Faculty of Computer Science
Artificial Intelligence and Data Science Department

Hotel Management System - Project Report

Prepared by:

- Akkouchi Nesrine 232331624020
- Ben Zahia Malak 222237349704
- Tebani Hiba 232333159220
- Delhoum Lina Fatmazohra 232331531110

3eme annee ING IA

2025/2026

1. Introduction

The *Hotel Management System* is a Java-based software application developed as part of the **Software Engineering (GL)** module. The main objective of this project is to design and implement a structured, modular, and extensible application that strictly follows the **Model–View–Controller (MVC)** architectural pattern and integrates several **object-oriented design patterns**.

The system simulates real hotel operations such as room management, reservations, pricing, messaging, and administrative monitoring. It serves both as a practical application and as an educational example demonstrating how good software design principles can simplify complex systems.

2. Motivation

The **problematic** that guided our choice of project is the following:

> *How can we design a modular and extensible system that manages hotel operations (authentication, room management, reservations, communication, and statistics) while ensuring clear separation of responsibilities and applying design patterns to improve maintainability and scalability?*

We selected this idea because hotel management is a **real-world problem** that requires:

- Centralized management of **users and roles** (clients vs administrators).
- Efficient handling of **rooms and reservations**, including filtering and availability.
- Flexible **pricing strategies** depending on conditions (normal vs discount).
- Real-time **notifications and communication** between clients and administrators.

This domain naturally lends itself to the **MVC model** and the required **design patterns**, making it an ideal case study to demonstrate our ability to apply theoretical concepts to a practical, realistic scenario.

3. Project Objectives

The main objectives of this project are:

- Apply the MVC architecture to ensure a clear separation of concerns
- Implement multiple design patterns in a coherent and structured way
- Develop a functional Java application using object-oriented programming principles
- Design an intuitive **Graphical User Interface (GUI)** using JavaFX
- Ensure data persistence using file-based storage
- Deliver clean, well-documented, and maintainable code

4. General Description of the Application

The Hotel Management System is a Java-based application that allows users to interact with hotel services through a **Graphical User Interface (GUI)** built using JavaFX.

The application strictly follows the Model–View–Controller (MVC) architecture, where:

- Models represent business data
- Controllers manage application logic
- Views handle user interaction through graphical components

The system supports the following functionalities:

- User authentication and role-based access
- Room consultation and filtering
- Reservation creation, confirmation, cancellation, and completion
- Dynamic pricing calculation
- Messaging
- Administrative statistics and monitoring

This GUI-only approach improves usability and provides a modern and user-friendly experience.

5. Functional Requirements

5.1 User Management

- User authentication using email and password
- Role-based access control
- Secure session handling

5.2 Room Management

- Storage of room details (type, price, status, features)
- Automatic update of room availability
- Support for individual rooms and room groups

5.3 Filtering

- Filtering by room type, price, availability, and features
- Combination of multiple filters

5.4 Reservation Management

- Reservation creation with date validation
- Reservation confirmation, cancellation, and completion
- Conflict prevention and availability checks

5.5 Pricing

- Dynamic pricing calculation
- Support for discounts and optional services

5.6 Messaging and Administration

- Messaging system
- Notifications
- Viewing system statistics

6. User Interface Design: Graphical User Interface (GUI)

6.1 Graphical User Interface Overview

The application uses a **fully graphical interface** implemented with JavaFX.
All user interactions are handled through windows, forms, buttons, and dialog boxes.

The GUI:

- Provides an intuitive and modern user experience
- Uses reusable UI components for consistency
- Communicates exclusively with controllers
- Contains no business logic

This design fully respects MVC principles by isolating presentation from application logic.

7. Project Constraints

7.1 Architectural Constraints

- Strict adherence to the MVC architecture
- No business logic inside GUI components
- Controllers act as intermediaries between models and views

7.2 Technical Constraints

- Programming language: Java
- Graphical interface: JavaFX
- Data storage: JSON files
- No external frameworks required

7.3 Academic Constraints

- Mandatory use of Singleton, Strategy, Composite, and Observer patterns
- Clear documentation
- Fully functional GUI-based application

8. System Architecture

```
src/
 ├── model/
 ├── controller/
 ├── view/
 ├── data/
 └── Main/
```

This structure ensures modularity, low coupling, and ease of maintenance.

9. Data Management

All application data is stored in JSON files. A centralized **DataManager**, implemented as a **Singleton**, is responsible for loading and saving data. Gson is used to serialize and deserialize Java objects.

This approach ensures consistent data access and persistence.

10. Specification of Design Patterns

10.1 Singleton Pattern

Used for the DataManager to ensure a single point of access to persistent data.

Benefits:

- Prevents inconsistent data access
- Centralizes file operations

10.2 Strategy Pattern

Used for:

- Room filtering
- Pricing calculation

Each strategy encapsulates a specific algorithm and can be selected at runtime.

Benefits:

- High flexibility
- Easy extension
- Clean separation of logic

10.3 Composite Pattern

Used to represent individual rooms and groups of rooms using a shared interface.

Benefits:

- Uniform handling
- Simplified room management

10.4 Observer Pattern

Used for messaging and notifications.

Benefits:

- Loose coupling
- Automatic updates

11. User Management and Authentication System

11.1 User Model

User information is represented by the UserModel class, which encapsulates all data related to application users. Each user is defined by the following attributes:

- **id**: a unique identifier generated using a UUID
- **firstName** and **lastName**: personal identification information
- **email**: used as a unique login credential
- **password**: used for authentication
- **phone**: contact information
- **role**: defines user permissions (ADMIN or USER)

The model also provides utility methods to check user roles, such as determining whether a user is an administrator or a regular user. This role-based design allows the system to control access to specific functionalities in a clear and extensible way.

The user model strictly belongs to the **model layer** and contains no logic related to user interaction or data persistence.

11.2 Role Management

User roles are defined using an enumeration (Role) with two possible values:

- **ADMIN**
- **USER**

Roles are used to differentiate system behavior and access rights. For example, administrative users may access statistics or management features, while regular users are limited to standard booking operations. This design supports scalability, as new roles can be added easily in the future.

11.3 User Controller – Authentication Logic

The UserController is responsible for handling all user-related operations and acts as the **controller layer** between the user model, persistent storage, and session management.

