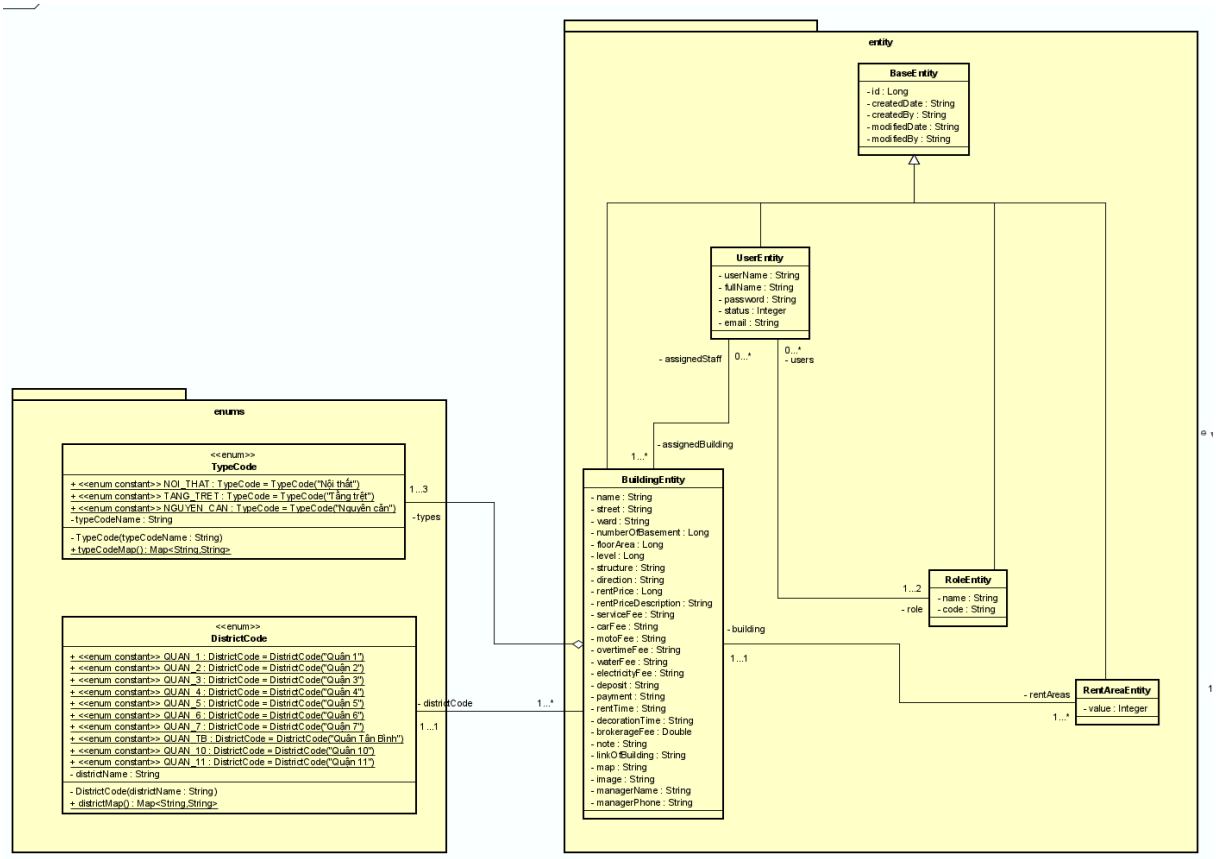



Figure 6: Entities

The Entity classes are organized within the `entity` package. For building types and districts, we have chosen to represent them as Java Enumerations within the `enums` package, named `TypeCode` and `DistrictCode` respectively. This decision is based on the fact that both district and building type classifications are closely tied to legal requirements and are unlikely to change frequently, thus negating the need for database persistence.

All database tables include audit columns. Consequently, in our class design, the entity classes inherit from a `BaseEntity` class. This `BaseEntity` class contains an ID field and the audit properties necessary for tracking changes at the row level within the database.

### 4.2.2. Data Transfer Object Modeling

DTOs are essentially the data exchanged between the user and the system. To accurately model DTOs, we should reference the Use Cases to identify which DTO(s) are needed for each scenario.

