Software Architecture Design Report

Submission 2 SAD

NüRoom
SWEN90007 SM2 2021 Project

In charge:

Name	Student ID	UoM Username	GitHub Username	Email
Guo Xiang	1227317	XIAGUO2	moneynull	xiaguo2@student.unimelb.edu.au
Basrewan Irgio	1086150	IBASREWAN	irgiob	ibasrewan@student.unimelb.edu.au
Bai Zhongyi	1130055	zhongyib	Marvin3Benz	zhongyib@student.unimelb.edu.au
Hernán Romano	1025543	hromanocuro	hromanoc	hromanocuro@student.unimelb.edu.au

Release Tag: SWEN90007_2022_Part2_Nuroom

Revision History

Date	Version	Description	Author
02/09/2022	01.00-D1	Initial document	Hernán Romano
05/09/2022	01.00-D2	Restructure of document	Team
06/09/2022	01.00-D3	Adding Identity Field design pattern	Hernán Romano
07/09/2022	01.00-D4	Updating Use Cases and Diagram	Zhongyi Bai
08/09/2022	01.00-D5	Added Data Mappers pattern	Irgio Basrewan
09/09/2022	01.00-D6	Adding Foreign key design pattern	Hernán Romano
10/09/2022	01.00-D7	Added Embedded Value	Zhongyi Bai
12/09/2022	01.00-D8	Added Data Mapper diagram	Xiang Guo
13/09/2022	01.00-D9	Added security pattern	Xiang Guo
15/09/2022	01.00-D10	Add lazy load & domain model patterns	Irgio Basrewan
16/09/2022	01.00-D11	Adding inheritance design pattern	Hernán Romano
17/09/2022	01.00-D12	Updating description of lazy load pattern	Irgio Basrewan
20/09/2022	01.00-D13	Added Class Diagrams	Zhongyi Bai
20/09/2022	01.00-D14	Updated security pattern description	Xiang Guo
21/09/2022	01.00	Version 1 & Create Release tag	Team

Contents

1. Introduction

1.1 Proposal

This document specifies the SWEN90007 project's detailed use cases, actors, updated domain model, Class Diagram and the description of each Design Pattern implemented in the system.

1.2 Target Users

This document is mainly intended for SWEN90007 students and the teaching team.

1.3 Conventions, terms and abbreviations

This section explains the concept of some important terms that will be used throughout this document. These terms are detailed alphabetically in the following table.

Term	**Description**
UML	Unified Modelling Language
Hotelier Group	A set of Hoteliers under a common hotel business interest
Amenity	A desirable or useful feature or facility of a hotel or room

2. Actors

Actor	**Description**
Customer	A person who makes use of the Hotel Booking System and acquires services
Administrator	A person who manages the Hotel Booking System
Hotelier	A person who manages hotel(s)

3. Expanded Use Cases

3.1 Use Case Diagram

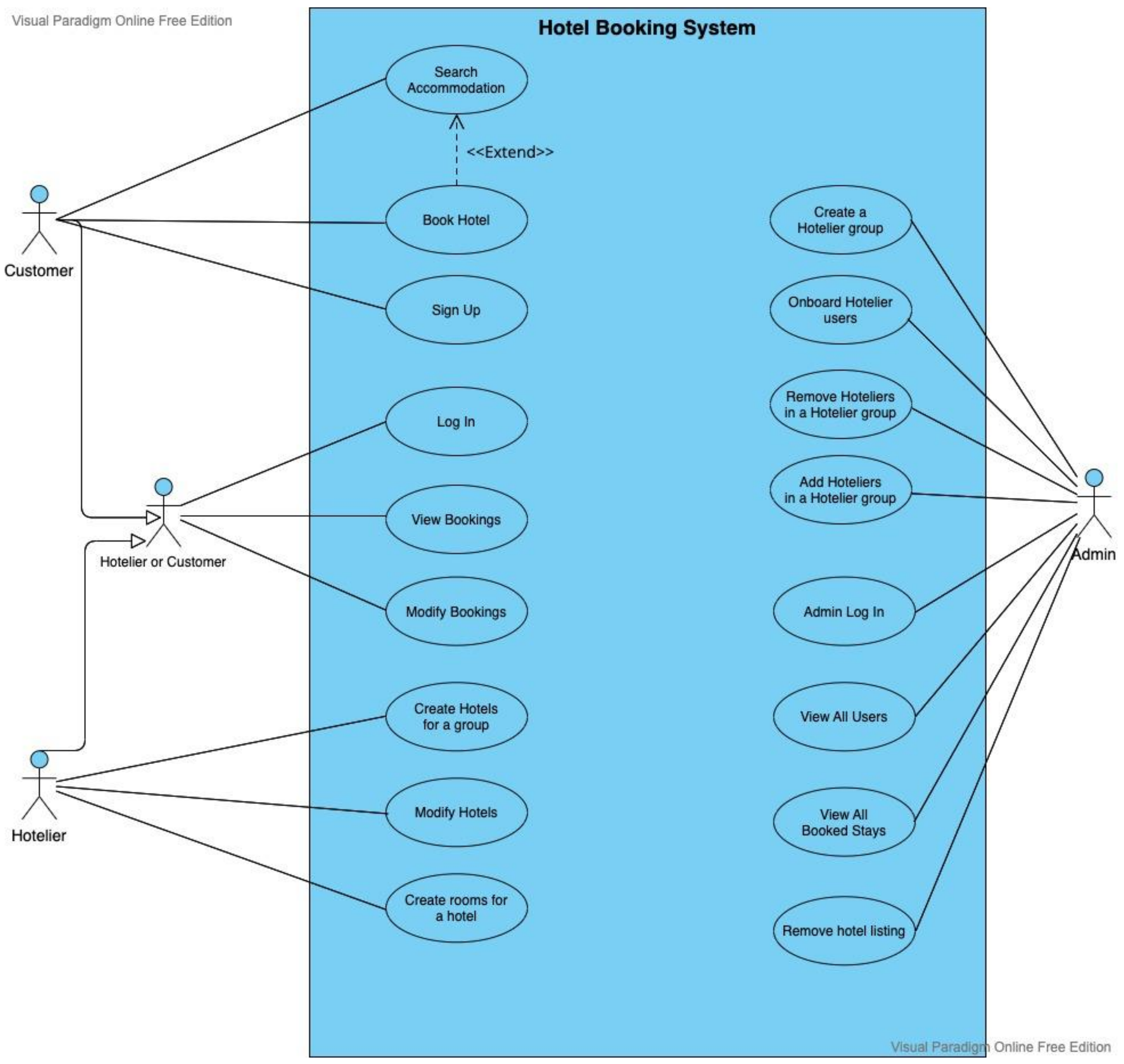


Figure 3.1: Use Cases Diagram

3.2 List of Use Cases

The **Table 3.1** shows a summary of the use cases considered in the project:

Table 3.1: List of Use Cases

Use Case ID	Use Case Name
01	Customer Signs Up
02	Customer Searches for accommodations
03	Customer Books Hotel Rooms
04	Hotelier or Customer Logs In
05	Hotelier or Customer Views Bookings
06	Hotelier or Customer Modifies Bookings: Alternative flows(Add more people, Changes dates, Cancel Booking)
07	Hotelier Creates Rooms for a Hotel
08	Hotelier Creates Hotels for a group
09	Hotelier Modifies Hotels
10	Admin Logs In
11	Admin Remove hotel listing
12	Admin Views All Users
13	Admin Views All Booked Stays
14	Admin Onboards Hoteliers
15	Admin Remove Hoteliers in a Hotelier group
16	Admin Adds Hoteliers in a Hotelier group
17	Admin Creates a Hotelier group

3.2.1Use Case 01: Customer Signs Up

Actors

1. Customer

Basic Flow

A customer wants to view the bookings; however, she cannot process the booking because she needs to login into the platform. It is her first time using it, and she will need to create a new account. After completing a form with basic personal information, which the system validates and records, she creates her new account successfully. This person receives a confirmation of the newly created account. The system redirects her back to the login page.

Table 3.2: Use Case 01

Use Case 01		
ID of Use Case	UC-01	
Name of Use Case	Customer Signs Up	
Actor(s)	- Customer	
Description	A customer wants to view the bookings; however, she cannot process the booking because she needs to login into the platform. It is her first time using it, and she will need to create a new account. After completing a form with basic personal information, which the system validates and records, she creates her new account successfully. This person receives a confirmation of the newly created account. The system redirects her back to the login page.	
Trigger	Customer wants to book a hotel.	
Pre-conditions	Customer's account is not existing in the current system.	
Post-conditions	A customer account has been created in the current system.	
Main flow	Step 1	Customer opens NuRoom homepage via browser
	Step 2	Customer selects "Sign Up"
	Step 3	System displays Sign Up page
	Step 4	Customer enters his/her email, first name, last name, date of birth, phone number, and password
	Step 5	Customer selects "Sign Up"
	Step 6	System validates account
	Step 7	System creates a new account for this customer

	Step 8	Customer logs in to NuRoom

Exception Flows

Table 3.3: Use Case 01 Exception Flow

Exception Flow 01	
ID of Exception Flow	EF-01-01
Steps	**Description**
Step 1	Customer enters his/her email, first name, last name, date of birth, phone number, and password
Step 2	System detects this email has been registered before
Step 3	System displays a message "Email has been registered"
Step 4	Customer uses another email to sign up

Additional Information

Table 3.4: Use Case 01 Additional Information

Field	**Description**	**Type (Length)**	**Mandatory**	**Unique**	**Editable**	**Filtered**	**Details**
Email	Customer's emial	String	✓	✓			The email will be used to log into the system
First name	Customer's first name	String	✓		✓		
Last name	Customer's last name	String	✓		✓		
Date of birth	Customer's date of birth	Date	✓		✓		
Phone number	Customer's phone number	String	✓		✓		The phone number used to contact
Passow rd	Customer's password	String	✓		✓		The password used to

Field	Description	Type (Length)	Mandatory	Unique	Editable	Filtered	Details
							verify customer

3.2.2Use Case 02: Customer Searches for accommodations

Actors

1. Customer

Basic Flow

On Monday, a user (customer or hotelier) wants to login into **NüRoom** to view his bookings. He fills out the form with his account and correct password on the Login page. After clicking the login button, he is authenticated to do further actions. (e.g. view bookings, cancel bookings, and modify bookings)

Table 3.5: Use Case 02

Use Case 02		
ID of Use Case	UC-02	
Name of Use Case	Customer Searches for accommodations	
Actor(s)	- Customer	
Description	A customer can search hotels via countries or postcode	
Trigger	Customer wants to book a hotel	
Pre-conditions	None	
Post-conditions	Customers can view a list of hotels	
Main flow	Step 1	Customer open NuRoom homepage via browser
	Step 2	Customer enters country’s name or hotel’s name or postcode and check in/out dates, number of rooms and number of guests.
	Step 3	Customer selects search icon
	Step 4	System finds all hotels based on customer’s input
	Step 5	System displays a list of hotels

Exception Flows

Table 3.6: Use Case 02 Exception Flow

Exception Flow 01	
ID of Exception Flow	EF-02-01
Steps	**Description**
Step 1	Customer enters country's name or hotel's name or postcode and check in/out dates, number of rooms and number of guests.
Step 2	System does not find a hotel in this location
Step 3	System displays "No hotels found"

Additional Information

Table 3.7: Use Case 02 Additional Information

Field	**Description**	**Type (Length)**	**Mandatory**	**Unique**	**Editable**	**Filtered**	**Details**
Country name	The country customer wants to travel to	String			✓	✓	The country name is used to filter hotels
Hotel name	The hotel customer wants to stay	String			✓	✓	The hotel name is used to filter hotels
Postcode	A postcode of a specific area	String			✓	✓	The postcode is used to filter hotels
Check in date	The date customer wants to check in	Date	✓		✓	✓	Check in date is used to check the availability of a hotel
Check out date	The date customer wants to	Date	✓		✓	✓	Check out date is used to

Field	Description	Type (Length)	Mandatory	Unique	Editable	Filtered	Details
	check out						check the availability of a hotel
Number of rooms	The number of rooms customer needs	Integer	✓		✓	✓	Check in date is used to check the availability of a hotel
Number of travellers	The number of travellers	Integer	✓		✓		

3.2.3Use Case 03: Customer Books Hotel Rooms

Actors

1. Customer

Basic Flow

A customer has an urgent work trip for the coming week. For this, she decides to search for the best accommodation deal close to downtown on her favourite Hotel Booking Platform: **NüRoom**

After successfully logging into the Hotel Booking Platform, the customer sees the landing page of the website. The customer visualizes a search bar and searches for accommodation by postcode, number of rooms, and amenities, and picks the booking dates. The system validates the data. The system retrieves the requested information. The customer visualizes a list of hotels with matched characteristics.

Table 3.8: Use Case 03

Use Case 03	
ID of Use Case	UC-03
Name of Use Case	Customer Books Hotel Rooms
Actor(s)	- Customer
Description	A customer can book a hotel using NuRoom

Trigger	Customer wants to book a hotel	
Pre-conditions	Customer logged into the system Customer found a hotel	
Post-conditions	A booking will be created in the system Cutomer can view/modify/cancel this booking	
Main flow	Step 1	Customer finds a hotel
	Step 2	Customer selects “Reserve”
	Step 3	Customer check booking information
	Step 4	Customer selects “Complete Booking”
	Step 5	System validates booking
	Step 6	System creates a booking
	Step 7	System displays booking result

Exception Flows

Table 3.9: Use Case 03 Exception Flows

Exception Flow 01	
ID of Exception Flow	EF-03-01
Steps	**Description**
Step 1	Customer finds a hotel
Step 2	Customer selects “Reserve”
Step 3	Customer check booking information
Step 4	Customer selects “Complete Booking”
Step 5	System validates booking
Step 6	System finds this hotel has been booked
Step 7	System displays “Hotels has been booked”
Step 8	Customer books other hotels

Additional Information

Table 3.10: Use Case 03 Additional Information

Field	**Description**	**Type (Length)**	**Mandatory**	**Unique**	**Editable**	**Filtered**	**Details**
Traveller first name		String	✓		✓		Traveller name is used to contact
Traveller last name		String	✓		✓		Traveller name is used to contact
Traveller phone number		String	✓		✓		Traveller phone number is used to receive text alerts about this trip

3.2.4 Use Case 04: Hotelier or Customer Logs In

Actors

1. Hotelier or Customer

Basic Flow

On Monday, a customer or hotelier wants to login into **NüRoom** to view his bookings. He fills out the form with his account and correct password on the Login page. After clicking the login button, he is authenticated to do further actions. (e.g. view bookings, cancel bookings, and modify bookings)

Table 3.11: Use Case 04

Use Case 04	
ID of Use Case	UC-04
Name of Use Case	Hotelier or Customer Logs in
Actor(s)	-Hotelier or Customer
Description	Hotelier or Customer can log into the system to do some actions.

Trigger	Hotelier or Customer wants to view bookings	
Pre-conditions	Hotelier or Customer's account is existing in the system	
Post-conditions	Hotelier or Customer is authenticated to log in the system and do some actions (e.g. view bookings, modify bookings)	
Main flow	Step 1	Hotelier or Customer opens NuRoom via browser
	Step 2	Hotelier or Customer selects "Sign In"
	Step 3	System displays Sign In page
	Step 4	Hotelier or Customer enters correct email and password
	Step 5	Hotelier or Customer selects "Sign In"
	Step 6	System validates email and password
	Step 7	Hotelier or Customer is authenticated (logged in)

Exception Flows

Table 3.12: Use Case 04 Exception Flows

Exception Flow 01	
ID of Exception Flow	EF-04-01
Steps	**Description**
Step 1	Hotelier or Customer opens NuRoom via browser
Step 2	Hotelier or Customer selects "Sign In"
Step 3	System displays Sign In page
Step 4	Hotelier or Customer enters wrong email and password
Step 5	Hotelier or Customer selects "Sign In"
Step 6	System validates email and password
Step 7	System display wrong password message
Step 8	Hotelier or Customer is not authenticated

Additional Information

Table 3.13: Use Case 04 Additional Information

Field	**Description**	**Type (Length)**	**Mandatory**	**Unique**	**Editable**	**Filtered**	**Details**
Email	Account email	String	✓	✓	✓		The emails are used to identify the customer or hotelier
Password	Account password	String	✓		✓		The password is used to validate account

3.2.5Use Case 05: Hotelier or Customer Views Bookings

Actors

1. Hotelier or Customer

Basic Flow

A customer or hotelier has a hotel reservation, and he wants to double-check whether the start date is correct or not. After he logs in to the system, he is able to view all bookings he has. The bookings information will be displayed, such as hotel name, location, number of rooms, number of beds, and others.