Its main responsibilities include:

- User registration (sign-up)

- User authentication (sign-in)
- Access to the currently logged-in user

User data is stored in a JSON file (user.json) and managed using the centralized **DataManager**, which ensures consistent reading and writing of user information.

11.4 User Registration (Sign-Up)

During the registration process, the controller performs the following steps:

1. Loads existing users from persistent storage
2. Checks whether the provided email address is already in use
3. Generates a unique user identifier using a UUID
4. Creates a new user with the default role of **USER**
5. Saves the updated user list back to the JSON file

If the email address already exists, the registration process fails, ensuring that each user has a unique login identity.

11.5 User Authentication (Sign-In)

User authentication is handled by verifying the provided email and password against stored user data. When valid credentials are found:

- The user is marked as logged in
- The session is updated using the session management system
- The authenticated user is returned to the application

If authentication fails, no session is created, and access to protected features is denied.

This mechanism ensures controlled and secure access to the system.

11.6 Session Management with the Singleton Pattern

Session handling is managed by the SessionManager, which is implemented using the **Singleton design pattern**. This guarantees that only one session instance exists throughout the application lifecycle.

The session manager is responsible for:

- Storing the currently logged-in user
- Determining whether a user is logged in
- Providing access to the current user

Using a Singleton for session management ensures global consistency and avoids conflicting session states across different parts of the application.

11.7 User Identification

User identifiers are generated using a dedicated utility class that produces universally unique identifiers (UUIDs). This guarantees that every user has a unique and collision-free identifier, independent of their personal information.

This approach improves data integrity and simplifies user referencing across the system.

11.8 Integration with MVC Architecture

The user management system strictly respects MVC principles:

- **Model**: UserModel and Role represent user data and permissions
- **Controller**: UserController manages authentication logic and validation
- **View**: User interaction (console or GUI) is handled separately

This separation ensures maintainability, testability, and flexibility.

11.9 Design Quality and Benefits

The user management and authentication system offers several advantages:

- Clear separation of concerns
- Secure and consistent session handling
- Role-based access control
- Easy extensibility for new roles or authentication features
- Clean integration with other system modules

11.10 User Views

Overview

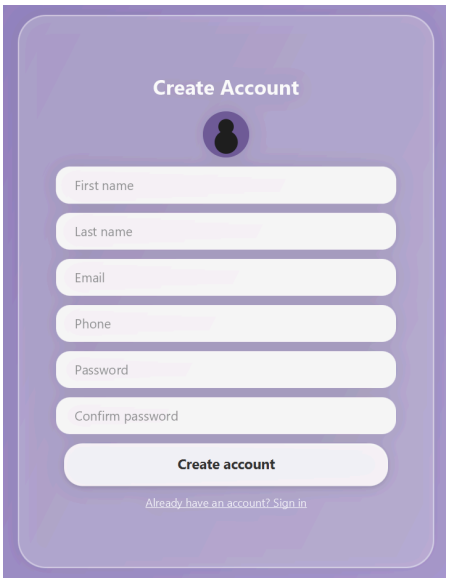
The user interaction layer of the system is entirely based on a **Graphical User Interface (GUI)** developed using JavaFX.
All user actions are performed through graphical components, ensuring a consistent and user-friendly experience.

Graphical User Interface for User Management

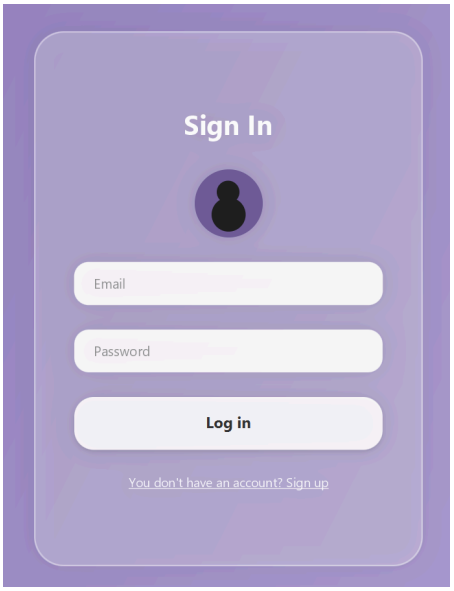
The GUI provides dedicated views for user authentication and navigation, including:

Authentication Views (Sign In / Sign Up)

- User registration (Sign Up)



- User authentication (Sign In)



These views collect user input through structured forms and delegate all validation and authentication logic to the UserController.
Feedback is provided using alert dialogs for:

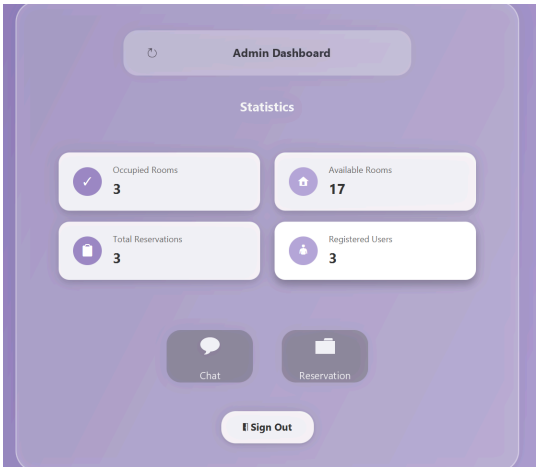
- Invalid input
- Authentication failure
- Successful registration or login

Reusable UI components such as rounded buttons, panels, and custom layouts are used to ensure visual consistency.

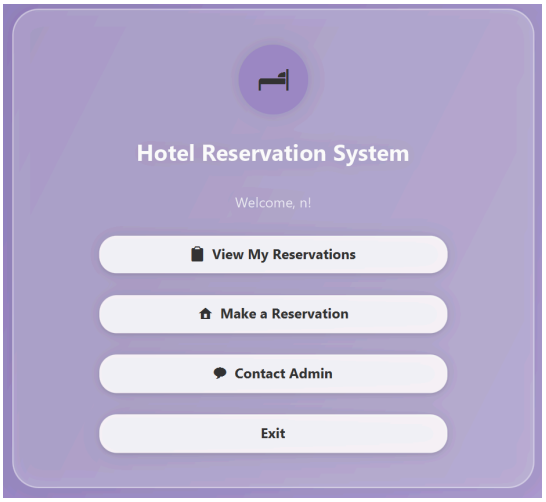
Dashboard Views and Role-Based Access

After authentication, users are redirected to a dashboard based on their role:

- **Administrator Dashboard**



- **Client Dashboard**



Access control is enforced through the session management system. Unauthorized access attempts automatically redirect the user to the sign-in view.