User Use Cases:

- Change User Info:
  - Change Full Name: `UserDTO`
  - Change Password: `PasswordDTO`
  - Search Manager/Staff Accounts: `UserDTO`
- Edit Accounts:
  - Create Accounts: `UserDTO`
  - Delete Staff Accounts: No DTO needed
  - Change Staff Accounts: `UserDTO`
- User Session Management:

- Login: `MyUserDetail` (extends the User class provided by the Spring Framework)
- Logout: No DTO needed

Building Use Cases:

- Search Buildings: `BuildingSearchRequest`, `BuildingSearchResponse`
- Assign Staffs to Buildings: `AssignmentBuildingDTO`
- Edit Buildings:
  - Add/Edit Buildings: `BuildingDTO`
  - Delete Building(s): No DTO needed

Similar to the Entities, DTOs requiring auditing inherit from `AbstractDTO` to include auditing properties. Additionally, DTOs that need pagination inherit from this base class to include pagination properties (items per page, page number, etc.).

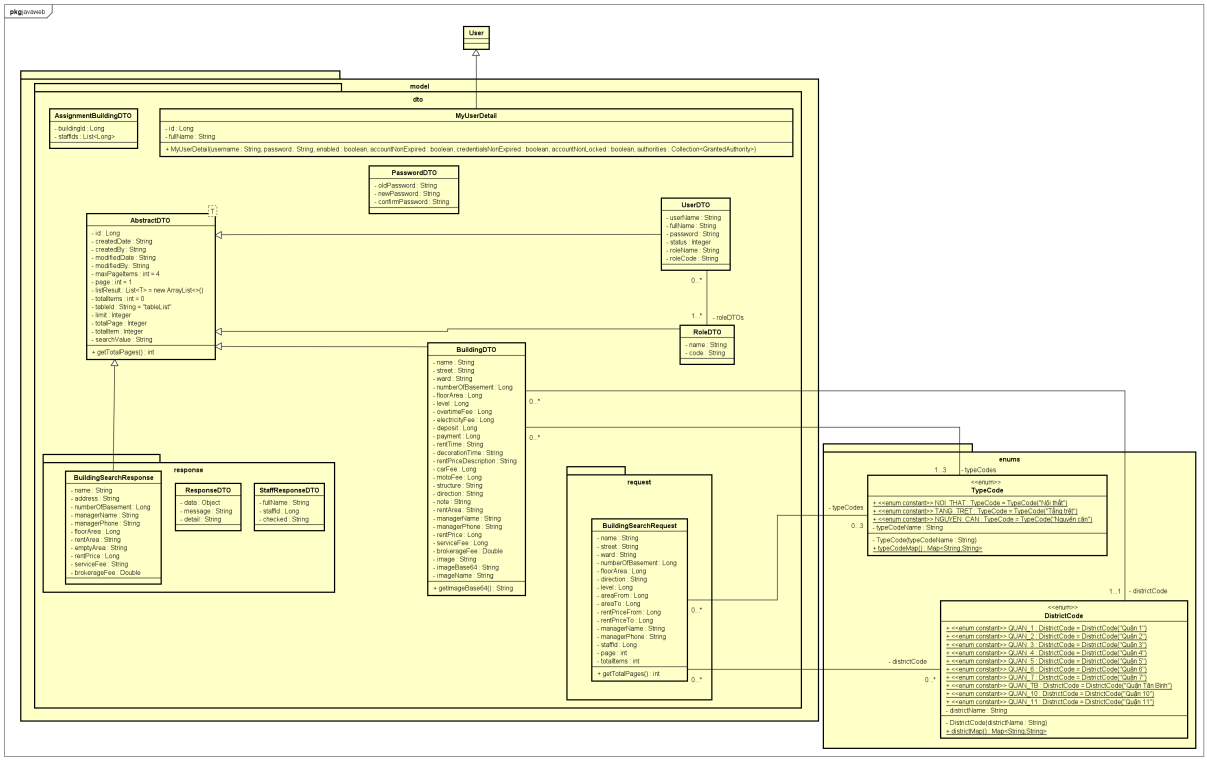
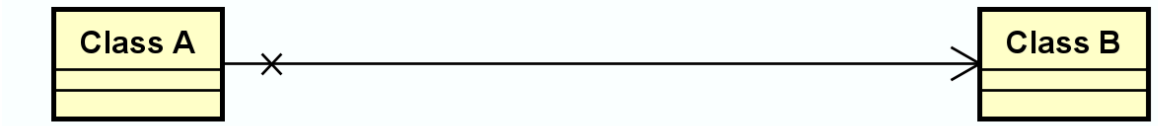