Table 3.14: Use Case 05

Use Case 05	
ID of Use Case	UC-05
Name of Use Case	Hotelier or Customer Views Bookings
Actor(s)	- Hotelier or Customer
Description	Hotelier or Customer can view bookings and know the basic information of each booking, such as check-in/out date, number of travellers and so on.
Trigger	Hotelier or Customer wants to view check-in/out date of a booking
Pre-conditions	Hotelier or Customer is logged into the system
Post-conditions	System displays all bookings information

Main flow	Step 1	Hotelier or Customer opens NuRoom via browser
	Step 2	Hotelier or Customer selects “Trips”
	Step 3	System validates Hotelier or Customer account
	Step 4	System displays a list of bookings

Alternatives Flows

Table 3.15: Use Case 05 Alternatives Flows

Alternative Flow 01	
ID of Alternative Flow	AF-05-01
Steps	**Description**
Step 1	Hotelier opens NuRoom via browser
Step 2	Hotelier selects “Trips”
Step 3	Hotelier selects a specific hotelier group
Step 4	System validates Hotelier account
Step 5	System displays a list of bookings tied to hotelier group

Exception Flows

Table 3.16: Use Case 05 Exception Flows

Exception Flow 01	
ID of Exception Flow	EF-05-01
Steps	**Description**
Step 1	Hotelier or Customer opens NuRoom via browser
Step 2	Hotelier or Customer selects “Trips” without logging in
Step 3	System validates Hotelier or Customer account
Step 4	System displays “Need to sign in first”

3.2.6 Use Case 06: Hotelier or Customer Modifies Bookings

Actors

1. Hotelier or Customer

Basic Flow

A customer or hotelier has a hotel reservation and wants to modify the start date. After he logs in to the system, he is able to view all bookings he has. The bookings information will be displayed. Then, he selects to edit the start date. After he picks a suitable date and confirms to submit this modification. The customer or hotelier will receive a notification that a booking modification needs to be approved. The hotelier or customer will approve the modification and the start date will be updated.

Table 3.17: Use Case 06

Use Case 06		
ID of Use Case	UC-06	
Name of Use Case	Hotelier or Customer Modifies Bookings	
Actor(s)	- Hotelier or Customer	
Description	Hotelier or Customer can modify a booking because of the travel plan changes	
Trigger	Hotelier or Customer wants to cancel a booking Customer wants to change check in date or add more people	
Pre-conditions	Hotelier or Customer is logged into the system and system displays a list of bookings	
Post-conditions	Hotelier or Customer modifies a booking System updates this booking's information	
Main flow	Step 1	Hotelier or Customer selects one booking
	Step 2	Hotelier or Customer selects "Cancel Booking"
	Step 3	System displays confirmation message
	Step 4	Hotelier or Customer selects "Confirm"
	Step 5	System validates account
	Step 6	System displays "Booking Cancelled"

Alternatives Flows

Table 3.18: Use Case 06 Alternatives Flows

Alternative Flow 01	
ID of Alternative Flow	AF-06-01
Steps	**Description**
Step 1	Customer selects one booking
Step 2	Customer changes check in/out date
Step 3	System displays confirmation message
Step 4	Customer selects “Confirm”
Step 5	System validates account
Step 6	System displays “Booking Information Updated”

Alternative Flow 02	
ID of Alternative Flow	AF-06-02
Steps	**Description**
Step 1	Customer selects one booking
Step 2	Customer adds more people on traveller field
Step 3	System displays confirmation message
Step 4	Customer selects “Confirm”
Step 5	System validates account
Step 6	System displays “Booking Information Updated”

Additional Information

Table 3.19: Use Case 06 Additional Information

Field	**Description**	**Type (Length)**	**Mandatory**	**Unique**	**Editable**	**Filtered**	**Details**
Check in/out date		Date			✓		The check-in/out date is used to specify when to

Field	Description	Type (Length)	Mandatory	Unique	Editable	Filtered	Details
							stay and leave

3.2.7Use Case 07: Hotelier Creates Rooms for a Hotel

Actors

1. Hotelier

Basic Flow

A hotelier has created a hotel for a group, and now he wants to create several rooms for this hotel. After logging into the system. He can select a hotel he just created, and he is able to set the number of bedrooms, bathrooms, and amenities of a room. Then, this new room will be recorded in the system. Customers can view and book these rooms.

Table 3.20: Use Case 07

Use Case 07		
ID of Use Case	UC-07	
Name of Use Case	Hotelier Creates Rooms for a Hotel	
Actor(s)	- Hotelier	
Description	A hotelier is able to create rooms for a hotel and setup room’s type	
Trigger	Hotelier wants to publish a hotel	
Pre-conditions	Hotelier is logged into the system A hotel has been created	
Post-conditions	Hotelier creates new rooms for a specific hotel System adds new rooms	
Main flow	Step 1	Hotelier open NuRoom via browser
	Step 2	Hotelier selects “List properties”
	Step 3	System validates account
	Step 4	System displays a list of hotels hotelier created
	Step 5	Hotelier selects one hotel
	Step 6	Hotelier adds required fields, unit number, number of bedrooms, number of bathrooms, number of beds, description, amenities, price, image (url)

	Step 7	System validates account
	Step 8	System adds this new room for a hotel
	Step 9	System display “Room added”

Additional Information

Table 3.21: Use Case 07 Additional Information

Field	Description	Type (Length)	Mandatory	Unique	Editable	Filtered	Details
Unit number		String	✓		✓		The unit number is used to identify different rooms
number of bedrooms		Integer	✓		✓		
number of bathrooms		Integer	✓		✓		
number of beds		Integer	✓		✓		
descriptions		String	✓		✓		The description of this room
amenities		Array	✓		✓		e.g. WiFi, Air conditioning
price		Integer	✓		✓		price per night

3.2.8Use Case 08: Hotelier Creates Hotels for a group

Actors

1. Hotelier

Basic Flow

As a hotelier of a brand new hotel prepares for its grand opening, they decide to choose NüRoom as the platform that will host bookings for rooms in their hotel. After logging into the platform, a hotelier follows the directions in setting up a new hotel on the site. The hotelier inputs information about the general hotel, such as its name, address, and available amenities. Next, the hotelier sets up the multiple room types (that vary in terms of the number of rooms, size, etc.) that are available to stay in at the hotel and denotes the number of rooms of each type that are available to be booked. After submitting the details of the hotel, the hotel is now available to be booked by customer users on the site and the hotelier is able to view the bookings made by customers.

Table 3.22: Use Case 08

Use Case 08		
ID of Use Case	UC-08	
Name of Use Case	Hotelier Creates Hotels for a group	
Actor(s)	- Hotelier	
Description	A hotelier can create a hotel for a hotelier group and set up his/her hotel, e.g. address, number of rooms, etc.	
Trigger	Hotelier wants to list his/her hotels	
Pre-conditions	Hotelier is logged into the system	
Post-conditions	Hotelier creates a hotel for a group System adds a new hotel for a group	
Main flow	Step 1	Hotelier opens NuRoom via browser
	Step 2	Hotelier selects “List properties”
	Step 3	Hotelier selects “Create a new hotel”
	Step 4	Hotelier adds required fields, such as hotel name, country, street address, unit number, city, state, postcode, number of rooms, hotelier group name.
	Step 5	System validates account
	Step 6	System adds new hotel
	Step 7	System display “Hotel added”

Additional Information

Table 3.23: Use Case 08 Additional Information

Field	**Description**	**Type (Length)**	**Mandatory**	**Unique**	**Editable**	**Filtered**	**Details**
Hotel name		String	✓		✓		
Countr y		String	✓		✓		
Street address		String	✓		✓		
Unit number		String			✓		
City		String	✓		✓		
State		String	✓		✓		
Postcod e		String	✓		✓		
Numbe r of rooms		Integer	✓		✓		
Amenit ies	Hotel’s amenities	Array	✓		✓		e.g. Pool, room service
Hotelie r group name		String	✓				

3.2.9Use Case 09: Hotelier Modifies Hotels

Actors

1. Hotelier

Basic Flow

After a dramatic increase in customers due to listing their hotel on NüRoom, the hotelier decides to buy the land surrounding the hotel to expand the hotel and the number of rooms available. After construction is complete in the hotel’s new wings, the hotelier updates the hotel listing on NüRoom by adding new amenities to the hotel.

Table 3.24: Use Case 09

Use Case 09		
ID of Use Case	UC-09	
Name of Use Case	Hotelier Modifies Hotels	
Actor(s)	- Hotelier	
Description	A hotelier can update hotel's information, such as amenities, number of rooms	
Trigger	Hotelier wants add new amenities to a hotel	
Pre-conditions	Hotelier is logged into the system A hotel has been created	
Post-conditions	Hotelier changes hotel's information System updates hotel's information	
Main flow	Step 1	Hotelier opens NuRoom via browser
	Step 2	Hotelier selects "List perporties"
	Step 3	Hotelier selects one hotel
	Step 4	Hotelier selects "Update"
	Step 5	Hotelier changes some fields, such as amenities, number of rooms, and postcode, etc
	Step 6	System validates account
	Step 7	System updates hotel's information
	Step 8	System displays "Hotel information updated"

Additional Information

Table 3.25: Use Case 09 Additional Information

Field	**Description**	**Type (Length)**	**Mandatory**	**Unique**	**Editable**	**Filtered**	**Details**
Hotel name		String	✓		✓		
Countr y		String	✓		✓		
Street address		String	✓		✓		
Unit		String			✓		

Field	Description	Type (Length)	Mandatory	Unique	Editable	Filtered	Details
number							
City		String	✓		✓		
State		String	✓		✓		
Postcode		String	✓		✓		
Number of rooms		Integer	✓		✓		
Amenities	Hotel's amenities	Array	✓		✓		e.g. Pool, room service

3.2.10 Use Case 10: Admin Logs In

Actors

1. Administrator

Basic Flow

On Monday, an administrator wants to log into the system via the admin portal to deactivate a user, because this hotelier uploaded an image against the policy. He fills out the form with his account and correct password on the admin login page. After clicking the login button, he is authenticated to do further actions. (e.g. deactivate users, view all users).

Table 3.26: Use Case 10

Use Case 10		
ID of Use Case	UC-10	
Name of Use Case	Admin Logs In	
Actor(s)	- Administrator	
Description	Admin is able to log into the system to do some actions.	
Trigger	Admin wants to onboard hotelier user	
Pre-conditions	Admin account is existing in the system	
Post-conditions	Admin is logged into the system	
Main flow	Step 1	Admin opens NuRoom admin login page via browser

	Step 2	Admin enters correct email and password
	Step 3	System validates admin account
	Step 4	Admin is authenticated (logged in)

Exception Flows

Table 3.27: Use Case 10 Exception Flows

Exception Flow 01	
ID of Exception Flow	EF-10-01
Steps	**Description**
Step 1	Admin opens NuRoom admin login page via browser
Step 2	Admin enters wrong email and password
Step 3	System validates email and password
Step 4	System display wrong password message
Step 5	Admin is not authenticated

Additional Information

Table 3.28: Use Case 10 Additional Information

Field	**Description**	**Type (Length)**	**Mandatory**	**Unique**	**Editable**	**Filtered**	**Details**
Email	Admin email	String	✓	✓			The emails is used to identify customer or hotelier
Passwo rd	Admin password	String	✓				The password is used to validate account

3.2.11 Use Case 11: Admin Remove hotel listing

Actors

1. Administrator

Basic Flow

An admin wants to remove a hotel from NuRoom because the hotel is not available anymore. After logging into the system, the admin is able to view all hotels and remove hotels from the list. After removing the hotel, this hotel will not be available to book.

Table 3.29: Use Case 11

Use Case 11		
ID of Use Case	UC-11	
Name of Use Case	Admin Remove hotel listing	
Actor(s)	- Administrator	
Description	After logging into the system, the admin is able to view all hotels and remove hotels from the list. After removing the hotel, this hotel will not be available to book.	
Trigger	Admin wants to remove a hotel from the hotel list	
Pre-conditions	Admin is logged into the system	
Post-conditions	The selected hotel will be marked as unavailable	
Main flow	Step 1	Admin selects “All hotels”
	Step 2	System displays a list of all hotels
	Step 3	Admin selects “Remove” of a hotel
	Step 4	System displays confirmation message
	Step 5	Admin confirm to remove
	Step 6	System validates account
	Step 7	System marks the hotel as unavailable

3.2.12 Use Case 12: Admin Views All Users

Actors

1. Administrator

Basic Flow

For the regular report of the current usage of the system, the administrator would like to have an overall view of how many users and hoteliers are registered in the system. When an administrator wants to check the profile and information of specific users and specific hoteliers, they also have access to view all the details of users to guarantee all the information provided by users is legal and accurate.

Table 3.30: Use Case 12

Use Case 12		
ID of Use Case	UC-12	
Name of Use Case	Admin Views All Users	
Actor(s)	- Administrator	
Description	Admin views all hoteliers and customers	
Trigger	Admin wants to view how many users are in the system	
Pre-conditions	Admin is logged into the system	
Post-conditions	System displays a list of hoteliers and customers	
Main flow	Step 1	Admin selects “Add users”
	Step 2	System validates account
	Step 3	System displays a list of all hoteliers and customers

3.2.13 Use Case 13: Admin Views All Booked Stays

Actors

1. Administrator

Basic Flow

The administrator might want to make an analysis of the popularity of different types of rooms and different location hotels and calculate the average number of daily booked stays. So, it is necessary for the administrator to monitor the vacancy and status of all hotels, and they can view all stays that customers have booked.

Table 3.31: Use Case 13

Use Case 13		
ID of Use Case	UC-13	
Name of Use Case	Admin Views All Booked Stays	
Actor(s)	- Administrator	
Description	An admin views all booked stays	
Trigger	Admin wants to view all booked stays	
Pre-conditions	Admin is logged into the system	
Post-conditions	Systems displays a list of all booked stays	
Main flow	Step 1	Admin selects “All booked Stays”
	Step 2	System validates account
	Step 3	System displays a list of all booked stays

3.2.14 Use Case 14: Admin Onboards Hoteliers

Actors

1. Administrator

Basic Flow

When new hotelier wants to join the platform to list their hotels, they can ask the administrator to onboard hotelier accounts for them, and the hoteliers can log into the application and create new hotels. Thus, the hoteliers cannot register by themselves, only the administrator can onboard hoteliers in the system.

Table 3.32: Use Case 14

Use Case 14	
ID of Use Case	UC-14
Name of Use Case	Admin Onboards Hoteliers
Actor(s)	- Administrator
Description	Admin onboards a hotelier
Trigger	Amind wants to create an account for a new hotelier
Pre-conditions	Admin is logged into the system

Post-conditions	System creates an account for the new hotelier	
Main flow	Step 1	Admin selects "Onboard Hoteliers"
	Step 2	Admin enters required fields, name, date of birth, phone number.
	Step 3	System validates account
	Step 4	System displays "Hotelier added"

Additional Information

Table 3.33: Use Case 14 Additional Information

Field	**Description**	**Type (Length)**	**Mandatory**	**Unique**	**Editable**	**Filtered**	**Details**
Email	Hotelier email	String	✓	✓			The email will be used to log into the system
First name	Hotelier first name	String	✓		✓		
Last name	Hotelier last name	String	✓		✓		
Date of birth	Hotelier date of birth	Date	✓		✓		
Phone number	Hotelier phone number	String	✓		✓		The phone number used to contact
Passow rd	Hotelier password	String	✓		✓		The password used to verify customer

3.2.15 Use Case 15: Admin Remove Hoteliers in a Hotelier group

Actors

1. Administrator

Basic Flow

After logging into the system, an admin is able to remove a hotelier from a hotelier group.

Table 3.34: Use Case 15

Use Case 15		
ID of Use Case	UC-15	
Name of Use Case	Admin Remove Hoteliers in a Hotelier group	
Actor(s)	- Administrator	
Description	After logging into the system, an admin is able to remove a hotelier from a hotelier group.	
Trigger	Admin wants to remove a hotelier from a hotelier group	
Pre-conditions	Admin is logged into the system	
Post-conditions	System remove the hotelier from the hotelier group	
Main flow	Step 1	Admin selects "Manage Hoteliers"
	Step 2	Admin selects "Remove" from the list of hoteliers group
	Step 3	System validates account
	Step 4	System remove hotelier from hotelier group

3.2.16 Use Case 16: Admin Adds Hoteliers in a Hotelier group

Actors

2. Administrator

Basic Flow

After logging into the system, the admin is able to add a hotelier to an existing hotelier group.

Table 3.35: Use Case 16

Use Case 16		
ID of Use Case	UC-16	
Name of Use Case	Admin Adds Hoteliers in a Hotelier group	
Actor(s)	- Administrator	
Description	After logging into the system, the admin is able to add a hotelier in a existing hotelier group.	
Trigger	Admin wants to add a hotelier in a hotelier group	
Pre-conditions	Admin is logged into the system A hotelier group is existing in the system	
Post-conditions	System adds a hotelier in a hotelier group	
Main flow	Step 1	Admin selects "Manage Hoteliers"
	Step 2	Admin selects "Add" from the list of hoteliers group
	Step 3	System validates account
	Step 4	System adds a hotelier in a hotelier group

3.2.17 Use Case 17: Admin Creates a Hotelier group

Actors

3. Administrator

Basic Flow

After logging into the system, the admin is able to create a hotelier group.