The dashboards act as central navigation points for all system functionalities.

12. Room Management and Filtering

12.1 Room Management

All room data is stored in rooms.json. Each room includes:

- id (unique identifier)
- number (unique room number)
- type (single, double, suite)
- status (available, occupied, etc.)
- price
- features

Data access is managed by the Singleton DataManager using Gson.

The **Composite pattern** allows the system to manage individual rooms and groups uniformly, forming the foundation of the model layer.

12.2 Filtering Rooms

Filtering is implemented using the **Strategy pattern**. A RoomFilter interface defines a common method applied by multiple concrete filters (type, price, features, availability, composite filters).

This design improves modularity, readability, and extensibility.

12.3 Room Controller

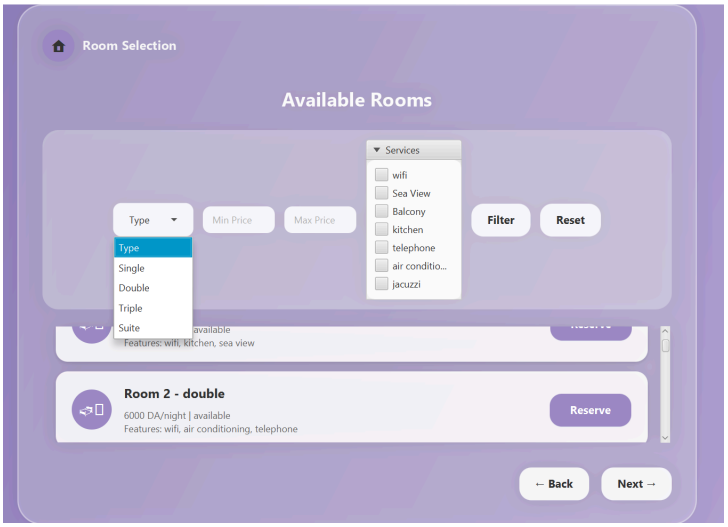
The RoomController manages:

- Loading and saving rooms
- Adding, updating, and deleting rooms
- Searching rooms by ID or number
- Room statistics
- Applying filtering strategies

It ensures consistency and reliability of room data.

12.4 Room View

The RoomView handles user interaction, displays room lists, and guides users through filtering options. It delegates all logic to the controller, ensuring clean separation of concerns.



13. Reservation Management and Pricing System

13.1 Reservation Model

Each reservation includes:

- Reservation ID
- User ID
- Room ID
- Check-in and check-out dates
- Number of nights
- Total price
- Status (active, cancelled, completed)
- Creation date

The model encapsulates behaviors such as activation, cancellation, and completion.

13.2 Reservation Controller

The ReservationController manages:

- Reservation creation and validation
- Room availability checks
- Pricing strategy selection
- Reservation state transitions
- Synchronization with room state
- Persistent storage

13.3 Reservation Validation

Before creating a reservation, the system checks:

- Room existence
- Valid date range
- No overlapping reservations
- Room availability

These validations prevent conflicts and ensure correctness.

13.4 Pricing with the Strategy Pattern

Pricing is implemented using the **Strategy pattern**:

- Normal pricing: base price × number of nights
- Discount pricing: percentage-based reduction

Optional services (restaurant, spa, parking) are calculated separately and added to the total price.

13.5 Reservation Lifecycle

Reservation states:

- Active
- Cancelled
- Completed

Room states are automatically updated based on reservation status.

13.6 Availability and Conflict Prevention

Date overlap checks prevent double booking. Only active reservations are considered when checking availability.

13.7 Reservation Views

Graphical User Interface for Reservations

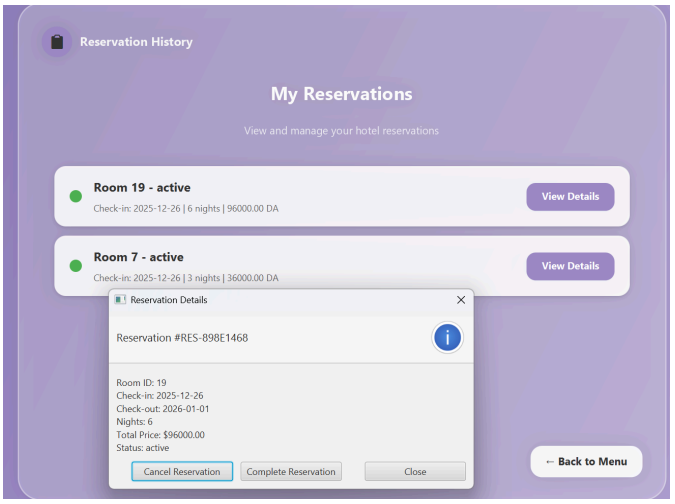
The reservation system is fully managed through graphical views that guide users through the complete booking workflow.

Current Reservations View

This view displays all existing reservations in a structured list, showing:

- Room identifier
- Check-in date

- Number of nights
- Reservation status



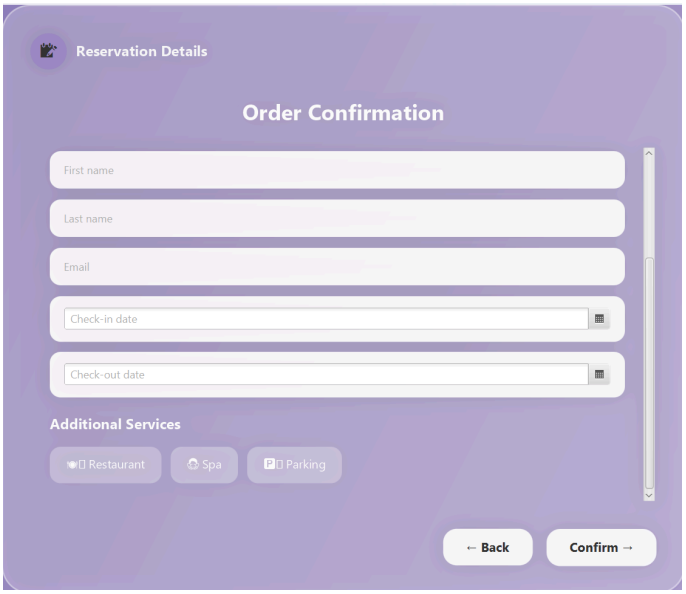
Users can view details, cancel, or complete reservations depending on their status.