Figure 7: DTOs

## 4.3. Three-Layer Implementation

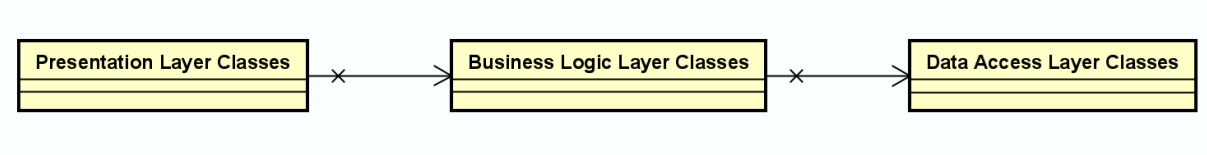
This part discusses the implementation of the 3-layer architecture in Java.

Note: In the general class diagrams, between layers, there are relationship lines that have `X` on one end and an arrow at the other end.



This means “Class A is dependent on Class B, Class B is independent of Class A and CANNOT use Class A”

This is to enforce one-way (unidirectional) dependencies: Presentation depends on Business Logic, Business Logic depends on Data Access, but not the other way around




### 4.3.1. Presentation Layer

First is the `api` package. The classes in this package only returns DTOs. In other words, they only return data, no web pages.

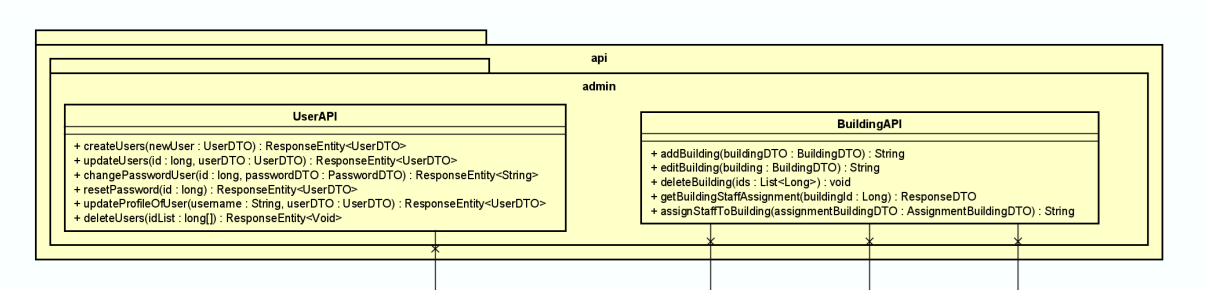



Figure 8: `api` package

Second, the `controller` package. The classes in this package returns web pages and if needed, along with data. The `ModelAndView` class is a special class to manipulate web pages in the Java Spring Framework.

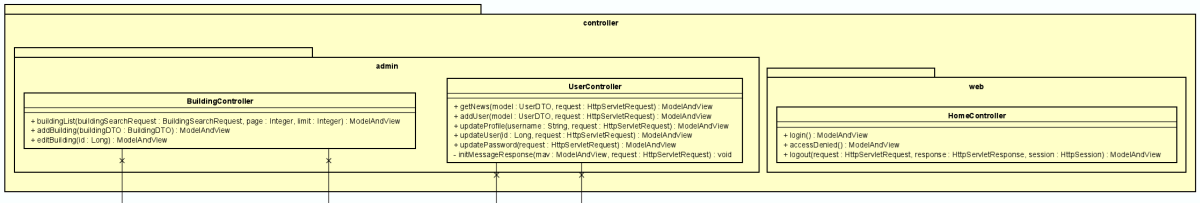



Figure 9: `controller` package

The classes in `api` and `controller` depends on the classes in the `service` classes to fetch data.