Table 3.36: Use Case 17

Use Case 17	
ID of Use Case	UC-17
Name of Use Case	Admin Creates a Hotelier group
Actor(s)	- Administrator
Description	After logging into the system, the admin is able to create a hotelier group.
Trigger	Admin wants to create a Hotelier group

Pre-condition	Admin is logged into the system	
Post-conditions	System creates a new hotelier group	
Main flow	Step 1	Admin selects “Manage Hoteliers”
	Step 2	Admin selects “Add new hotelier group”
	Step 3	Admin enters group name
	Step 4	System validates account
	Step 5	System creates a new hotelier group
	Step 6	System displays “Hotelier group created”

Additional Information

Table 3.37: Use Case 17 Additional Information

Field	**Description**	**Type (Length)**	**Mandatory**	**Unique**	**Editable**	**Filtered**	**Details**
Name	Hotelier group name	String	✓	✓			

4. Domain Model

4.1 Domain Model Diagram

The Domain Model shown in **figure 4.1**, which applies the feedback received by our supervisor during the Part 1 submission.

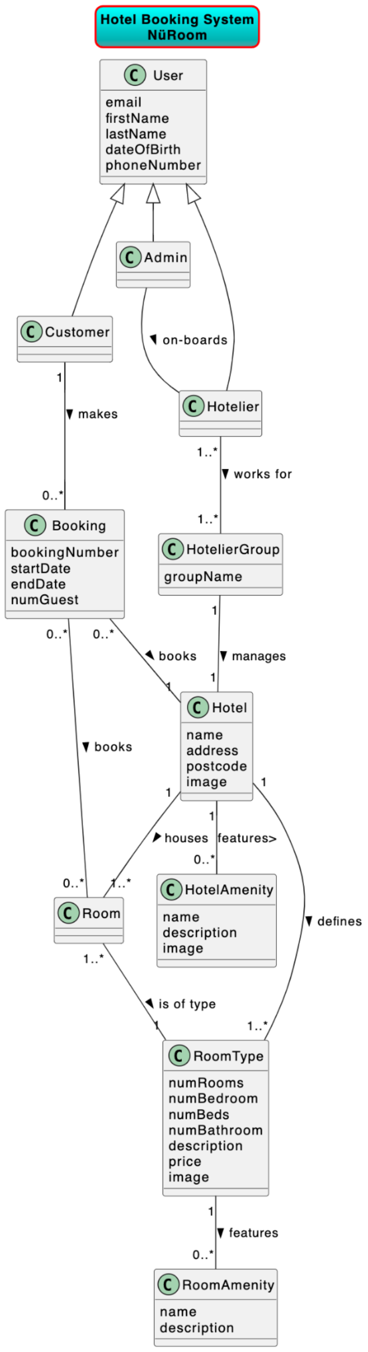


Figure 4.1

4.2 Domain Model Description

The domain model for the NüRoom Hotel Booking System is structured surrounding the main 3 **user** types and their relationships with the different entities of the system. The first **user**, the **admin**, serves as the entity that **manages** the application as a whole and **on-boards** vetted **hoteliers**, the second **user** type onto the system. These **hoteliers** are responsible for managing their **hotels** through **hotel groups** (with each **group** comprising the **hoteliers managing** said **hotel**). Each **hotel defines** its **rooms** through a set of **room types**, each varying in size, **amenities**, price, etc. The **hotel** also defines its own set of **amenities** that its **hotel features**. Finally, there is the third **user** type of **customer**. The **customer** is responsible for making **bookings**, which connect the **customer** with the **room**(s) (and therefore also the **hotel**) that they wish to book.

In summary, the specifications of the hotel booking system dictate the following business requirements:

- **Users** *can be either* an **Administrator**, **Customer**, or **Hotelier**;
- **Administrators** add **Hoteliers** to the system and can view all **Bookings** and **Users;**
- **Hoteliers** can manage *one or more* **Hotels** through **HotelGroups**;
- Each **HotelGroup** manages *one* **Hotel**, and each **Hotel** has *one or more* rooms that can be of *one of multiple* **RoomTypes;**
- **Hotels** can feature *one or more* **HotelAmenity**(s) and **Rooms** can feature *one or more* **RoomAmenity**(s);
- **Customers** can make *one or more* **Bookings** for *one or more* **Rooms;**
- **Rooms** can be in *one or more* **Bookings** at different points in time;

Entities have been bolded; <u>attributes</u> have been underlined; and important *associations* have been italicized.

5. Architecture

In our project, we deployed our frontend on Vercel and backend on Heroku. We use REFTful API to send requests from frontend to backend. Our servlet programs are deployed on the Tomcat server, and everytime Tomcat gets a request from the frontend, it will invoke methods on the servlet program. Then, servlet programs will use JDBC to connect and execute SQL commands on our PostgreSQL database.

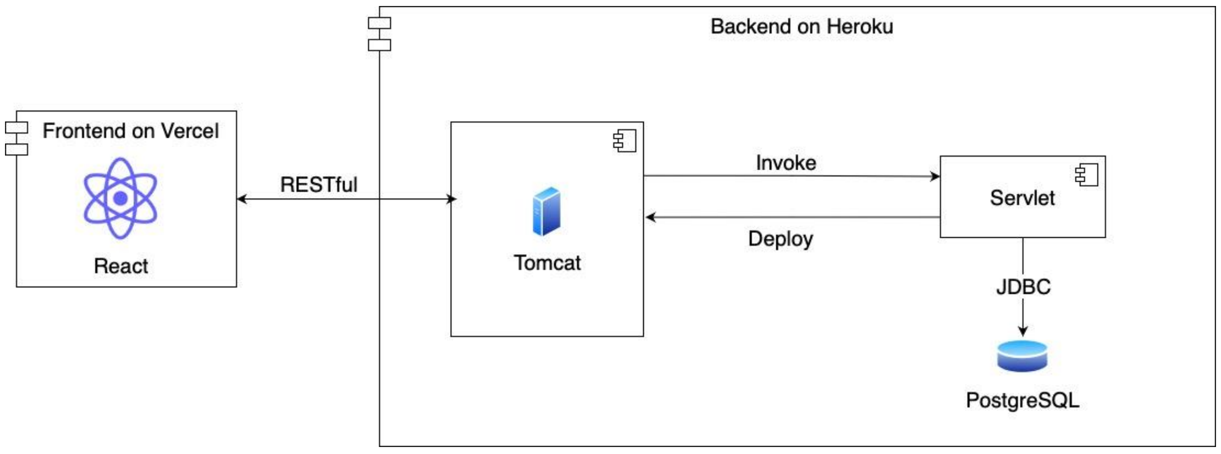


Figure 5.1: High-Level Architecture Diagram

6. Class Diagram

6.1 Description

In our project, we mainly have 3 layers, Controller/Service layer, Domain layer, and Data Source layer. The business logic flow and authentication will be handled in Controller/Service layer. In the Domain layer, we declared all objects that will be used in the system. We put all necessary SQL queries in our Data Source layer so that we can simply invoke the methods to insert/update data in the database. Using these layers, our architecture of the system will be clearer and has high modularity.

6.2 Application Diagram

As shown in **figure 6.1**, we separated frontend and backend development instead of using JSP; thus, we decided to use Application Controller + Operation Script Service to handle the logic flow. Each web page decides which controller to run and respond to through RESTful API requests. Application Controller consumes the methods/service provided in Operation Script Service. Then Operation Script Serice will inject the object into Domain Model. Domain Model sends a query to the Data Mapper. Then, Data Mapper executes SQL command on the database to get results and return Domain

Object(s) back to Domain Model. After Domain Model receives the object(s) from Data Mapper, it will forward the Domain object or just a boolean to Operation Script Service. Afterwards, the result will be formatted and transmitted to Application Controller.

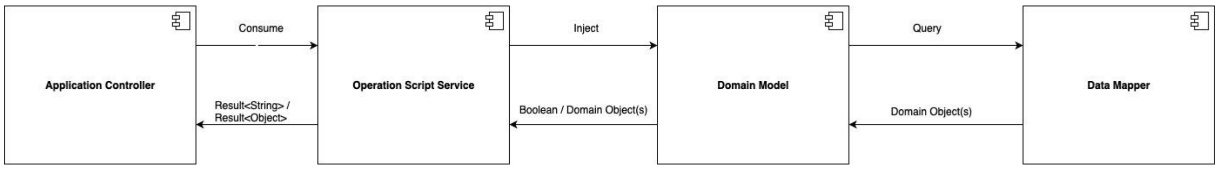


Figure 6.1: Application Diagram

We use JSON string as the data format for communication between the front end and the back end. When a request reaches the controller, the JSON string will be deserialised first. After processing the request, a serialised Domain Object will be sent back to the front end with a JSON object.

6.3 Controller/Service Layer

In the Controller/Service layer, we have 4 controllers and 5 services. The classes in each controller correspond to specific endpoints. For most “view” operations, we just use the GET method, and the others use POST since it requires getting some input from the front end.

Account Control mainly handles the logic related to the user’s account, for instance, login, customer signs up, admin views all users, etc. Additionally, each method is also corresponding to a specific function in service. Account Control connects to the Account Service. Some services’ methods also make use of ViewObject (VO) to ensure the system can manage the data format of requests from the front end properly. The VO will also be used for the deserialisation of JSON objects.

Booking Control mainly handles the logic related to bookings, such as booking a hotel, modifying a booking, and so on. The corresponding service is Booking Service.

Group Control mainly handles the logic related to the hotelier group, such as creating a hotelier group, adding/removing a hotelier in/from a hotelier group, etc. The corresponding service is Group Service.

Hotel Control mainly handles the logic related to hotels, for example, creating a hotel, creating a room for a hotel, modifying the hotel’s information and so on. The corresponding service is Hotel Service.

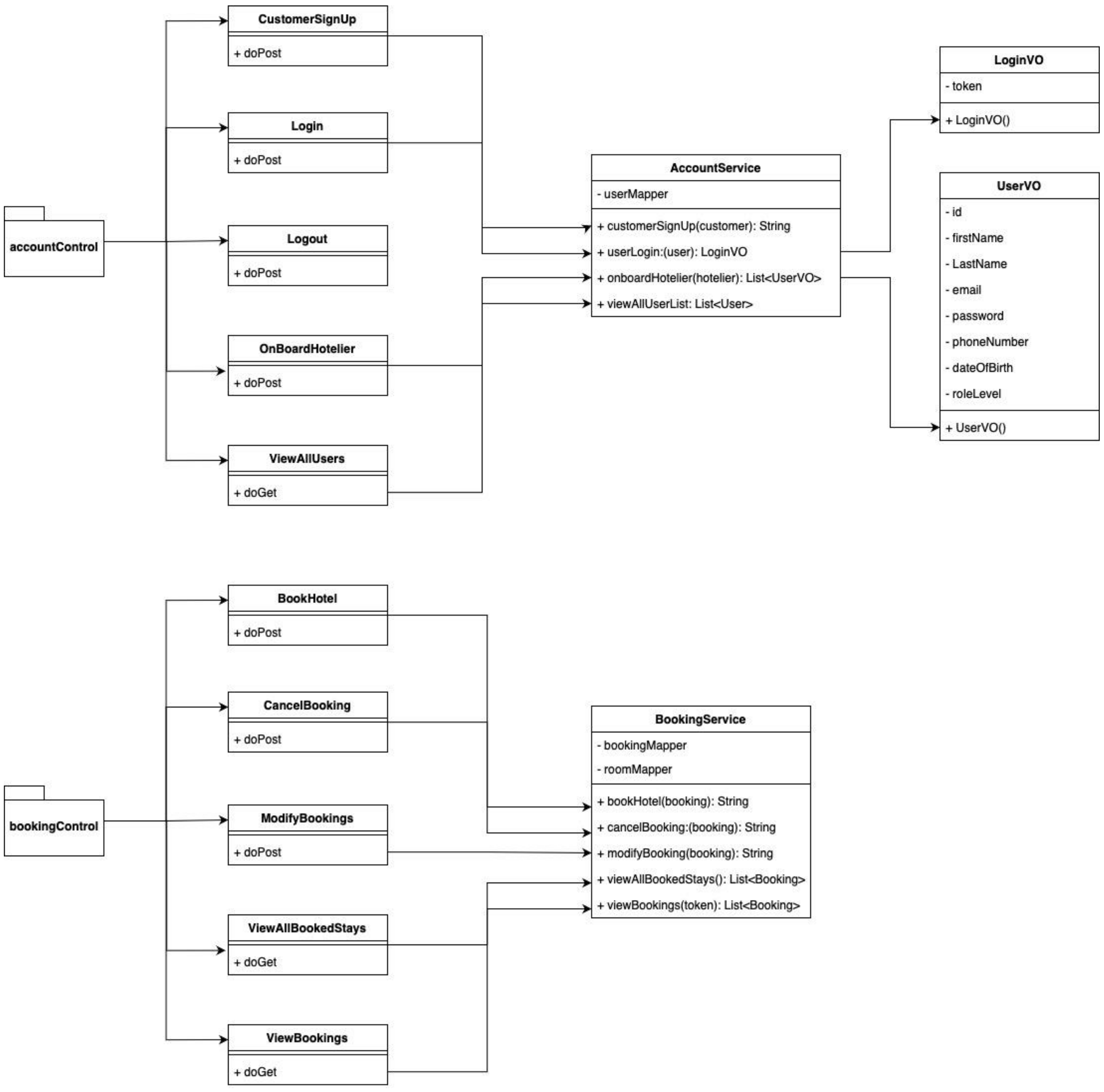


Figure 6.2: Controller/Service Layer Diagram - part 1

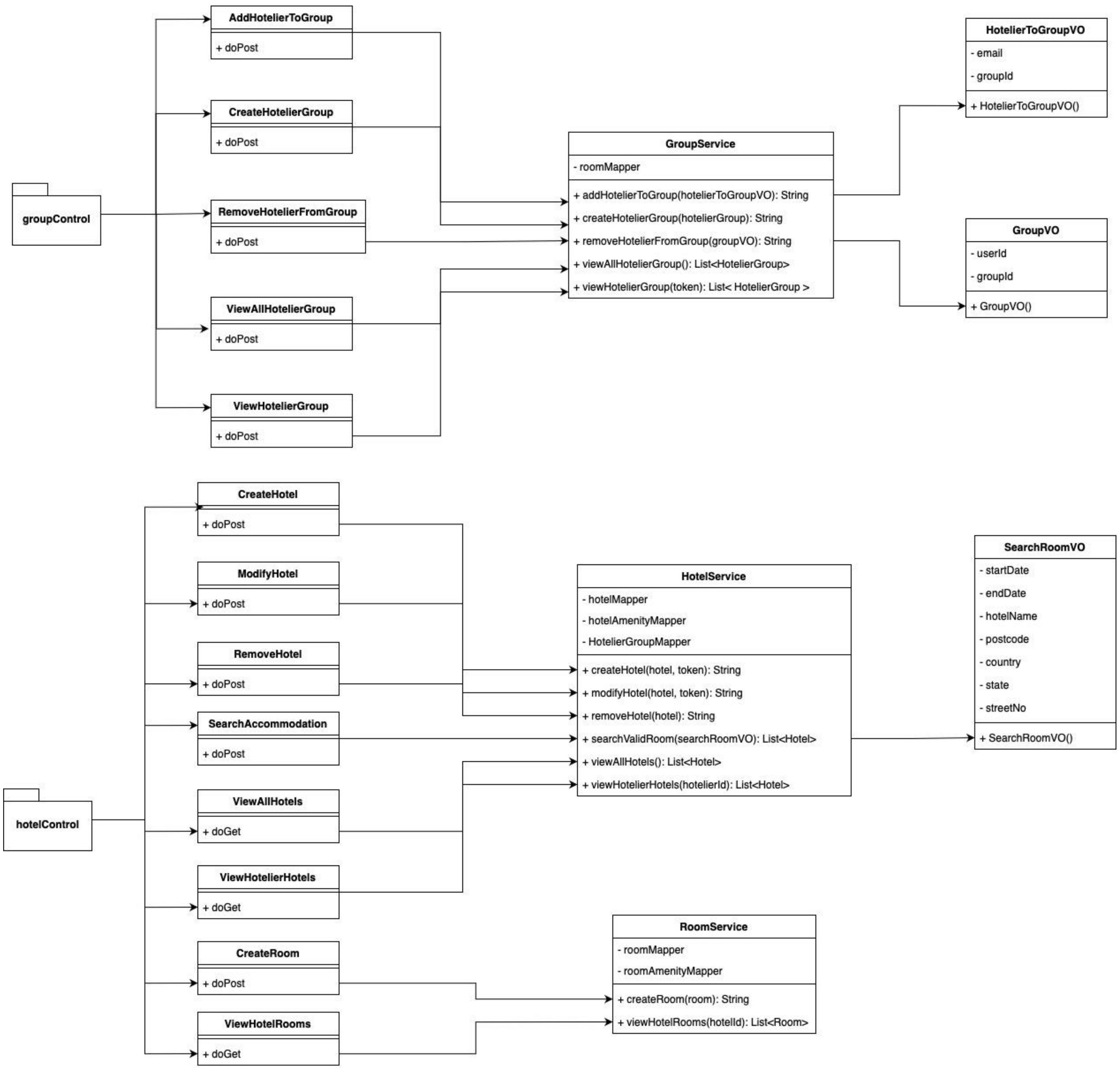


Figure 6.3: Controller/Service Layer Diagram - part 2

6.4 Domain Layer

In the Domain layer, it contains all required domain objects that will be used in the system. For most classes, they mainly have a constructor, getter and setter for each object.