Reservation Creation View

The reservation creation interface collects:

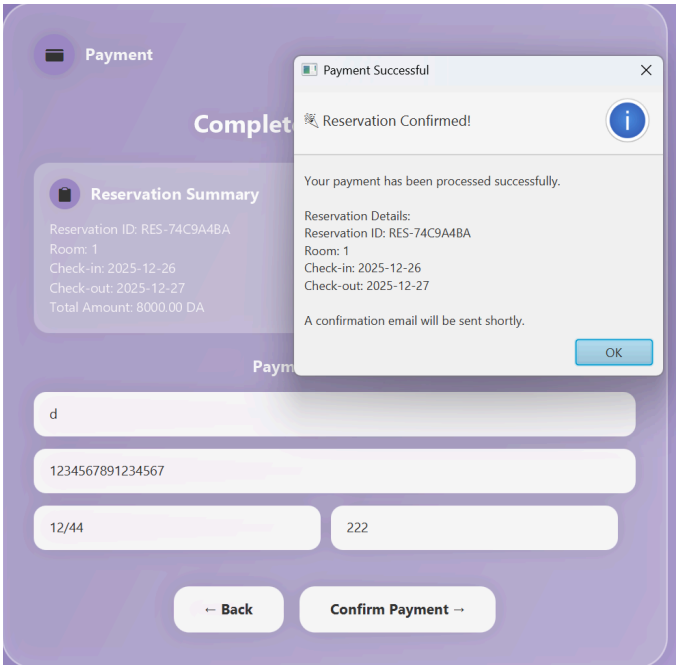
- Check-in and check-out dates
- Selected room
- Optional services (restaurant, spa, parking)

Input validation is performed before submission, and pricing is calculated automatically by the controller.



Payment View

The payment interface displays a reservation summary and collects payment details.
After successful validation, the reservation is finalized and confirmed.



16. Messaging System and Notifications

16.1 Overview

The messaging system provides a communication mechanism between users and the system. It allows users to send structured messages such as complaints, bug reports, or questions. The system is designed to be extensible, reactive, and loosely coupled, using the **Observer design pattern** to keep different components synchronized.

16.2 Message Model

Each message is represented by the MessageModel class, which encapsulates all information related to a single message. A message includes the following attributes:

- **id**: unique identifier of the message
- **username**: sender's display name
- **title**: subject or topic of the message
- **content**: main body of the message
- **date**: message creation date
- **messageType**: category of the message (complaint, bug report, question, etc.)
- **senderId**: identifier of the sender
- **isOpened**: indicates whether the message has been read

The message model strictly belongs to the **model layer** and focuses only on data representation and basic behavior, such as formatting message output.

16.3 Message Service and Data Persistence

The MessageServiceModel acts as the central service responsible for managing messages. It maintains a list of all messages and handles their persistence using the DataManager.

Key responsibilities include:

- Loading messages from persistent storage at startup
- Storing new messages
- Updating observers whenever the message list changes

By centralizing message storage and updates, the system ensures data consistency and reliability.

16.4 Observer Design Pattern Implementation

The messaging system is built around the **Observer design pattern**, enabling automatic notification when the message list changes.

Subject

The MessageServiceModel implements the Subject interface and maintains a list of observers. When a new message is added:

1. The message list is updated
2. Data is saved to persistent storage
3. All registered observers are notified

Observer

The MessageController implements the Observer interface and registers itself with the message service. When notified, it updates its local copy of messages automatically.

This design ensures:

- Loose coupling between components
- Real-time synchronization of message data
- Easy extension to additional observers (e.g., GUI inbox view, admin dashboard)

16.5 Message Controller

The MessageController acts as the controller layer between the message model, service, and views. Its responsibilities include:

- Sending messages
- Receiving updates from the message service
- Providing access to the current message list
- Counting messages for statistics or notifications

When a message is sent, the controller delegates storage and notification to the message service, ensuring separation of concerns and clean architecture.

16.6 Graphical User Interface for Messaging

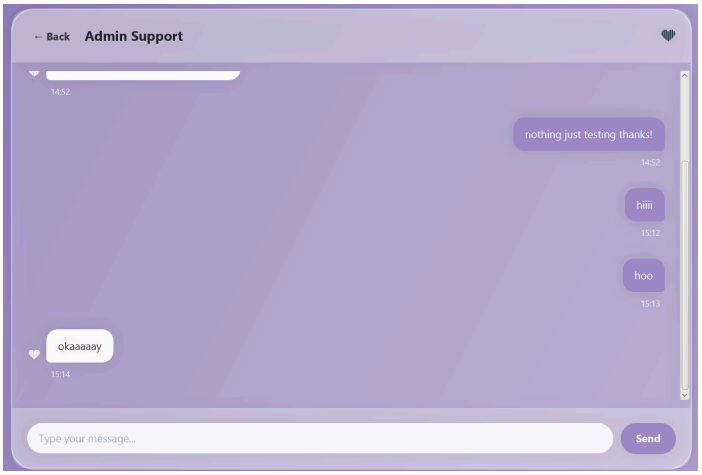
The messaging system is entirely GUI-based and includes three main views:

Client Messaging View (Contact Admin)

The ContactAdminView provides users with a **chat-style interface** to communicate with administrators.
Features include:

- Display of message history
- Real-time message updates
- Visual distinction between user and admin messages
- Automatic scrolling and timestamps

Users can type and send messages using an input field, and each message is immediately saved and displayed.