### 4.3.2. Business Logic Layer

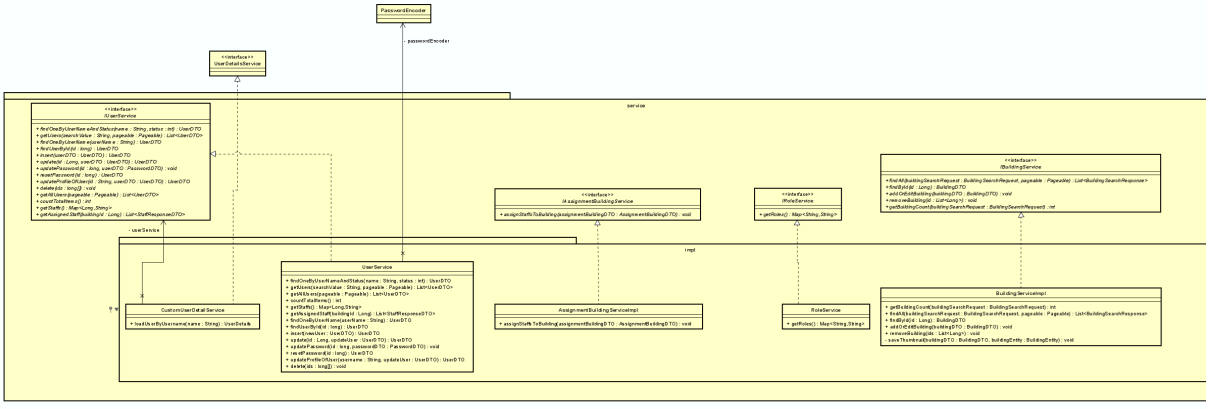



Figure 10: `service` package

Interfaces are used so that the Presentation Layer depends on the interfaces instead of concrete implementations. The actual implementations are organized in the `impl` sub-package, as shown in Figure 10.

### 4.3.3. Data Access Layer

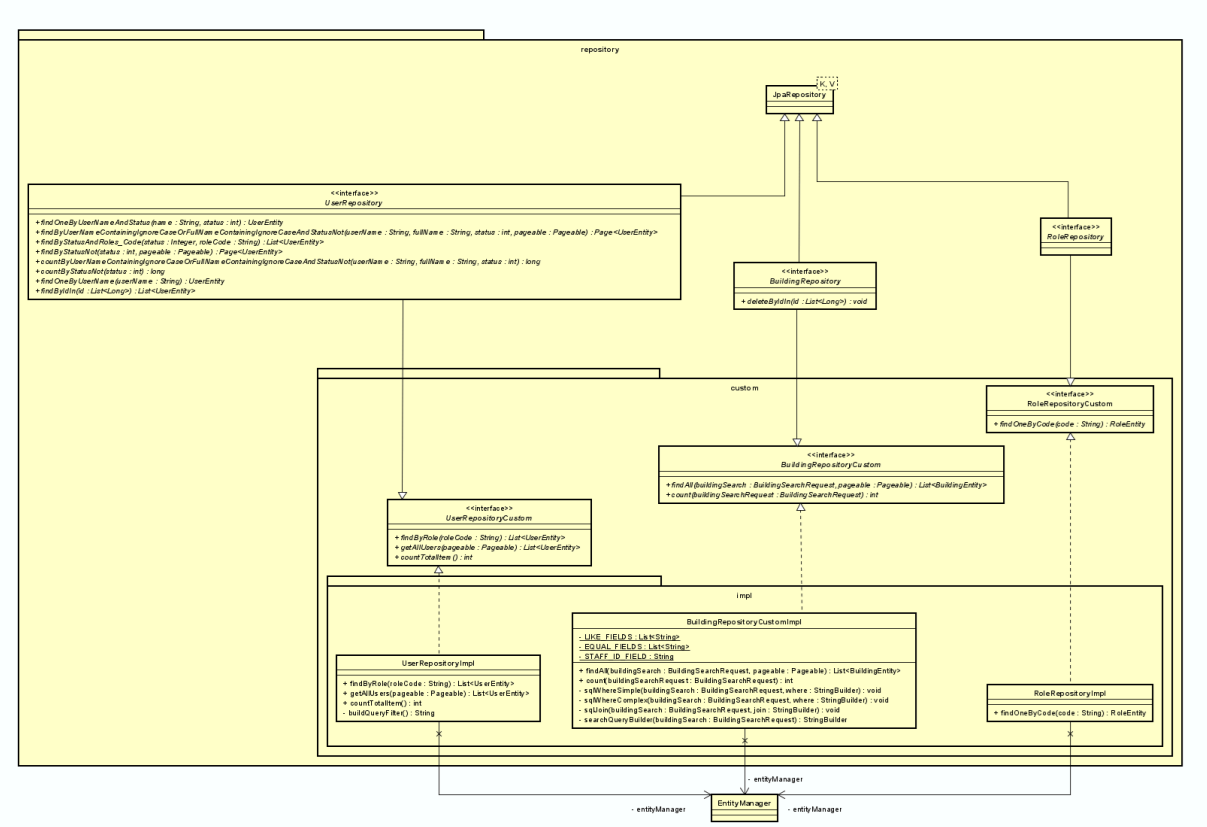



Figure 11: `repository` package

Interfaces are used to conceal the implementation from the classes in the `service` package, similar to how the `service` package conceals the implementation from the `controller` and `api` packages.