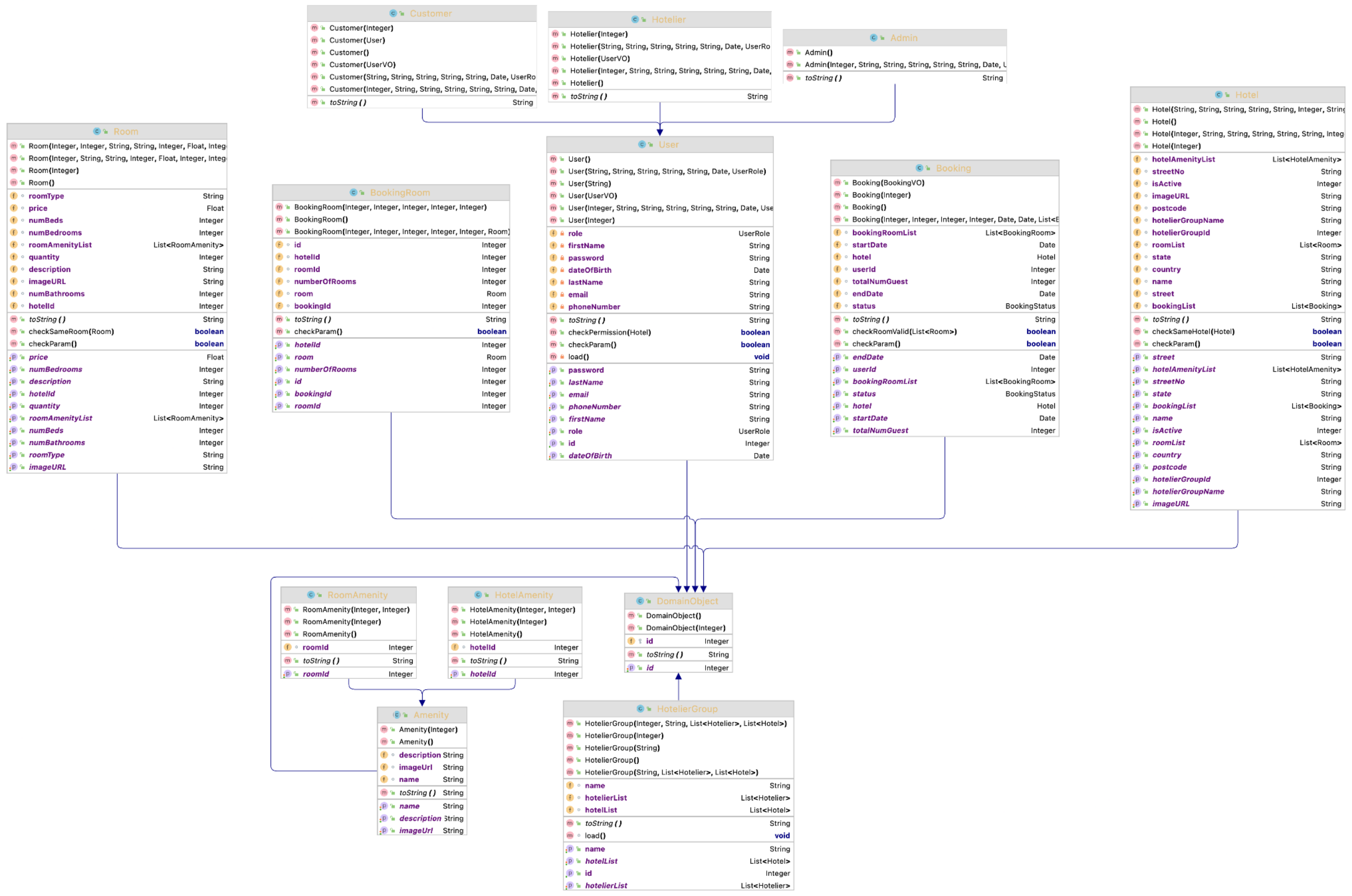


Figure 6.4: Domain Layer Diagram

6.5 Data Source Layer

In Data Source layer, we have a parent class, DataMapper that contains insert(), delete(), update(), getMapper() and findById(). The remaining child classes correspond to database tables. We chose to use this layer due to the following reasons:

1. Our domain logic is complex, especially the book a hotel and modifying hotels.
2. Using a data source layer can help reduce the coupling and separate the domain layer and data mapper, thus changing one will not compromise the others.
3. Additionally, the database schema has also been changed and updated to meet the use case requirements and performance improvement; therefore, it is not a good idea to use table data gateway, row data gateway, or active record.

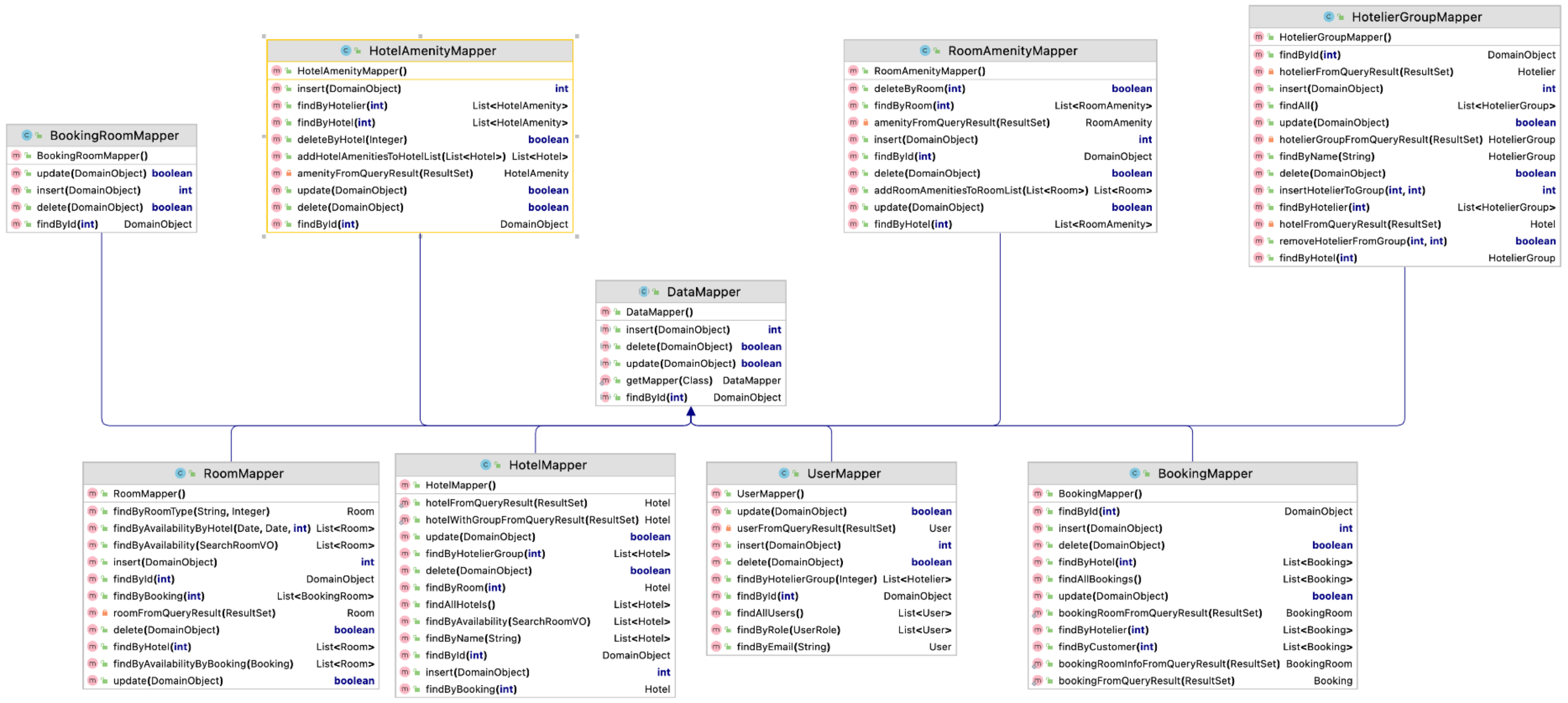

BookingRoomMapper
BookingRoomMapper()
update(DomainObject) boolean
insert(DomainObject) int
delete(DomainObject) boolean
findById(int) DomainObject
HotelAmenityMapper
HotelAmenityMapper()
insert(DomainObject) int
findByHotelier(int) List<HotelAmenity>
findByHotel(int) List<HotelAmenity>
deleteByHotel(Integer) boolean
addHotelAmenitiesToHotelList(List<Hotel>) List<Hotel>
amenityFromQueryResult(ResultSet) HotelAmenity
update(DomainObject) boolean
delete(DomainObject) boolean
findById(int) DomainObject
RoomAmenityMapper
RoomAmenityMapper()
deleteByRoom(int) boolean
findByRoom(int) List<RoomAmenity>
amenityFromQueryResult(ResultSet) RoomAmenity
insert(DomainObject) int
findById(int) DomainObject
delete(DomainObject) boolean
addRoomAmenitiesToRoomList(List<Room>) List<Room>
update(DomainObject) boolean
findByHotel(int) List<RoomAmenity>
HotelierGroupMapper
HotelierGroupMapper()
findById(int) DomainObject
hotelierFromQueryResult(ResultSet) Hotelier
insert(DomainObject) int
findAll() List<HotelierGroup>
update(DomainObject) boolean
hotelierGroupFromQueryResult(ResultSet) HotelierGroup
findByName(String) HotelierGroup
delete(DomainObject) boolean
insertHotelierToGroup(int, int) int
findByHotelier(int) List<HotelierGroup>
hotelFromQueryResult(ResultSet) Hotel
removeHotelierFromGroup(int, int) boolean
findByHotel(int) HotelierGroup
DataMapper
DataMapper()
insert(DomainObject) int
delete(DomainObject) boolean
update(DomainObject) boolean
getMapper(Class) DataMapper
findById(int) DomainObject
RoomMapper
RoomMapper()
findByRoomType(String, Integer) Room
findByAvailabilityByHotel(Date, Date, int) List<Room>
findByAvailability(SearchRoomVO) List<Room>
insert(DomainObject) int
findById(int) DomainObject
findByBooking(int) List<BookingRoom>
roomFromQueryResult(ResultSet) Room
delete(DomainObject) boolean
findByHotel(int) List<Room>
findByAvailabilityByBooking(Booking) List<Room>
update(DomainObject) boolean
HotelMapper
HotelMapper()
hotelFromQueryResult(ResultSet) Hotel
hotelWithGroupFromQueryResult(ResultSet) Hotel
update(DomainObject) boolean
findByHotelierGroup(int) List<Hotel>
delete(DomainObject) boolean
findByRoom(int) Hotel
findAllHotels() List<Hotel>
findByAvailability(SearchRoomVO) List<Hotel>
findByName(String) List<Hotel>
findById(int) DomainObject
insert(DomainObject) int
findByBooking(int) Hotel
UserMapper
UserMapper()
update(DomainObject) boolean
userFromQueryResult(ResultSet) User
insert(DomainObject) int
delete(DomainObject) boolean
findByHotelierGroup(Integer) List<Hotelier>
findById(int) DomainObject
findAllUsers() List<User>
findByRole(UserRole) List<User>
findByEmail(String) User
BookingMapper
BookingMapper()
findById(int) DomainObject
insert(DomainObject) int
delete(DomainObject) boolean
findByHotel(int) List<Booking>
findAllBookings() List<Booking>
update(DomainObject) boolean
bookingRoomFromQueryResult(ResultSet) BookingRoom
findByHotelier(int) List<Booking>
findByCustomer(int) List<Booking>
bookingRoomInfoFromQueryResult(ResultSet) BookingRoom
bookingFromQueryResult(ResultSet) Booking

Figure 6.5: Data Source Layer

7. Design Patterns

7.1 Pattern 1: Domain model

7.1.1Description

The domain model pattern is essentially an object-oriented model of the entities within the business/problem domain. It divides the business logic into the entities that would handle that logic from a domain perspective. It maps the characteristics of entities in the domain through class attributes, and the actions those entities take as methods. The objects representing the entities of the domain are created using data from a database and generated as required for the use case that needs to be performed.

Using the domain model pattern to handle business logic helps manage complexity in the domain. Consider the domain diagram in section 4 made earlier during the design phase. The domain diagram and the entities defined within it directly translate to the domain layer of the software system. The only inclusions made were abstract parent classes for the different user types and amenity types. Beyond that, the relationships between the different entities in the domain diagram are exactly the same as the relationship between the domain classes in the software.

This helps manage the complexity of the problem domain as it is easier to compare the code written directly with the conceptual domain, compared to using another approach like transaction scripts. This also makes the system more extensible as changes can be made to the individual entities that are being modified, or for new features, they can simply be inserted into the existing domain space.

7.1.2Sequence Diagram

The sequence diagram below shows one of the example interactions between different objects/classes in the domain model, particularly the interactions between a hotelier, a hotel, and the hotel's associated hotelier group to check if a hotelier has permission create said hotel within the created hotel's assigned hotelier group.

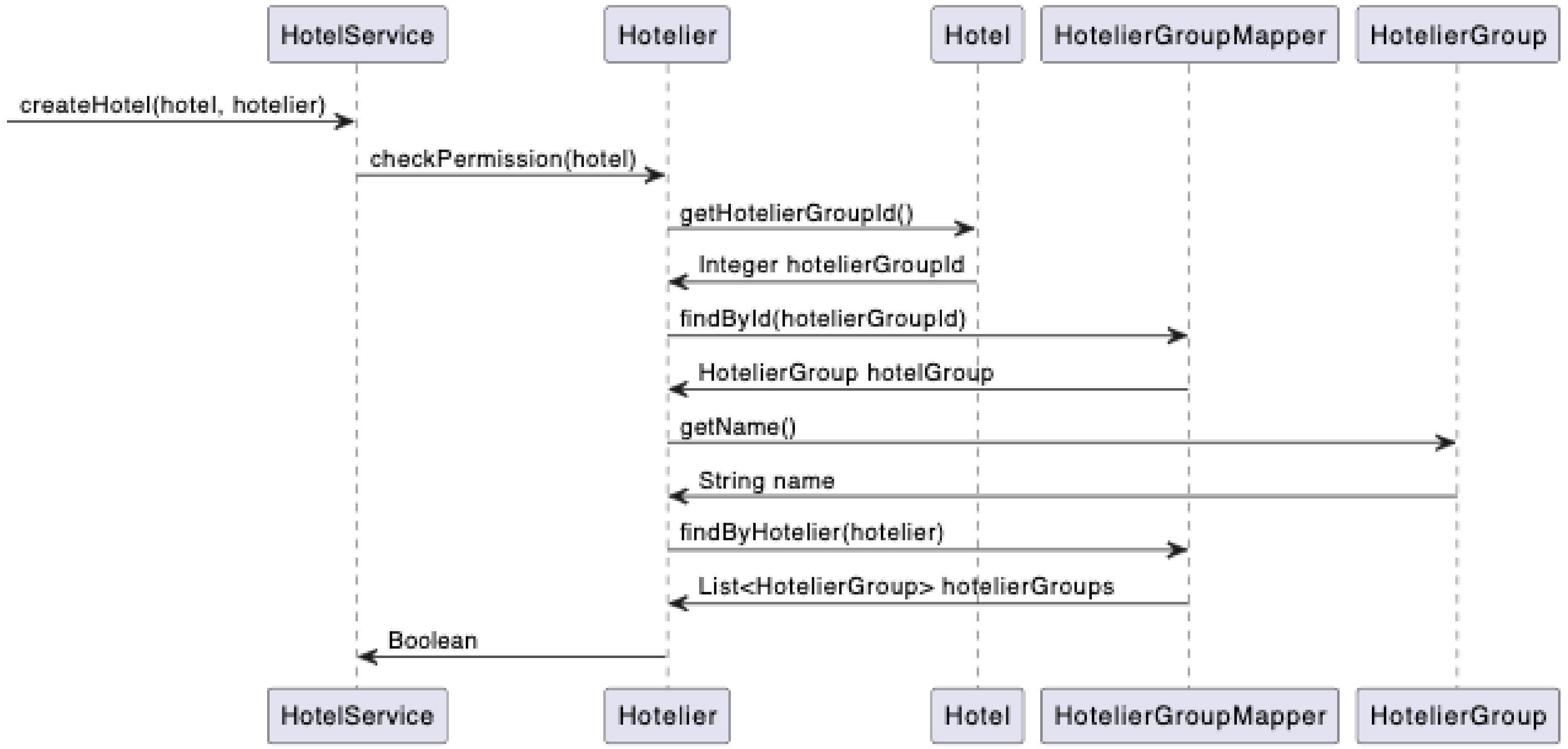
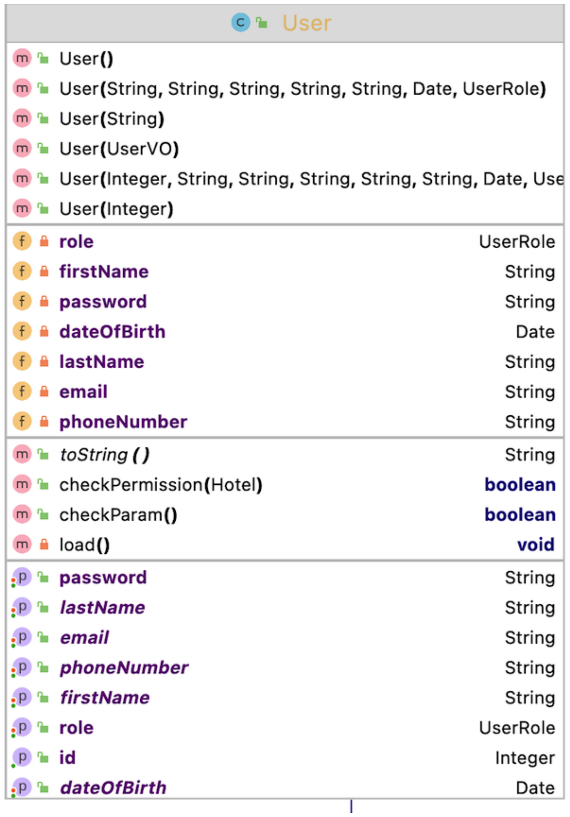
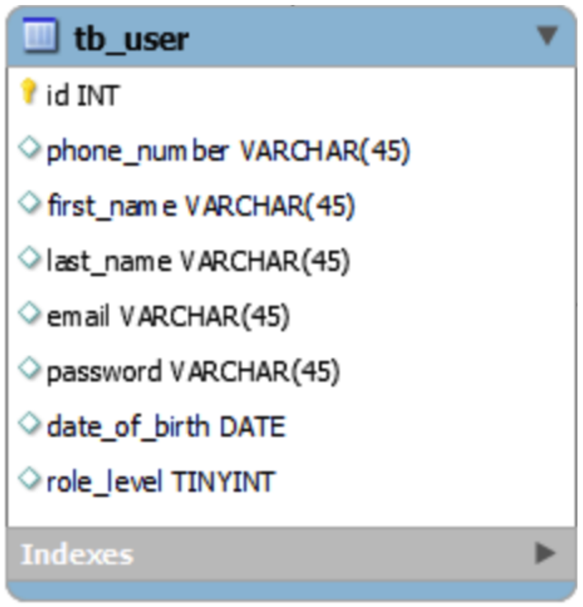