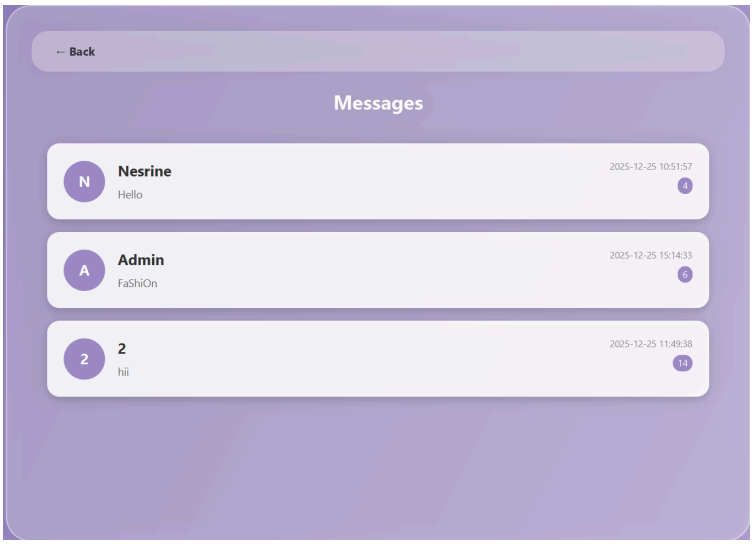


Admin Messages Overview

The AdminMessagesView allows administrators to view all user conversations in a structured format.
Messages are grouped by sender, and each conversation card displays:

- User name
- Message preview
- Date of last message
- Total number of messages

This view enables administrators to quickly identify and access ongoing conversations.

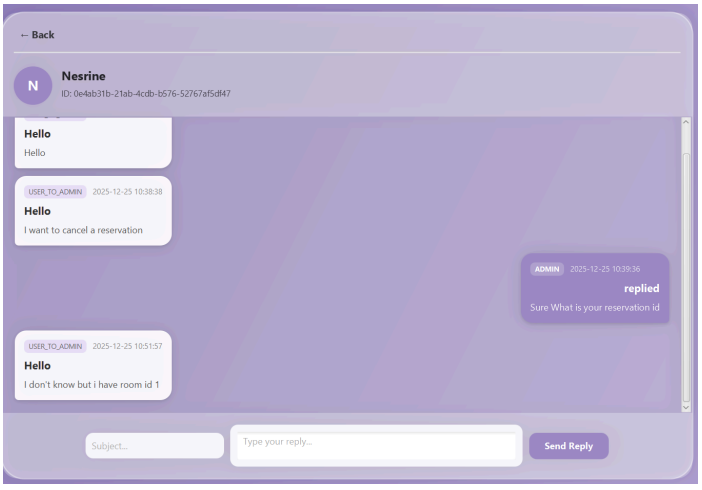


Conversation View (Admin Side)

The ConversationView displays the full conversation between an administrator and a specific user.
It supports:

- Chronological display of messages
- Visual separation between user and admin messages
- Reply functionality directly from the interface

Replies sent by administrators are stored with the user's identifier, ensuring they appear correctly in the client's conversation.



17. Statistics and Monitoring Module

17.1 Overview

The Statistics and Monitoring module provides administrators with a **global and structured view of the hotel system's data**.
It enables consultation and management of:

- Registered users
- Hotel rooms (available, occupied, maintenance)
- Reservations and their statuses

The module is implemented following the **MVC (Model–View–Controller)** architecture.
All statistical data is retrieved dynamically from the **Model and Service layers via Controllers**, ensuring data consistency and separation of concerns.

The module focuses on **operational statistics and monitoring**, not predictive analytics or charts.

17.2 Architectural Design

17.2.1 Model and Service Layer Responsibilities

The statistics module relies on the following services:

- UserService
- RoomService
- ReservationService

These services are responsible for:

- Loading data from persistent JSON storage using DataManager
- Performing filtering, counting, and state validation
- Ensuring fresh data by reloading files before returning results

17.2.2 Controller Layer Responsibilities

The following controllers act as intermediaries between the GUI and the services:

- UserController
- RoomController
- ReservationController

Controllers:

- Expose read-only and deletion operations for statistics

- Delegate all business logic to services
- Prevent direct access to data files from the UI

The statistics views communicate **only with controllers**, never directly with models or services.

17.3 User Statistics

17.3.1 Registered Users List

The system provides administrators with a complete list of all registered users.

Data Source

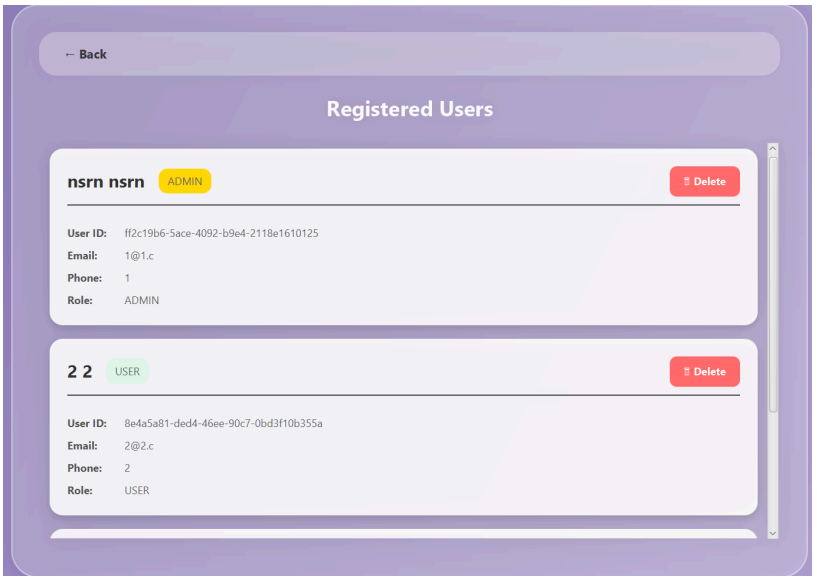
- UserService.getAllUsers()
- Always reloads user data from user.json to ensure accuracy

Access Path

UserController → UserService → DataManager

Displayed Information

- User ID
- First name and last name
- Email address
- Phone number
- User role (ADMIN or USER)



17.3.2 User Count Statistics

The following numerical statistics are available:

- Total number of users
- Number of administrators
- Number of regular users

17.3.3 User Deletion Monitoring

Administrators may delete users directly from the statistics interface.