There is no `impl` package because the interfaces extend `JpaRepository`, an interface provided by Spring Data JPA that generates SQL queries based on method signatures. This allows developers to focus on application logic rather than string concatenation to build SQL queries in Java, which can be arduous.

However, for highly complex queries, developers still need to write the SQL queries themselves. The interfaces for these queries are organized in the `custom` sub-package. The implementation of these custom interfaces is placed in the `impl` sub-package. These implementations rely on the `EntityManager` class provided by the framework to handle database connections, execute the queries, obtain the results, and map the results to the Entity classes.

### 4.3.4. Others

This section discusses the classes that help the 3 layers function properly.

#### 4.3.4.1. Converters

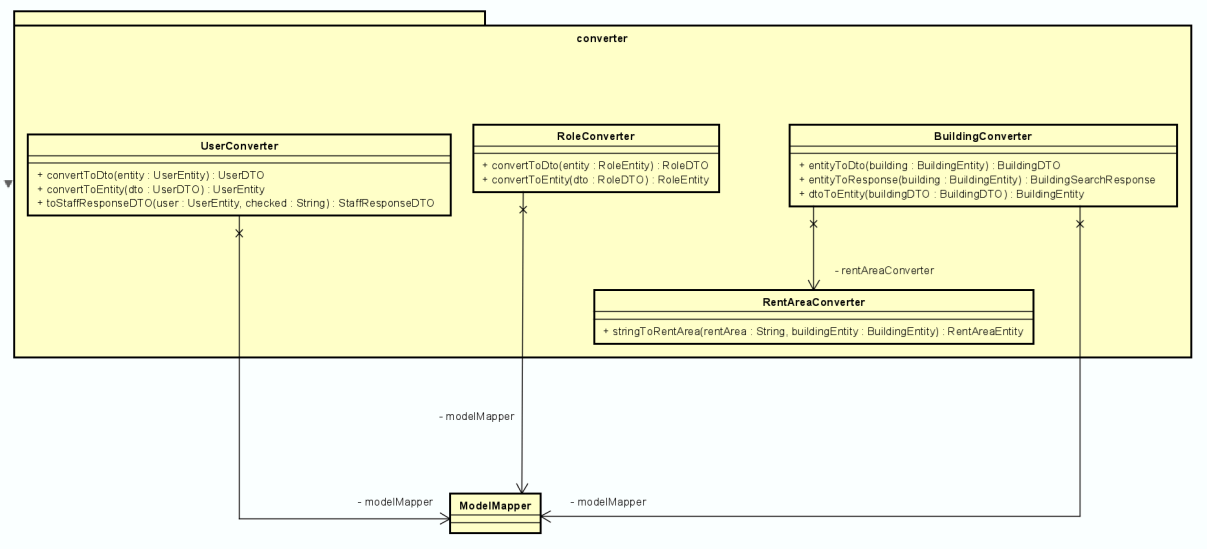



Figure 12: `converter` package

The classes in the `converter` package facilitate Entity-DTO and DTO-Entity conversion. They are utilized by the classes in the `service` package. Dedicating classes specifically for conversion simplifies debugging of the conversion processes and keeps the code in the `service` package much cleaner. The converter classes use `ModelMapper`, which assists in mapping values between objects with similar structures, eliminating the need to manually use getters and setters to transfer values between objects.