Figure 7.1: Domain Model Sequence Diagram

7.1.3Entity Relationship Diagram

Domain Object for User	User Table
	

```
User
User()
User(String, String, String, String, String, Date, UserRole)
User(String)
User(UserVO)
User(Integer, String, String, String, String, String, Date, Use
User(Integer)
role                                   UserRole
firstName                              String
password                               String
dateOfBirth                            Date
lastName                               String
email                                  String
phoneNumber                            String
toString ()                            String
checkPermission(Hotel)                 boolean
checkParam()                           boolean
load()                                 void
password                               String
lastName                               String
email                                  String
phoneNumber                            String
firstName                              String
role                                   UserRole
id                                     Integer
dateOfBirth                            Date
```

```
tb_user
id INT
phone_number VARCHAR(45)
first_name VARCHAR(45)
last_name VARCHAR(45)
email VARCHAR(45)
password VARCHAR(45)
date_of_birth DATE
role_level TINYINT
Indexes
```

7.2 Pattern 2: Data mapper

7.2.1 Description

The data mapper pattern aims to provide a layer of separation and decoupling between the domain model that powers the business logic of the system, and the database that stores a representation of objects in said domain model. The data mapper classes are responsible for making queries to the database and transforming the results of those queries into domain objects (and vice versa). The main idea is to have a collection of data mapper classes that map to different entities in the domain layer and form a data mapper or data source layer. This allows the domain layer to access data from the database without needing to know how the database works or how it is structured. It also allows changes to the database structure or changes to domain objects do not affect each other.

In this specific system, each data mapper maps one domain object with one table of the database (or two for the case of the HotelierGroup class due to the presence of an association link table). Each mapper inherits from the abstract DataMapper class, which requires every mapper to have its own implementation of common database operations (insert, update, delete, and finding by ID). This pattern was adopted to allow the mappers to work more extensively with the unit of work class. By having all mappers share command methods, and by having the additional getMapper method that returns any of the child mappers at runtime based on the input class, the unit of work is able to directly work with all current and any future mappers of the system without further modification, helping improve polymorphism and extendability. Each mapper also contains a static helper method that converts a result set into that mapper's associated domain object. Each mapper also has its own set of unique methods for its specific domain object, which are used directly by the service layer and other data mappers to accomplish more specific tasks.

The mappers are accessed by the service layer to generate domain objects that the service layer uses to conduct its business logic. Once the logic is completed those objects are passed back to the mapper in order to make appropriate changes to the database. Again, this allows for a layer of abstraction as the higher layers of the system have no information about the database or how it works, only that the data mappers can provide domain objects representing the data on it. As an example, the sequence diagram below represents a use case of accessing the User data mapper to get a list of all the users. The controller representing the API endpoint calls the service that handles the use case, which calls the data mapper to return a list of users in the form of a list of User objects.

7.2.2 Sequence Diagram

The general sequence of interactions is that services called by the controller layer would first fetch data in the form of a domain object from the database using one of the data mappers (depending on which class is required to fulfill the business logic). After the business logic has been performed using the domain object, said object is passed back to the mapper to handle any updates or insertions.

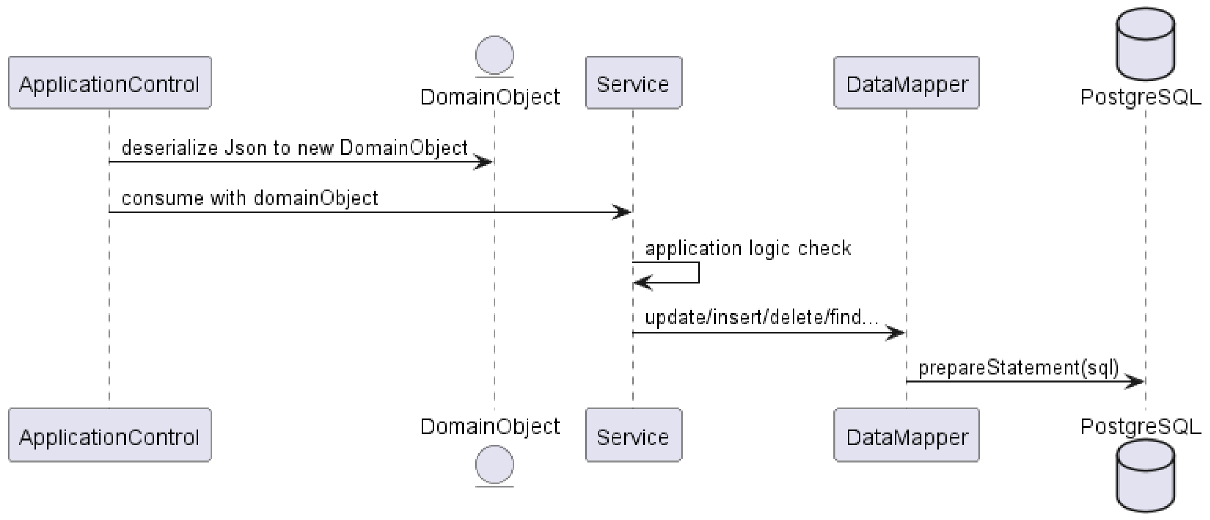


Figure 7.2: Data Mapper Sequence Diagram - 1

Below is a real example from the software system. Certain use cases and services fetched associated objects through accessing another data mapper, and combining the results of the two mappers inside said mapper. Other, services may fetch each domain object separately combine them in the service layer instead. Based on the requirements of the current situation, the service can access the hotel mapper which returns Hotel class objects, and the hotel mapper can optionally access the hotel amenity mapper to also access the associated amenities.

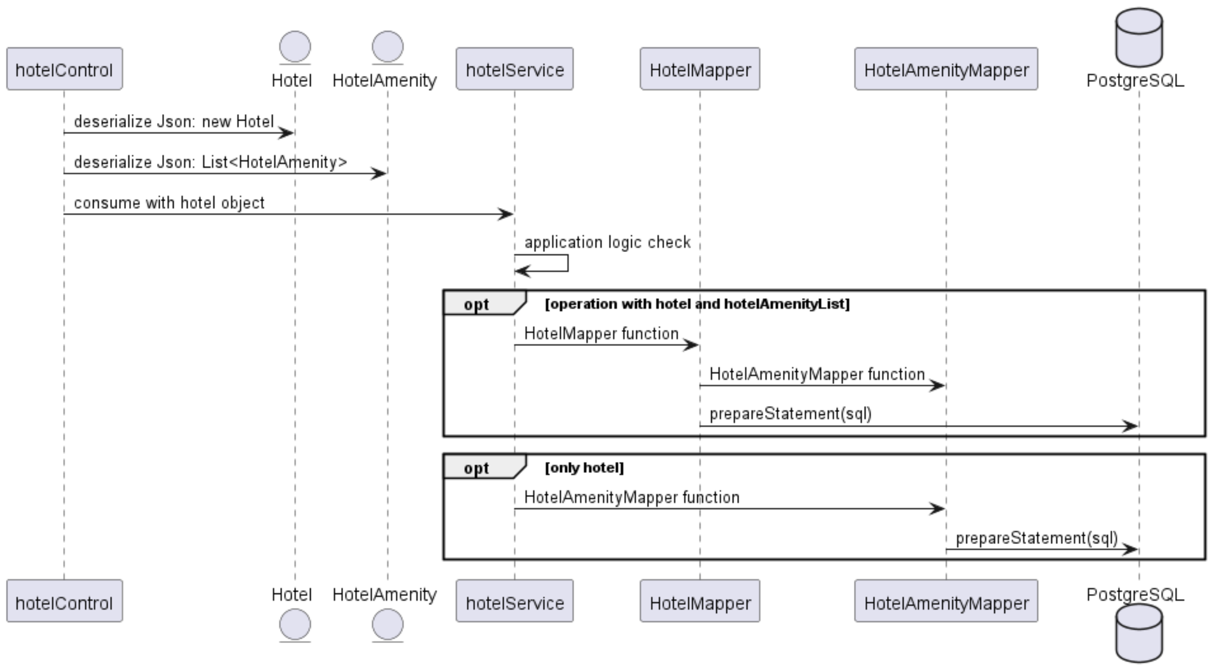

hotelControl
Hotel
HotelAmenity
hotelService
HotelMapper
HotelAmenityMapper
PostgreSQL
deserialize Json: new Hotel
deserialize Json: List<HotelAmenity>
consume with hotel object
application logic check
opt
[operation with hotel and hotelAmenityList]
HotelMapper function
HotelAmenityMapper function
prepareStatement(sql)
opt
[only hotel]
HotelAmenityMapper function
prepareStatement(sql)

Figure 7.3: Data Mapper Sequence Diagram - 2

7.2.3Entity Relationship Diagram

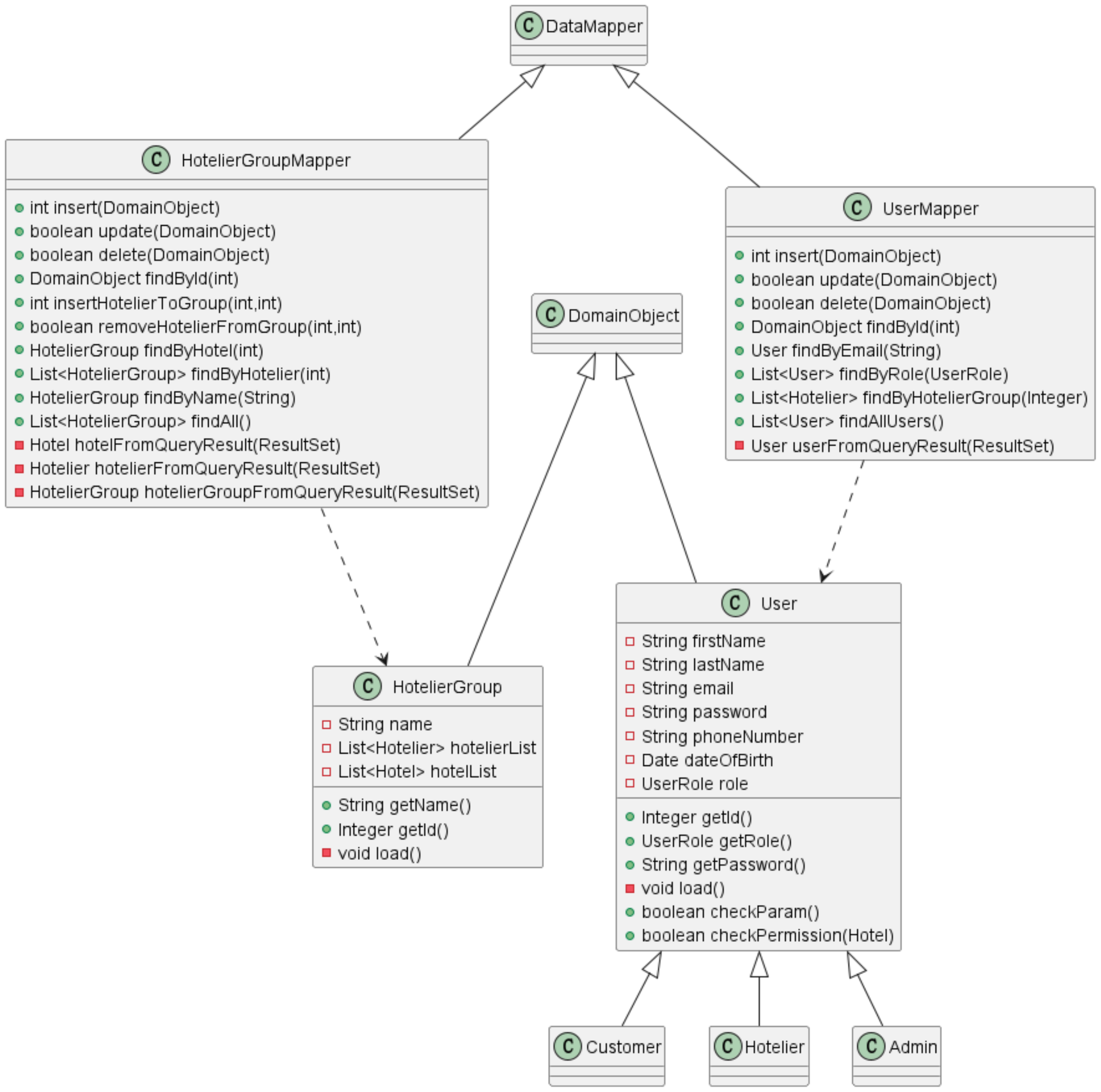

DataMapper
HotelierGroupMapper
int insert(DomainObject)
boolean update(DomainObject)
boolean delete(DomainObject)
DomainObject findById(int)
int insertHotelierToGroup(int,int)
boolean removeHotelierFromGroup(int,int)
HotelierGroup findByHotel(int)
List<HotelierGroup> findByHotelier(int)
HotelierGroup findByName(String)
List<HotelierGroup> findAll()
Hotel hotelFromQueryResult(ResultSet)
Hotelier hotelierFromQueryResult(ResultSet)
HotelierGroup hotelierGroupFromQueryResult(ResultSet)
UserMapper
int insert(DomainObject)
boolean update(DomainObject)
boolean delete(DomainObject)
DomainObject findById(int)
User findByEmail(String)
List<User> findByRole(UserRole)
List<Hotelier> findByHotelierGroup(Integer)
List<User> findAllUsers()
User userFromQueryResult(ResultSet)
DomainObject
HotelierGroup
String name
List<Hotelier> hotelierList
List<Hotel> hotelList
String getName()
Integer getId()
void load()
User
String firstName
String lastName
String email
String password
String phoneNumber
Date dateOfBirth
UserRole role
Integer getId()
UserRole getRole()
String getPassword()
void load()
boolean checkParam()
boolean checkPermission(Hotel)
Customer
Hotelier
Admin