17.4 Room Statistics

17.4.1 Global Room Statistics

The system provides quantitative statistics about hotel rooms:

- Total number of rooms
- Number of available rooms
- Number of occupied rooms
- Number of rooms under maintenance

17.4.2 Available Rooms View

This view displays all rooms whose state is marked as **available**.

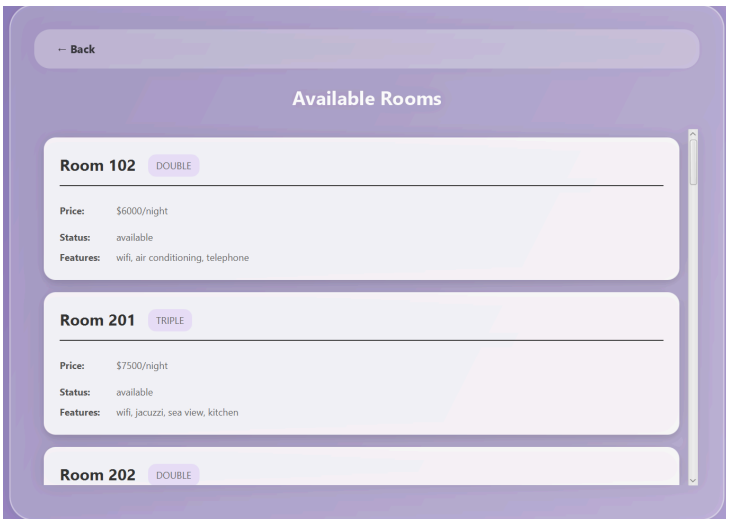
Logic

- Rooms are filtered by state using RoomService.filterByState("available")
- Only rooms with state "available" are displayed

Displayed Information

- Room ID and room number
- Room type
- Price per night
- Current state

This view is informational and does not modify room data.



17.4.3 Occupied Rooms View

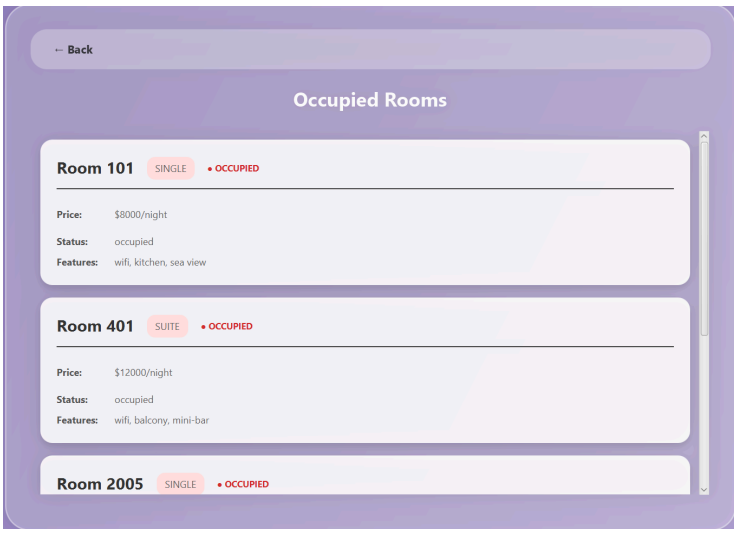
This view displays all rooms currently in use.

Logic

- Rooms are filtered by state "occupied"
- State changes occur automatically through reservation confirmation

Room State Synchronization
Room states are updated automatically when:

- A reservation is confirmed → state becomes occupied
- A reservation is completed or cancelled → state becomes available

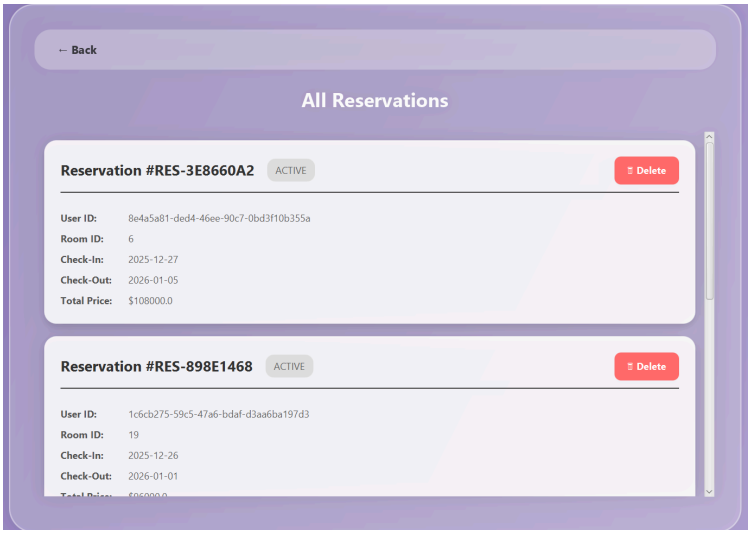


17.5 Reservation Statistics

17.5.1 All Reservations View

Displayed Information

- Reservation ID
- User ID
- Room ID
- Check-in date
- Check-out date
- Total price
- Reservation status (active, completed, cancelled)



17.5.2 Reservation Status Monitoring

Reservation status changes are reflected automatically in statistics:

- **Active** → confirmed and room occupied
- **Completed** → stay finished, room released
- **Cancelled** → reservation cancelled, room released

Status transitions trigger **room state updates**, ensuring consistency between room and reservation statistics.

17.5.3 Reservation Deletion

Administrators may delete reservations from the statistics interface.

17.6 User Interface Integration

The statistics module is implemented using **JavaFX** and includes:

- Scrollable card-based layouts
- Clear section titles
- Color indicators for room and reservation states

- Confirmation dialogs for destructive actions

The interface is read-only except for controlled delete operations.

18. Integration and Data Flow

1. User authentication
2. Room consultation and filtering
3. Reservation creation
4. Pricing calculation
5. Data persistence
6. Messaging and notifications
7. Administrative monitoring

All interactions respect MVC principles.