#### 4.3.4.2. Configurations

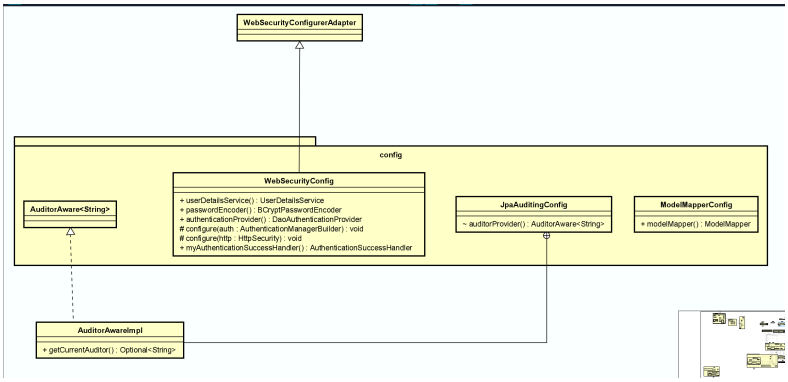



Figure 13: `config` package

The classes in the `config` package initialize dependencies used by other classes. `JpaAuditingConfig` assists with populating audit fields of DTOs and Entities. `ModelMapperConfig` defines custom mappings and constructs the `ModelMapper` object. `WebSecurityConfig` specifies the tools needed for security, such as `passwordEncoder`, `userDetailsService`, and others.

#### 4.3.4.3. Security

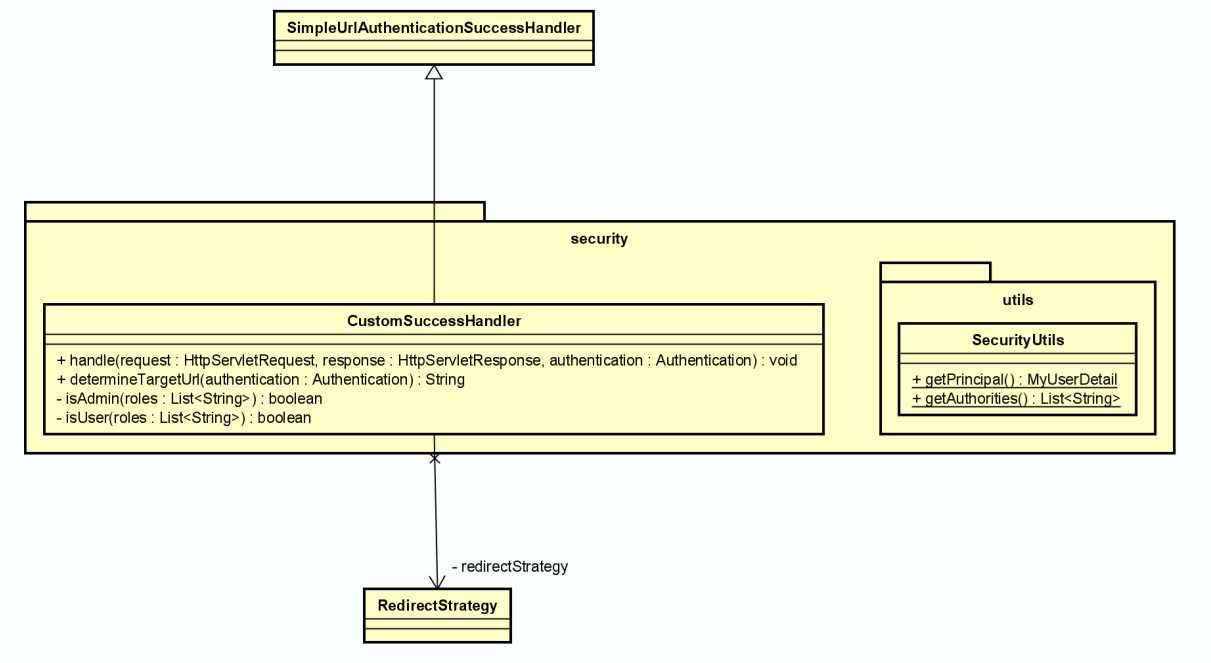



Figure 14: `security` package

The `CustomSuccessHandler` class and the `SecurityUtils` class are used to handle authentication and authorization.

`CustomSuccessHandler`: This class, which extends `SimpleUrlAuthenticationSuccessHandler` from Spring Security, is utilized to manage actions following a user's successful login.

`SecurityUtils`: This utility class provides methods for getting the details of the currently authenticated user. The `getPrincipal` method is used to return the authenticated user's details, and the `getAuthorities` method is employed to return a list of the user's authorities (roles).