Figure 7.5: User Mapper

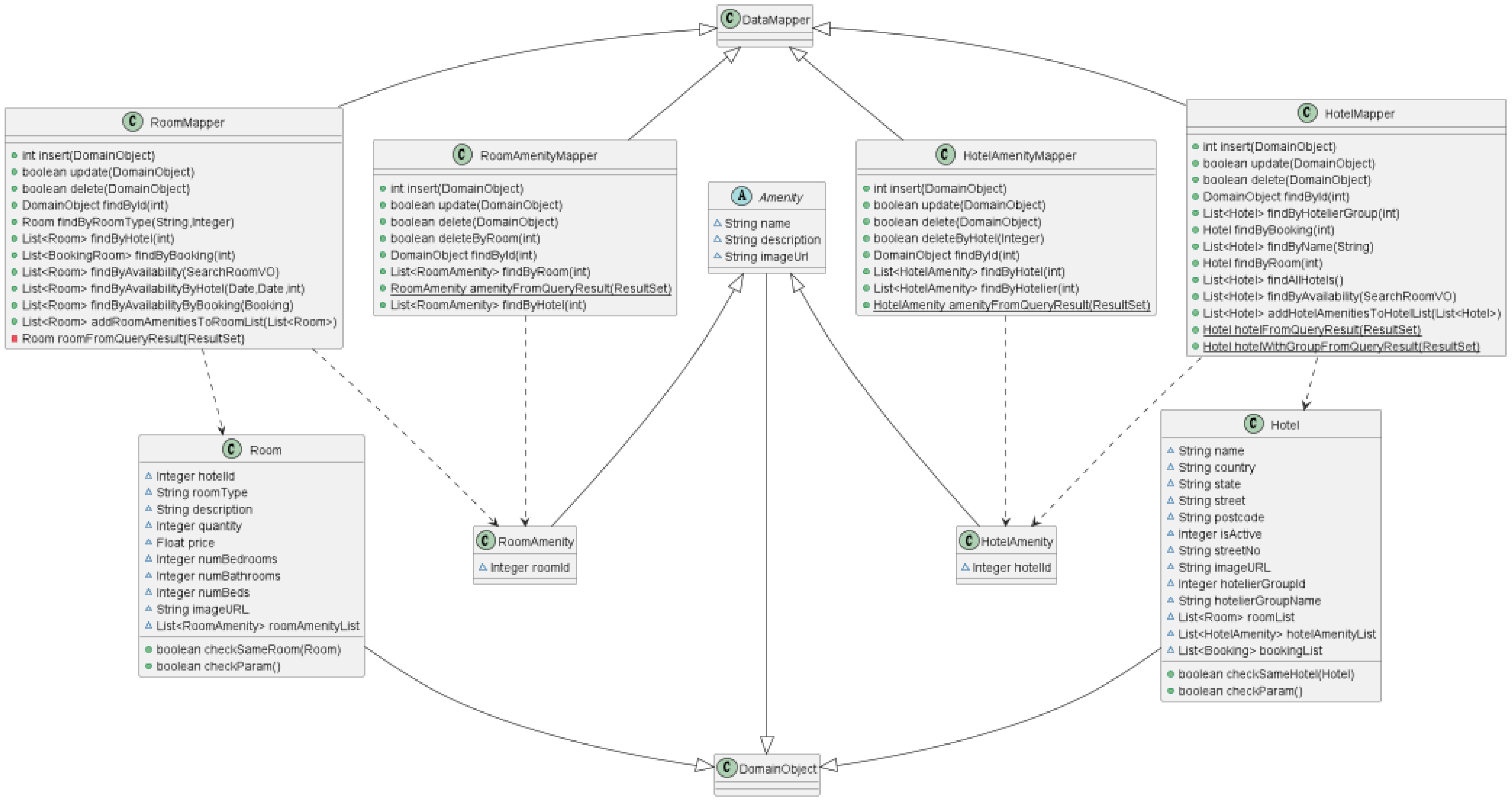

DataMapper
RoomMapper
int insert(DomainObject)
boolean update(DomainObject)
boolean delete(DomainObject)
DomainObject findById(int)
Room findByRoomType(String,Integer)
List<Room> findByHotel(int)
List<BookingRoom> findByBooking(int)
List<Room> findByAvailability(SearchRoomVO)
List<Room> findByAvailabilityByHotel(Date,Date,int)
List<Room> findByAvailabilityByBooking(Booking)
List<Room> addRoomAmenitiesToRoomList(List<Room>)
Room roomFromQueryResult(ResultSet)
RoomAmenityMapper
int insert(DomainObject)
boolean update(DomainObject)
boolean delete(DomainObject)
boolean deleteByRoom(int)
DomainObject findById(int)
List<RoomAmenity> findByRoom(int)
RoomAmenity amenityFromQueryResult(ResultSet)
List<RoomAmenity> findByHotel(int)
Amenity
String name
String description
String imageUrl
HotelAmenityMapper
int insert(DomainObject)
boolean update(DomainObject)
boolean delete(DomainObject)
boolean deleteByHotel(Integer)
DomainObject findById(int)
List<HotelAmenity> findByHotel(int)
List<HotelAmenity> findByHotelier(int)
HotelAmenity amenityFromQueryResult(ResultSet)
HotelMapper
int insert(DomainObject)
boolean update(DomainObject)
boolean delete(DomainObject)
DomainObject findById(int)
List<Hotel> findByHotelierGroup(int)
Hotel findByBooking(int)
List<Hotel> findByName(String)
Hotel findByRoom(int)
List<Hotel> findAllHotels()
List<Hotel> findByAvailability(SearchRoomVO)
List<Hotel> addHotelAmenitiesToHotelList(List<Hotel>)
Hotel hotelFromQueryResult(ResultSet)
Hotel hotelWithGroupFromQueryResult(ResultSet)
Room
Integer hotelId
String roomType
String description
Integer quantity
Float price
Integer numBedrooms
Integer numBathrooms
Integer numBeds
String imageURL
List<RoomAmenity> roomAmenityList
boolean checkSameRoom(Room)
boolean checkParam()
RoomAmenity
Integer roomId
HotelAmenity
Integer hotelId
Hotel
String name
String country
String state
String street
String postcode
Integer isActive
String streetNo
String imageURL
Integer hotelierGroupId
String hotelierGroupName
List<Room> roomList
List<HotelAmenity> hotelAmenityList
List<Booking> bookingList
boolean checkSameHotel(Hotel)
boolean checkParam()
DomainObject

Figure 7.6: Hotel and Room mapper

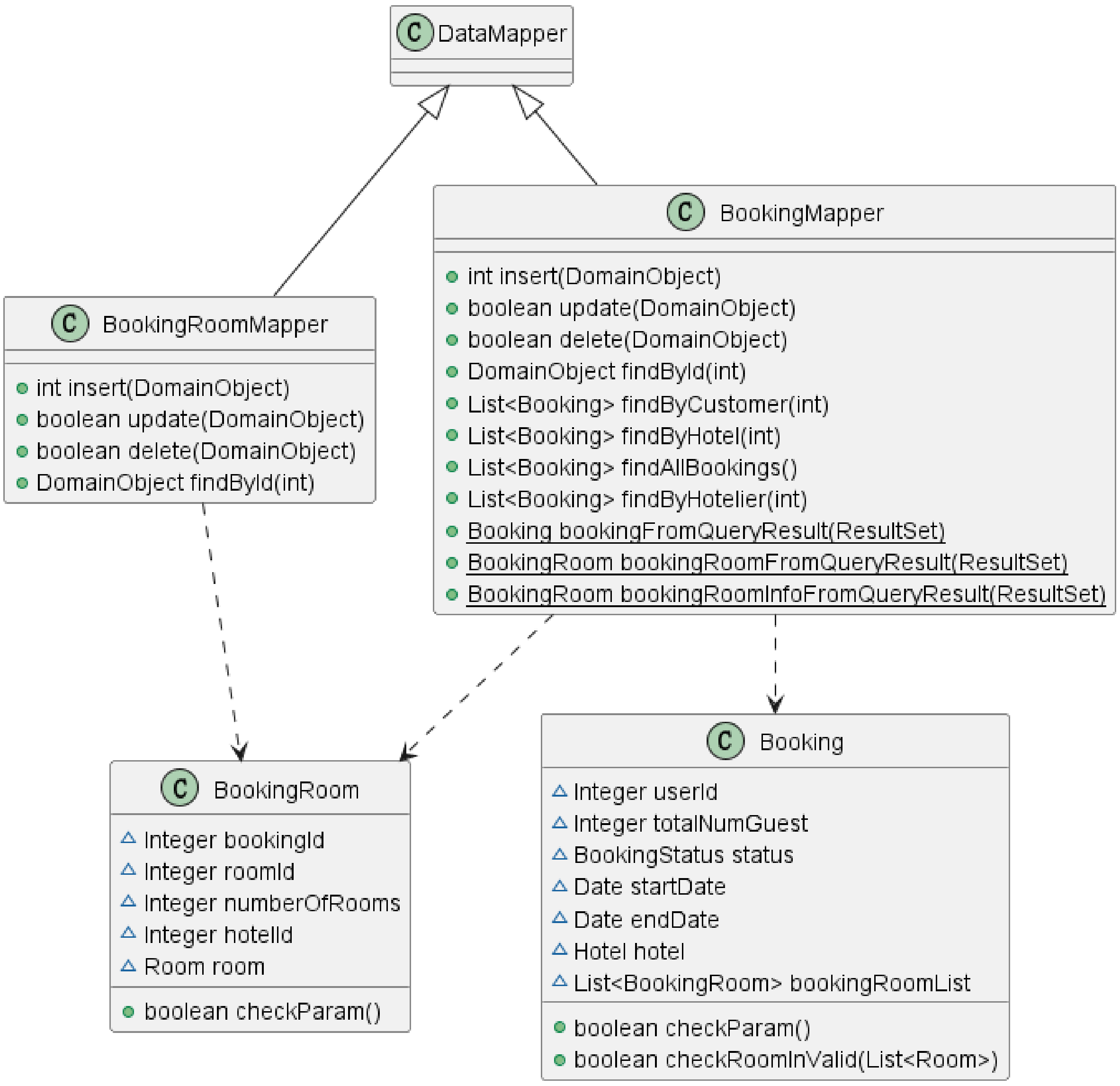

DataMapper
BookingMapper
int insert(DomainObject)
boolean update(DomainObject)
boolean delete(DomainObject)
DomainObject findById(int)
List<Booking> findByCustomer(int)
List<Booking> findByHotel(int)
List<Booking> findAllBookings()
List<Booking> findByHotelier(int)
Booking bookingFromQueryResult(ResultSet)
BookingRoom bookingRoomFromQueryResult(ResultSet)
BookingRoom bookingRoomInfoFromQueryResult(ResultSet)
BookingRoomMapper
int insert(DomainObject)
boolean update(DomainObject)
boolean delete(DomainObject)
DomainObject findById(int)
BookingRoom
Integer bookingId
Integer roomId
Integer numberOfRooms
Integer hotelId
Room room
boolean checkParam()
Booking
Integer userId
Integer totalNumGuest
BookingStatus status
Date startDate
Date endDate
Hotel hotel
List<BookingRoom> bookingRoomList
boolean checkParam()
boolean checkRoomInValid(List<Room>)

Figure 7.7: Booking mapper

7.3 Pattern 3: Unit of work

7.3.1Description

To keep the consistency of each **object read and modified by a single business transaction** that can affect the database, we decided to implement the Unit of Work design pattern in order **to maintain a list of such objects** and organise the writing out of changes. Additionally, the implementation of this design pattern will also allow us **to prepare the Hotel Booking System for any future resolution of concurrency problem**s.

In this regard, we considered applying the Unit of Work design pattern for the following business transactions:

- **Customer** associated business transactions:
    - **Book a Hotel**
        - Endpoint called: /BookHotel
            - This business transaction requires the implementation of Unity of Work because, every time a Customer BookHotel it will trigger the creation of a new row in the tb_booking table, which will also require to add new information into the tb_booking_room which keeps the many to many relationships between a Booking and a Room Object.
    - **Modify a booking** which includes:
        - Adding more people to an existing booking
            - Endpoint called: /ModifyBookings
                - When a customer decides to add more people to an existing booking, we need to update the booking table and also ensure that there is enough room space in the case that others users will also book the same types of room. Therefore, the data of this business transaction is vital to be inserted as a single unit when going back to the database.
        - Changing dates
            - Endpoint called: /ModifyBookings
                - When a customer wants to change the check-in and check-out dates of an existing booking, we need to update the booking table and also need to ensure during the new check-in/our dates, the rooms are still available. Therefore, the data of this business transaction is important to be inserted as a single unit.

- **Hotelier** associated business transactions:
    - **Create hotel** for a group
        - Endpoint called: /CreateHotel
            - When a hotelier wants to create a hotel, we need to insert the hotel's information, such as name, address, hotelier group name into the hotel table and amenities into hotel amenity table. In this transaction, two tables will be updated; therefore, the data of this business transaction is vital to insert as a single unit.
    - **Create hotel rooms** for a hotel
        - Endpoint called: /CreateRoom

      - When a hotelier wants to create rooms for a hotel, we need to insert rooms' information, such as name, number of beds, number of bedrooms into the room table and room amenities into room amenity table. In this transaction, two tables will be updated, therefore, the data of this transaction is important to be inserted as a single unit.
  - **Modify a hotel**:
    - Endpoint: /ModifyHotel
      - When a hotelier decides to modify a hotel, we need to update the hotel table and hotel amenity table since not only the normal information of a hotel is able to be modified (e.g. name, street address, postcode, etc), but also hotel amenities. These two tables will be updated at the same time, therefore, the data of this business transaction is vital to be inserted as a single unit.

Additionally, to ensure that the Unit of Work works as expected, there were some technical considerations we had to make in order to let our Unit of Work implementation know what objects it should keep track of. For this, **we decided to use the "Caller registration"** approach, since its intuitive implementation lets us decide which object to register with the Unit of Work for any type of change. This decision helped us to constantly and explicitly consider the participation of the Unit of Work to register or not an object.

7.3.2Sequence Diagram

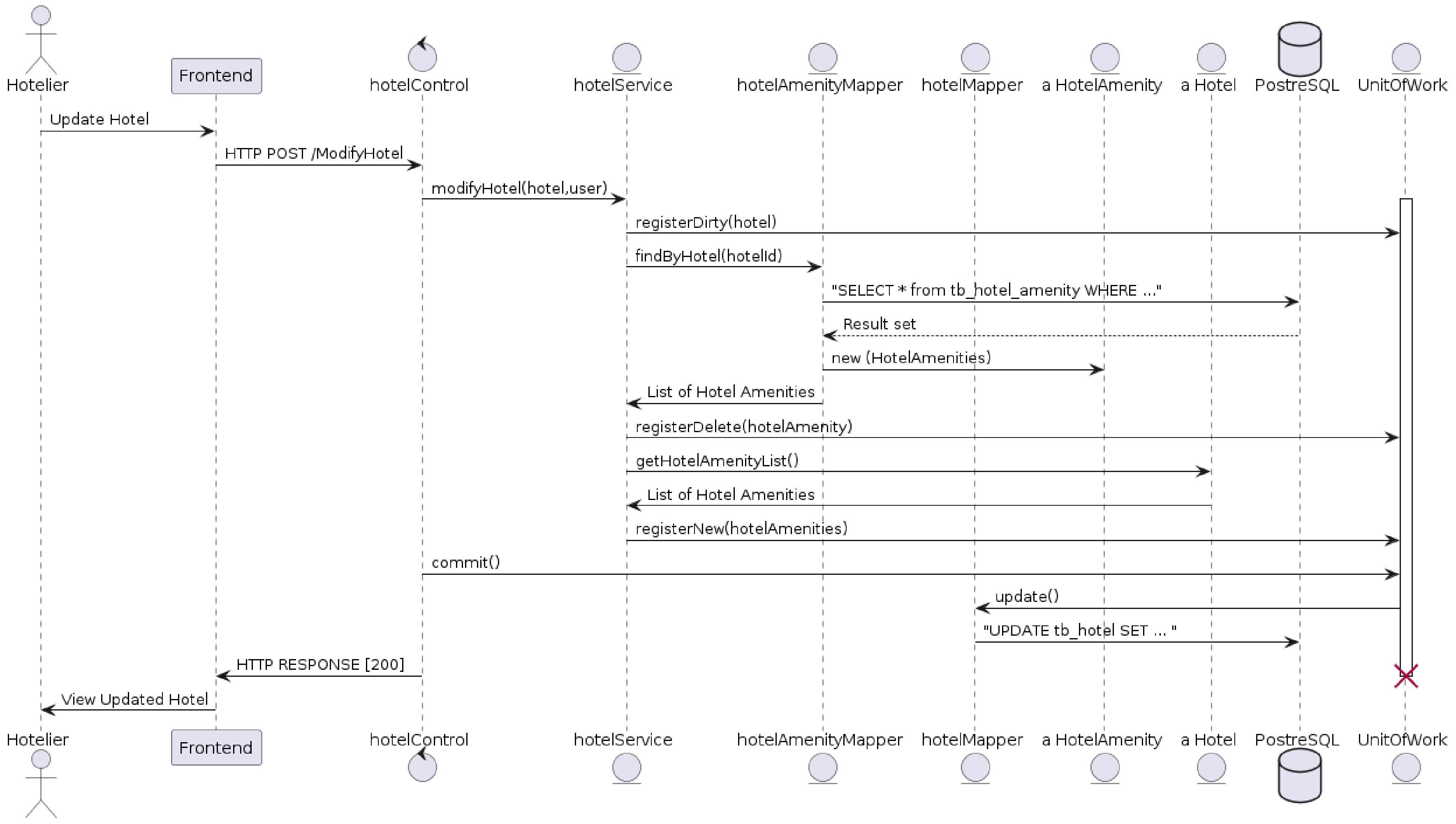


Figure 7.8: Unit of Work Sequence Diagram

7.4 Pattern 4: Lazy load

7.4.1Description

Sometimes the domain layer of a system requires an in-memory representation of an object from the database, without actually requiring any of the data associated with that object. In these types of situations, querying the data is not necessary and even inefficient as it is never used. However, since some form of the object (sometimes referred to as a dummy version of the object) is still required, we may not have a choice. This is where the lazy load pattern, sometimes referred to as the inverse of the unit of work pattern, can be useful.