19. Class Diagram

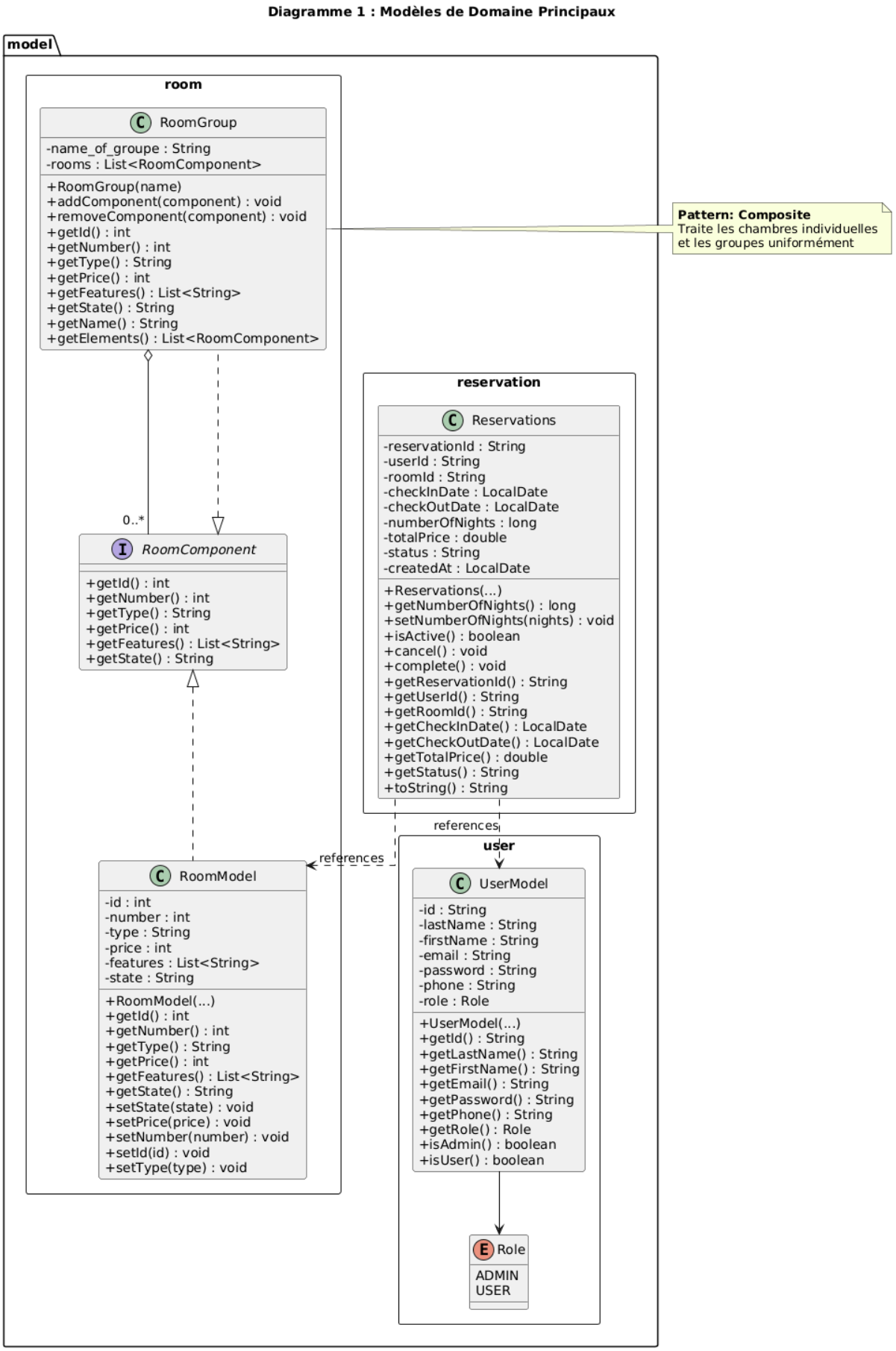


Diagramme 2 : Stratégies de Prix et Filtres

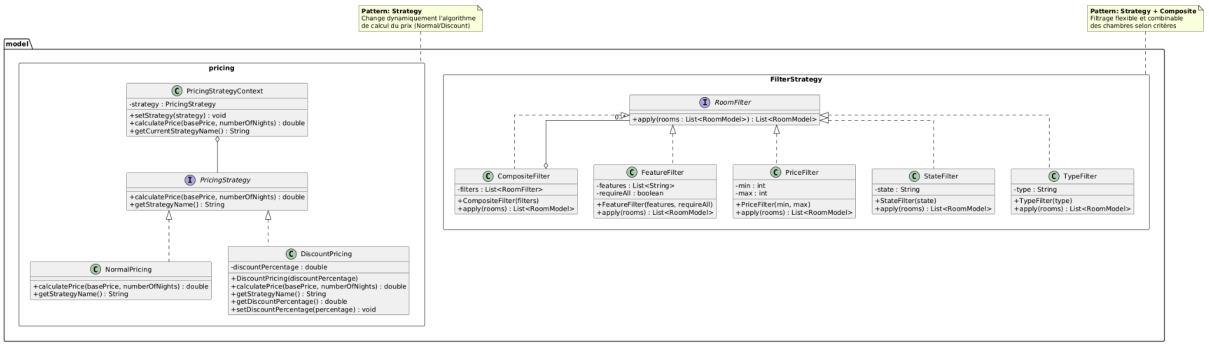


Diagramme 3 : Système de Messages avec Pattern Observer

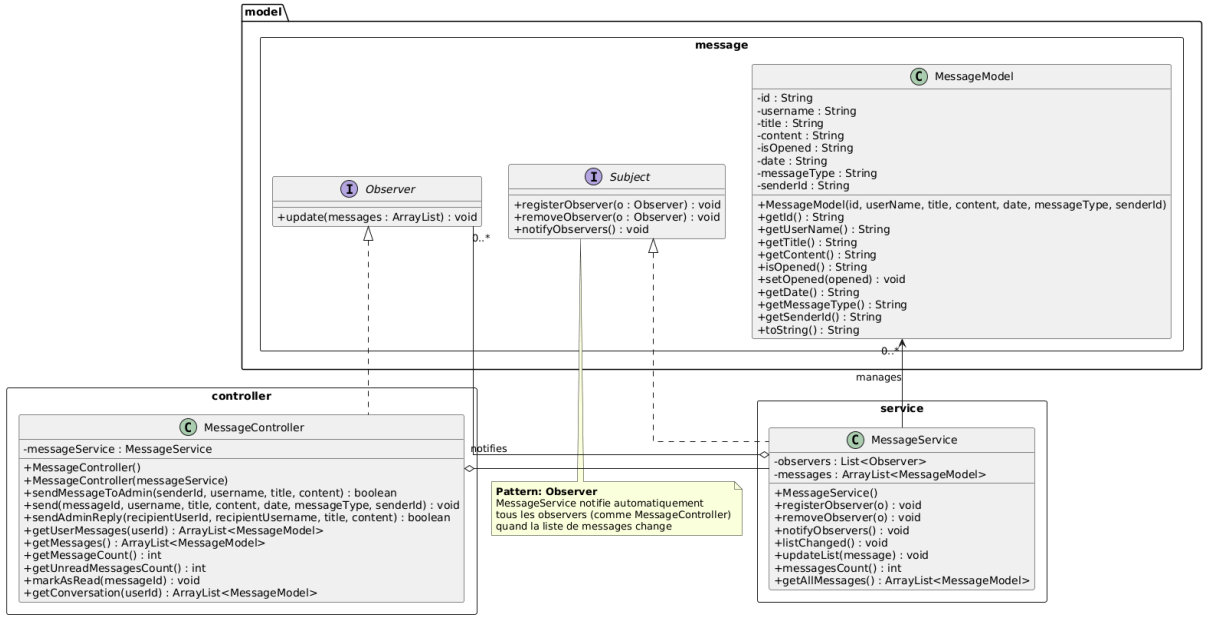


Diagramme 4 : Couches Service et Controller

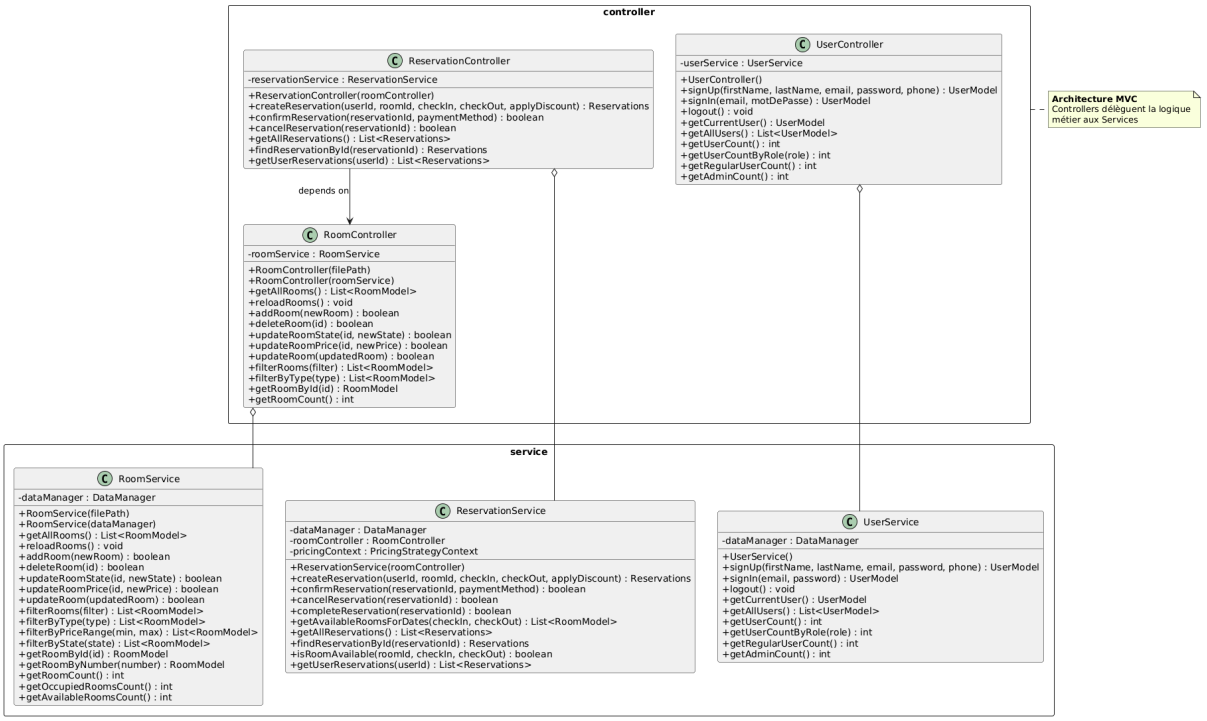
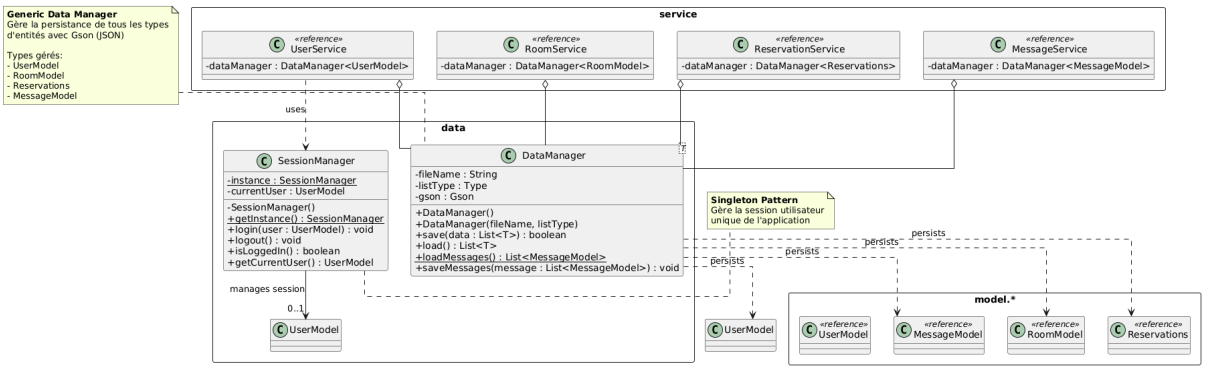


Diagramme 5 : Couche Data - Persistance et Gestion de Session



20. Conclusion

The Hotel Management System successfully demonstrates the application of MVC architecture and multiple design patterns in a **GUI-based Java application**.
By relying exclusively on JavaFX for user interaction, the system provides an intuitive, modern, and maintainable interface while preserving a clean separation of concerns.

The project fulfills all academic and technical requirements and serves as a strong example of well-structured software engineering design.