#### 4.3.4.4. Exception Handler

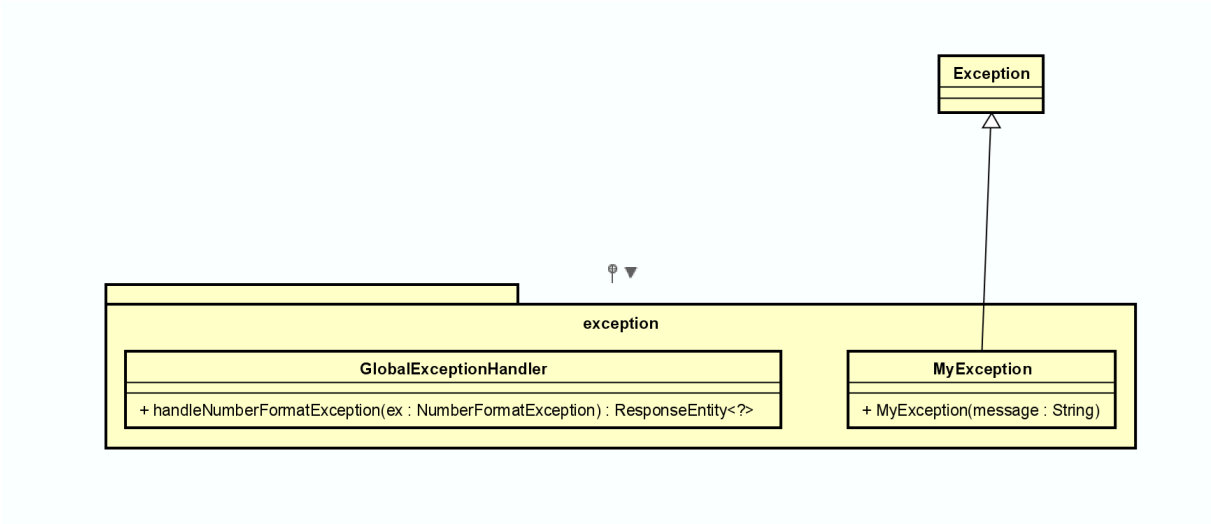



Figure 15: `exception` package

This package contain custom exception the class `MyException` and the exception handler `GlobalExceptionHandler`.

#### 4.3.4.5. Utilities

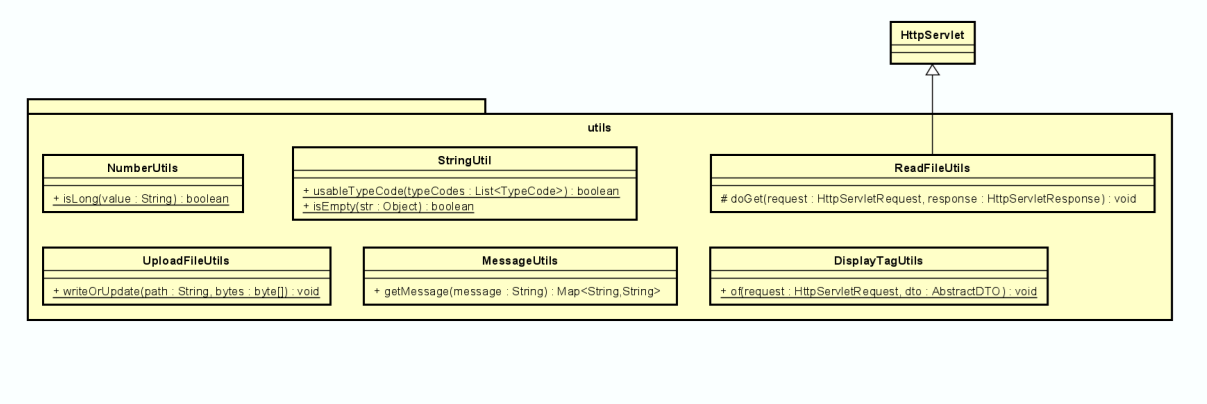



Figure 16: `utils` package

These are the classes that only hold static methods, dealing with small tasks that are repetitive such as number checking, string validation, read/write files, etc.

# 5. Team Member Assignments

- Nguyen Huu Hoang Hai Anh: Use Case Diagram, Enums Class Diagram, Presentation Layer Diagram and their Java code.
- Tran Anh Dung: Business Logic Layer Diagram, DTO Class Diagram and their Java code, also organize team members' code, writing the report and user manual
- Nguyen Dinh An: Database schema, Data Access Layer Diagram, Entity Diagram and their Java code.
- Le Ngoc Quang Hung: Configuration Diagram, Exception Diagram, Utility Diagram, Converter Diagram, Web Security Diagram and their code.