When utilising the lazy load pattern, the domain objects themselves are not actually initialised with any data of the object it is supposed to represent. Instead, it knows how to initialise its own data, and will only do so if another object actually tries to access said data. If the data is never attempted to be accessed, then the data will simply never be fetched from the database and never initialised into the dummy object. This can be very beneficial for increasing performance when the majority of data may not be needed to complete a certain function. The lazy load pattern is typically embedded directly into the getter methods of the objects in the domain layer, so higher layers that utilise the domain objects such as the service layer do not know the underlying domain objects and even use lazy loading.

Lazy loading can be implemented in numerous ways, and for this system, the “Ghost” implementation was selected. The Ghost implementation first initialises the entire object without any data, but will then initialise all the attributes of the objects if any of them are accessed, not just the attribute that was accessed. The reason the ghost implementation was selected was because it minimises calls to the database, compared to the lazy initialization method which initialises each attribute separately and therefore each first accessing of each attribute requires its own call to the database. Minimising database access was very important for this system given the high amount of data required just to complete a single use case.

The high amount of data also leads to one of the main drawbacks of lazy load for this system: **ripple loading**, where many more calls to the database than required are made due to fetching data for each domain object individually. For example, when fetching a list of bookings, said bookings must also contain all the information for the bookings associated hotel, as well as the rooms that were booked.

Using lazy loading, in this case, would mean that for every booking queried, each booking’s hotel and booked rooms would need to be queried separately, which could result in hundreds if not thousands of database calls depending on how many bookings were returned. This would drastically lower the performance of the system, which defeats the purpose of the lazy load. This is why lazy load was mostly avoided for most of the domain objects of the system and only used for domain objects where ripple loading would not be an issue, such as with the User class and HotelierGroup class.

7.4.2Sequence Diagram

The sequence diagram shows how service layers access domain object getters, those domain objects will check if the data is actually loaded yet, then proceed to load the data if it is not using the lazy load method, which fetches the data using the appropriate data mapper.

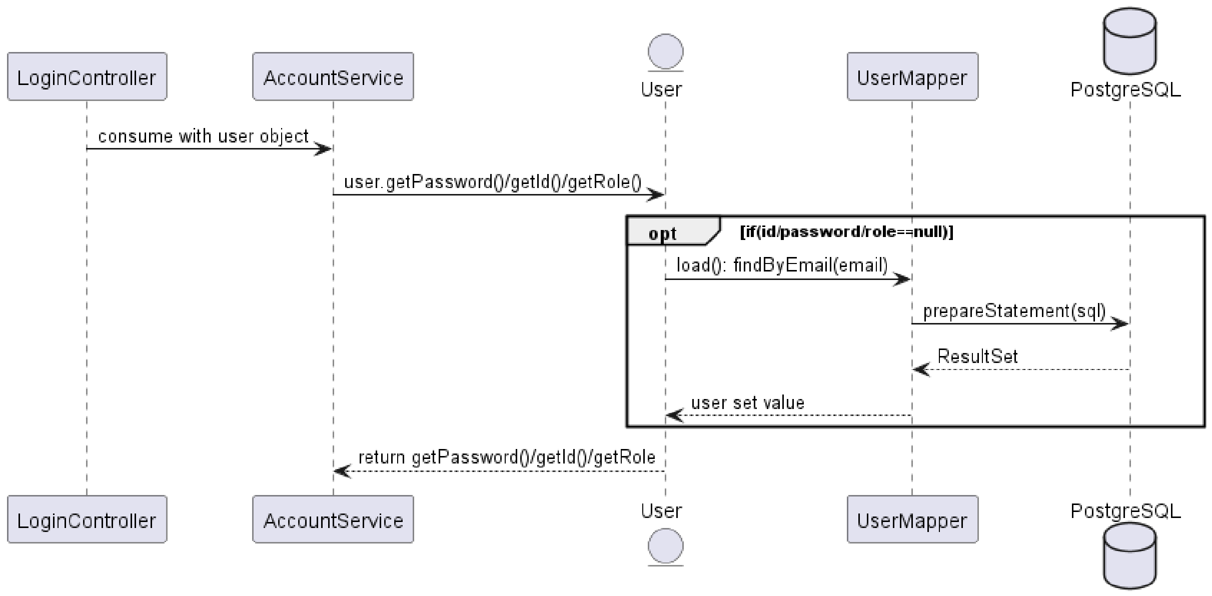

LoginController
AccountService
User
UserMapper
PostgreSQL
consume with user object
user.getPassword()/getId()/getRole()
opt
[if(id/password/role==null)]
load(): findByEmail(email)
prepareStatement(sql)
ResultSet
user set value
return getPassword()/getId()/getRole

Figure 7.9: Lazy Load Sequence Diagram

7.4.3Entity Relationship Diagram

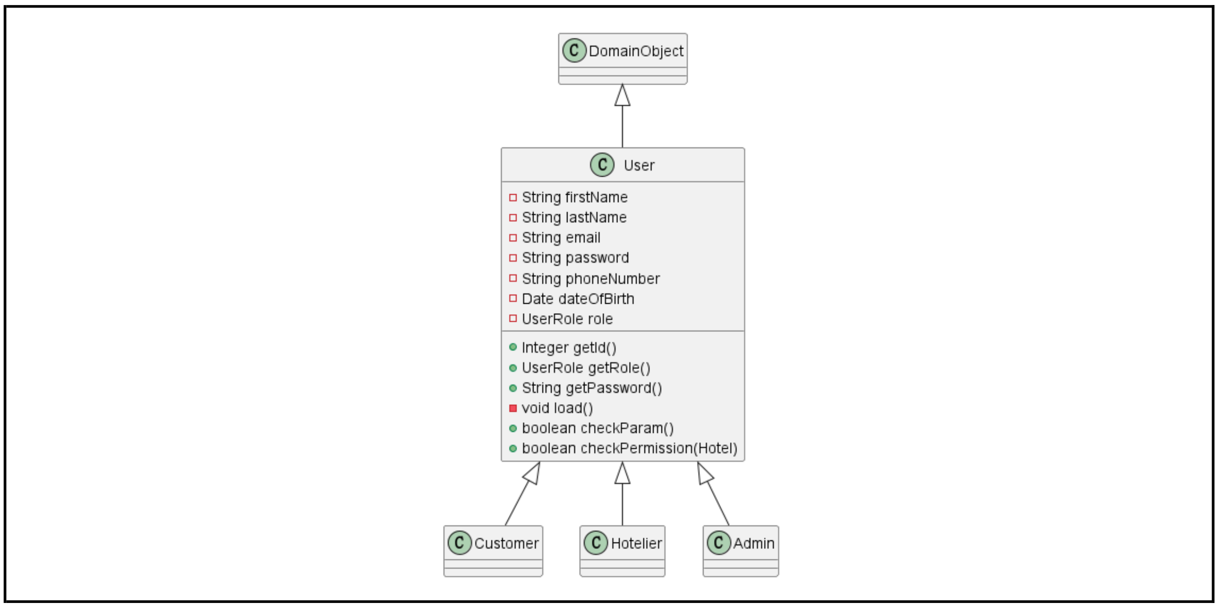
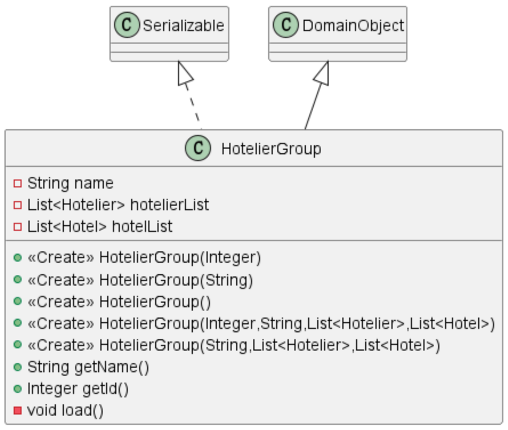


Table 7.4.1 Code Block

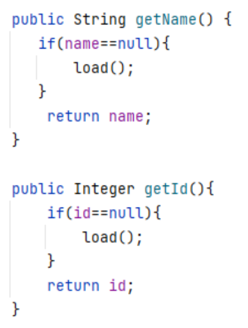
The below figures represent the Ghost implementation of lazy load used within the software system. On the left, the modified getters are shown that instead of directly returning attributes first check if those attributes have been initialised. On the right are the actual contents of the load method called by the getters in order to initialise the data for the object.

```java
public Integer getId() {
    if(id==null){
        load();
    }
    return super.getId();
}

public UserRole getRole() {
    if(role==null){
        load();
    }
    return role;
}

public String getPassword() {
    if(password==null){
        load();
    }
    return password;
}
```

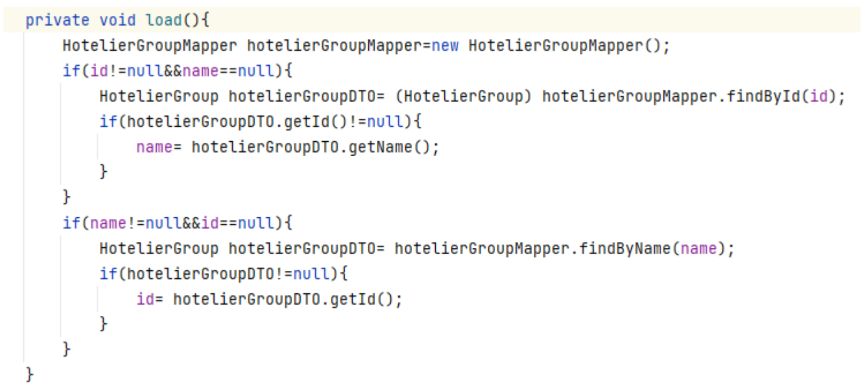
```java
private void load() {
    UserMapper userMapper=new UserMapper();
    User userDTO=userMapper.findByEmail(email);
    if(userDTO!=null){
        if(this.id==null){
            this.id=userDTO.getId();
        }
        if(this.role==null){
            this.role=userDTO.getRole();
        }
        if(this.password==null){
            this.password=userDTO.getPassword();
        }
        this.phoneNumber=userDTO.getPhoneNumber();
        this.dateOfBirth= userDTO.getDateOfBirth()
                null:new java.sql.Date(userDTO.get
        this.lastName=userDTO.getLastName();
        this.firstName=userDTO.getFirstName();
    }
}
```



Serializable
DomainObject
HotelierGroup
String name
List<Hotelier> hotelierList
List<Hotel> hotelList
«Create» HotelierGroup(Integer)
«Create» HotelierGroup(String)
«Create» HotelierGroup()
«Create» HotelierGroup(Integer,String,List<Hotelier>,List<Hotel>)
«Create» HotelierGroup(String,List<Hotelier>,List<Hotel>)
String getName()
Integer getId()
void load()



```java
public String getName() {
   if(name==null){
       load();
   }
    return name;
}

public Integer getId(){
    if(id==null){
        load();
    }
    return id;
}
```



```java
private void load(){
    HotelierGroupMapper hotelierGroupMapper=new HotelierGroupMapper();
    if(id!=null&&name==null){
        HotelierGroup hotelierGroupDTO= (HotelierGroup) hotelierGroupMapper.findById(id);
        if(hotelierGroupDTO.getId()!=null){
            name= hotelierGroupDTO.getName();
        }
    }
    if(name!=null&&id==null){
        HotelierGroup hotelierGroupDTO= hotelierGroupMapper.findByName(name);
        if(hotelierGroupDTO!=null){
            id= hotelierGroupDTO.getId();
        }
    }
}
```

7.5 Pattern 5: Identity field

7.5.1Description

A design pattern destinated to map an in-memory object corresponds to a row in a database, there is a need to link both entities together to ensure that the corresponding object is written back to the correct rows.

For our implementation, we decided to take the following design decision related to the use of the pattern:

- Meaningful vs Meaningless keys:

  Meaningless keys are used to ensure that keys are unique between entities of the same abstraction, this way, we can reduce mitigate the risk of human error when creating Meaningful keys.

- Simple keys vs. Compound keys:

  Simple keys are used for time cost reduction in manipulating complex keys and for simplification; therefore, we can use the same code to manipulate all keys. At the database implementation level, in PostgreSQL, we decided to create keys per table by using **auto-incremented integer values from a sequence**.

  Furthermore, by using auto-generated simple keys, it will allow us to generate the next keys faster in comparison with other datatypes such as strings, which will require more complex operations than incrementing strings.

  This implementation of sequences will reduce the time to generate keys, keeping the integrity and lock of transactions for such a common task.

- Table-unique key vs Database-unique keys:

  Table-unique keys in a combination of sequences are used to reduce the future impact of concurrent transactions when multiple objects request a key simultaneously**.**

  In newer versions of PostgreSQL, the use of Identity Field as the primary key is encouraged due to its SQL compatibility support, but this data type can not specify the use of a specific table sequence (https://www.postgresql.org/message-id/YVQxL6YWXxhQwcft%40hjp.at). In order to follow the SQL standard, we decided to use Identity field types, but **following the table-unique key approach**, a new table sequence will be created for each table of the database.

7.5.2Sequence Diagram

When a user request is received through the API endpoint, the system will execute queries to the database to serve the user. After the information is retrieved from the database, the system will keep track of each row extracted from the database by its corresponding table-unique key. This will map the in-memory object to the corresponding row in the database.

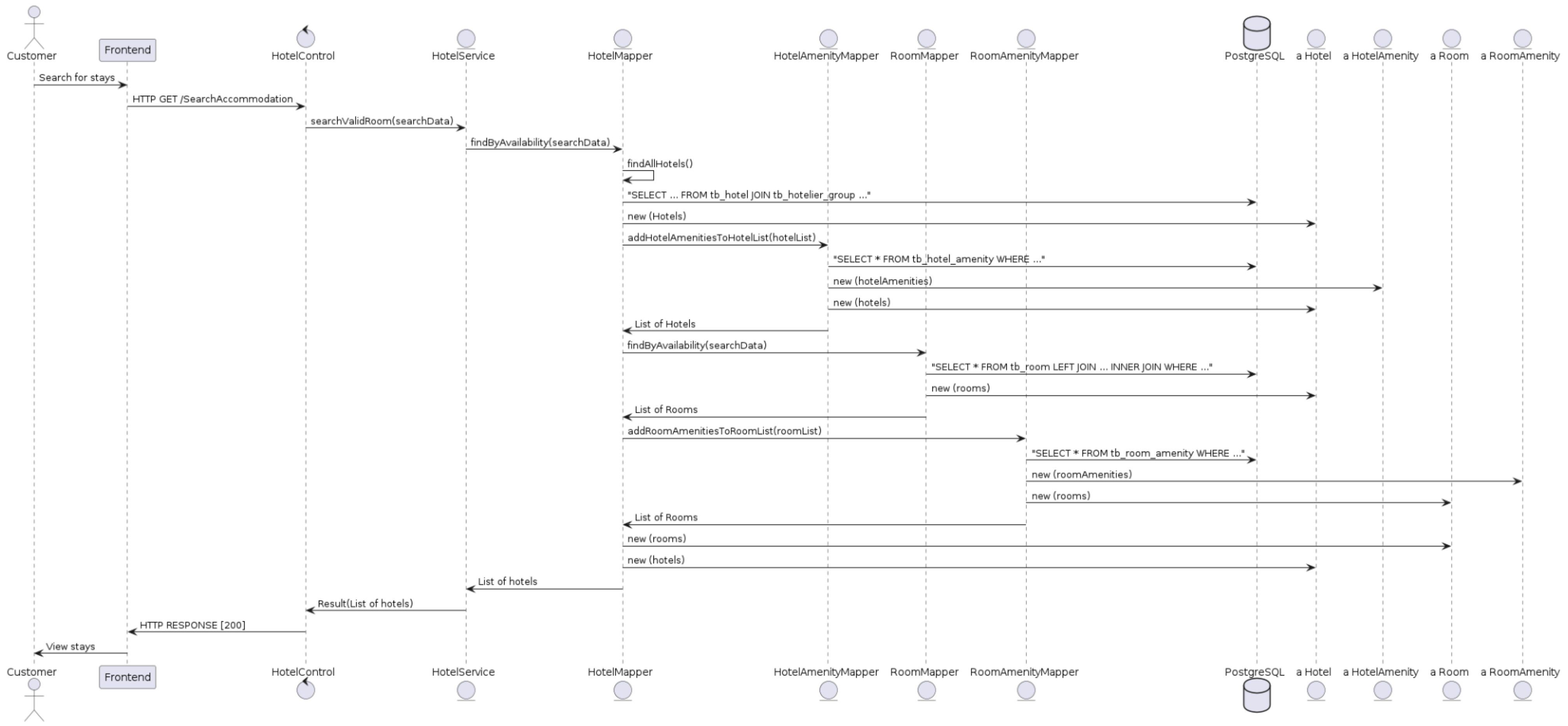
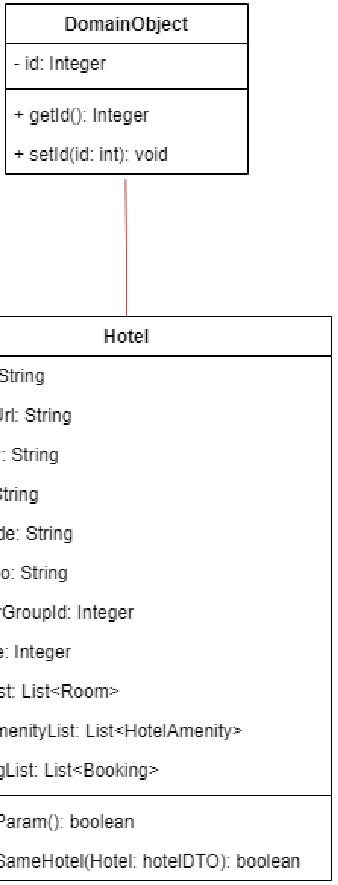
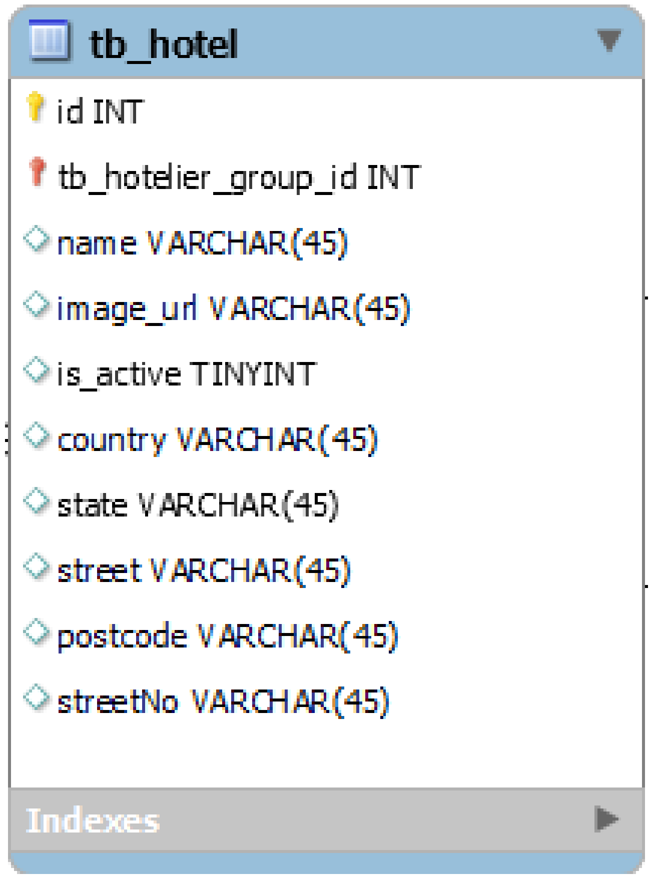


Figure 7.10: Identity Field Sequence Diagram

7.5.3Entity Relationship Diagram

Note: To improve readability, Information of the Class and Database Diagram has been omitted

Class Diagram	Database Diagram
	

7.6 Pattern 6: Foreign key mapping

7.6.1Description

After linking each domain object to its corresponding rows in database tables, through the use of the identity field pattern as explained in **section 7.5**; it will also be necessary to keep track of the references between objects in the domain layer. We will use Foreign Key Mapping to solve this problem, by using the same concepts used in the Identity Field pattern but with the addition that each object will contain references to other domain objects through its correspondence with foreign keys.

These references between objects will allow us to keep track of the same persistent relationship from the database tables, such as one-to-one, one-to-many and many-to-many (with the help of association table mapping design pattern) relationships. For instance:

- **One-to-one relationship**: In our implementation and design, each Hotel has an address within the same hotel database table. But an alternative implementation could require associating a Hotel with a particular address in a one-to-one relationship. This was considered initially by the team, as stated in **section 7.8.3**. This could have required keeping track of the one-to-one reference between a Hotel and its corresponding physical address.

- **One-to-many relationship:** Our Hotel Booking System allows a Customer to book hotels multiple times, which requires keeping track of the references of every booking to its corresponding Customer.

7.6.2Sequence Diagram

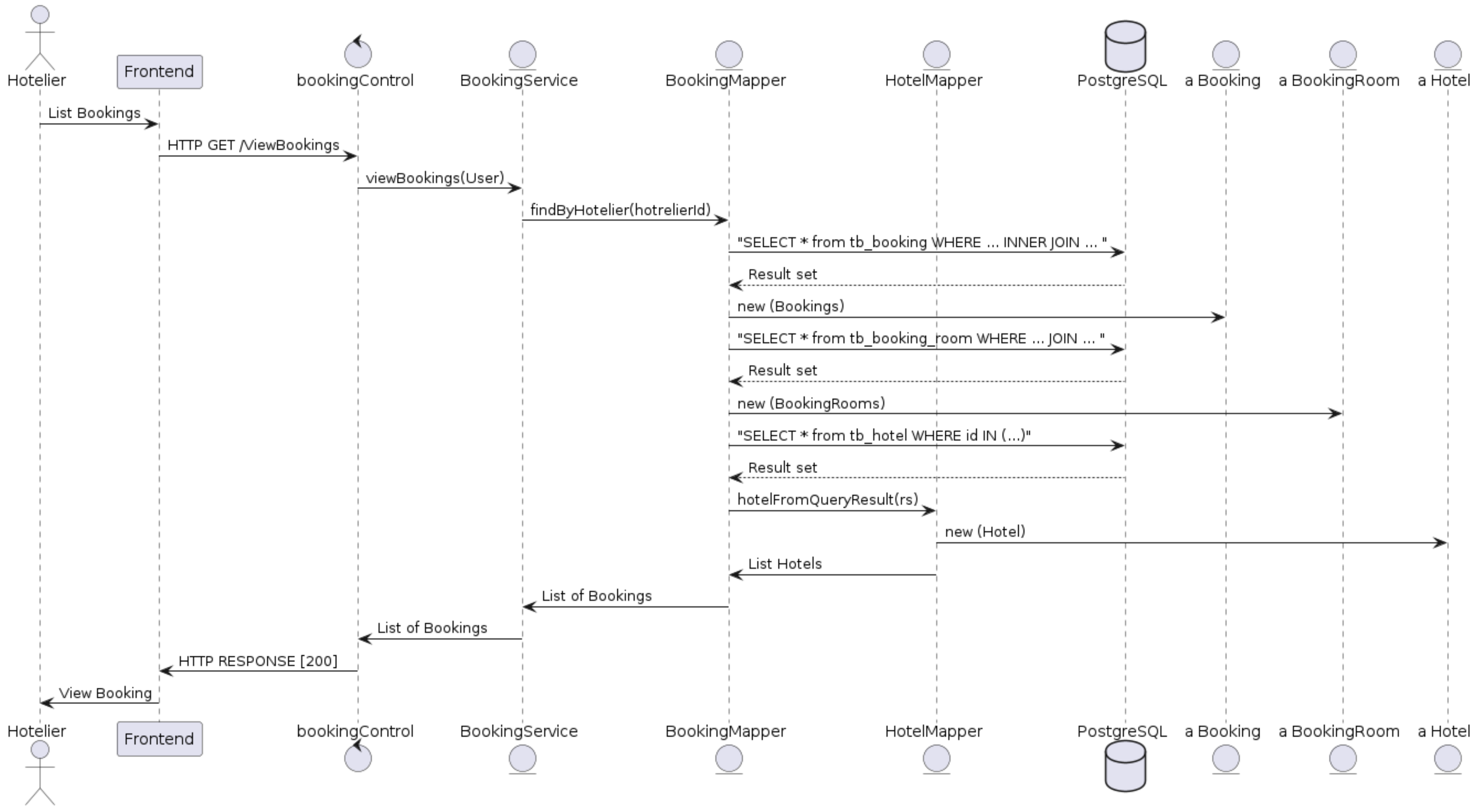
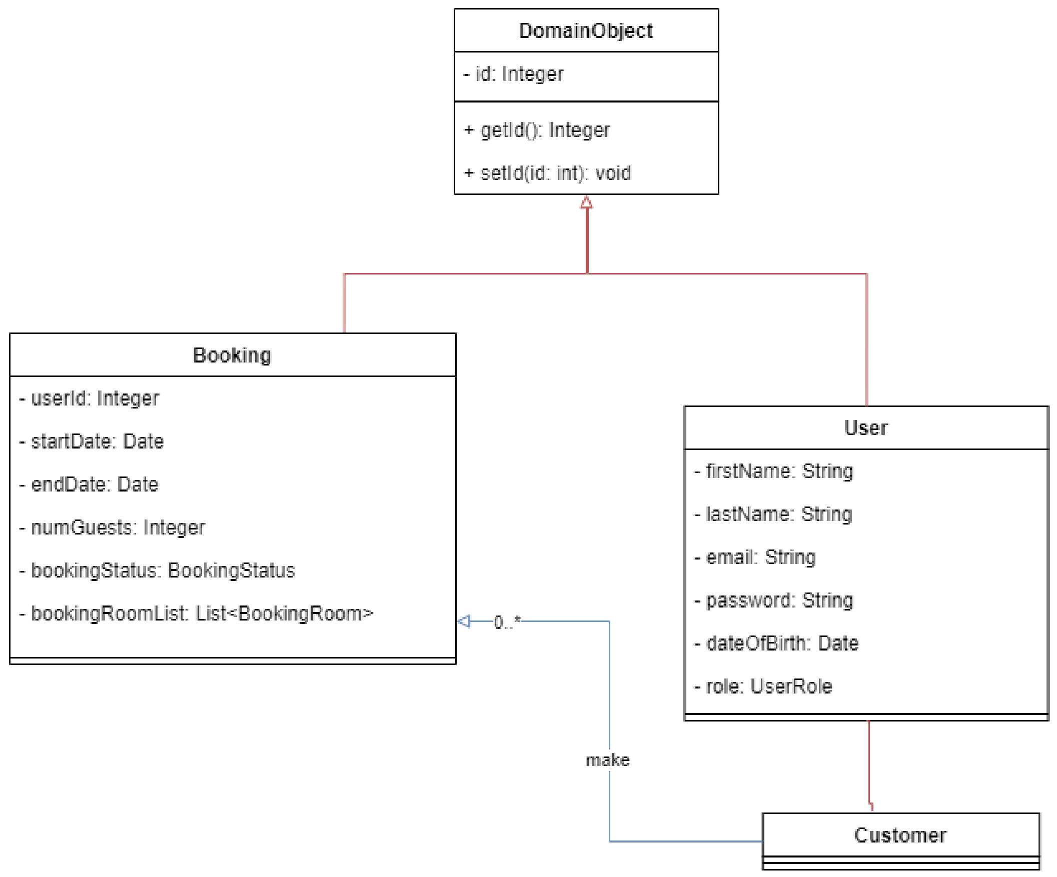
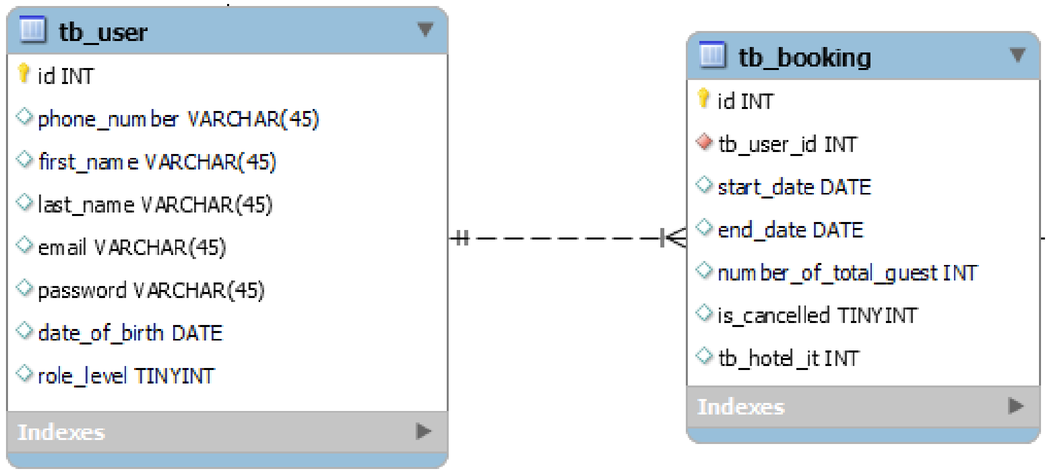


Figure 7.11: Foreign Key mapping Sequence Diagram

7.6.3Entity Relationship Diagram

Note: To improve readability, Information of the Class and Database Diagram has been omitted

Class Diagram	Database Diagram
	

7.7 Pattern 7: Association table mapping

7.7.1Description

As explained in **sections 7.5.1** and **7.6.1**, when using Object-oriented programming language, we needed to reference domain objects into a relational database; furthermore we need to keep track of its references to other objects, too. Therefore, to keep track of one domain object with its corresponding structure in the database, we used the Identity Field pattern; additionally, we decided to implement the Foreign Key pattern to keep track of associations between objects, but it was only useful for one-to-one and one-to-many mapping. Those patterns were suitable for mapping a Customer with its Bookings or a Room with its amenities.

For this reason, due to the robustness of our system, we needed **to keep track of the many-to-many relationship between objects; therefore, we decided to implement the association table mapping** in cases such as associating Hotelier with their corresponding Hotelier Group and vice-versa. In this case, we created a new table to represent the mapping called "tb_group", as shown in **section 7.7.3.**

In our implementation, The *tb_group* will hold the foreign keys of each side of the relationship: A Hotelier (User) and Hotelier Group keys. This structure allowed us to record the many-to-many relationships between the objects. Additionally, an important consideration to do here is that the *tb_group* association table has no corresponding in-memory object; instead, the relationship is been recorded using references between a User object and a HotelierGroup object as shown in **section 7.7.3**.

7.7.2Sequence Diagram

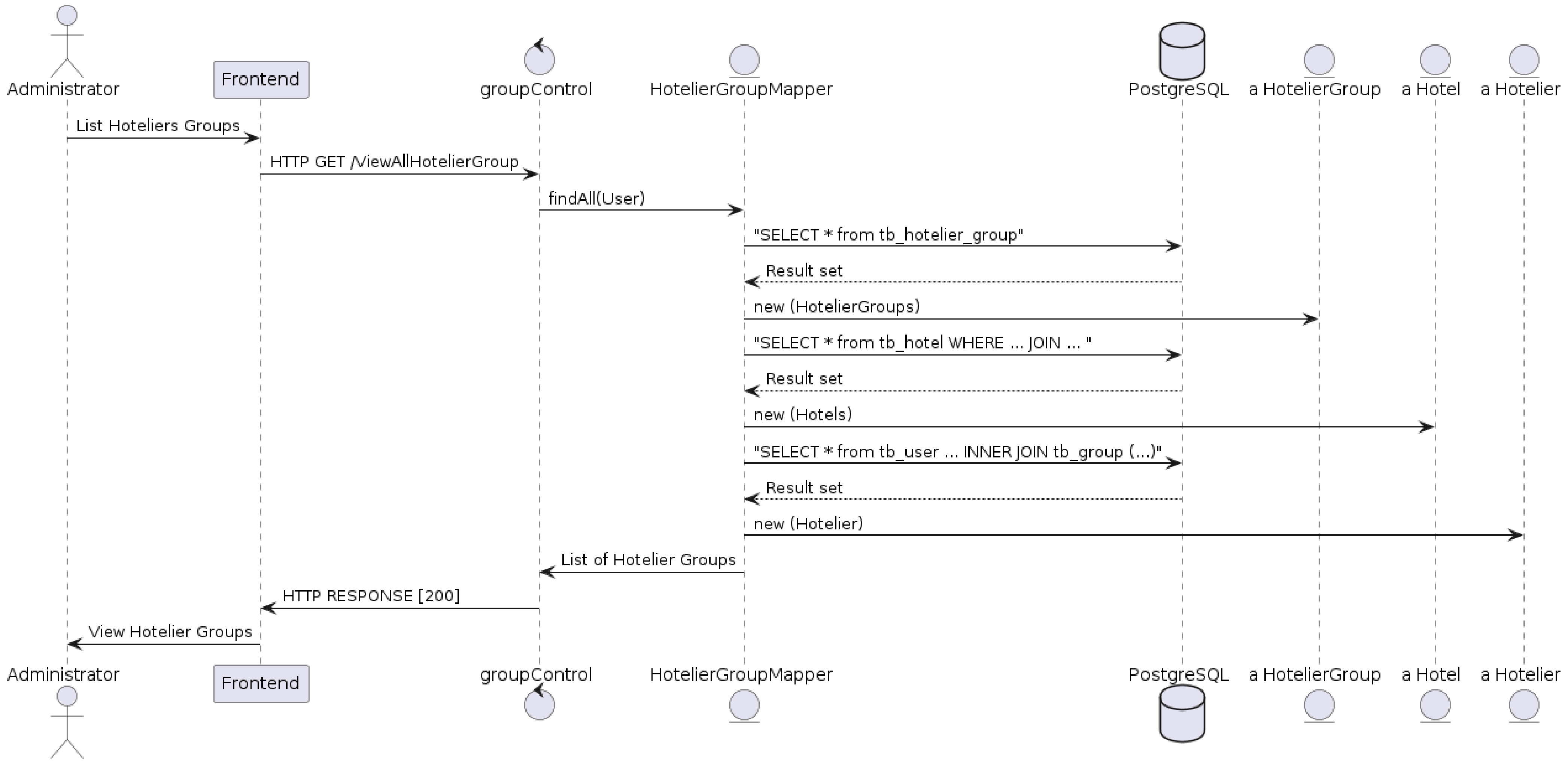
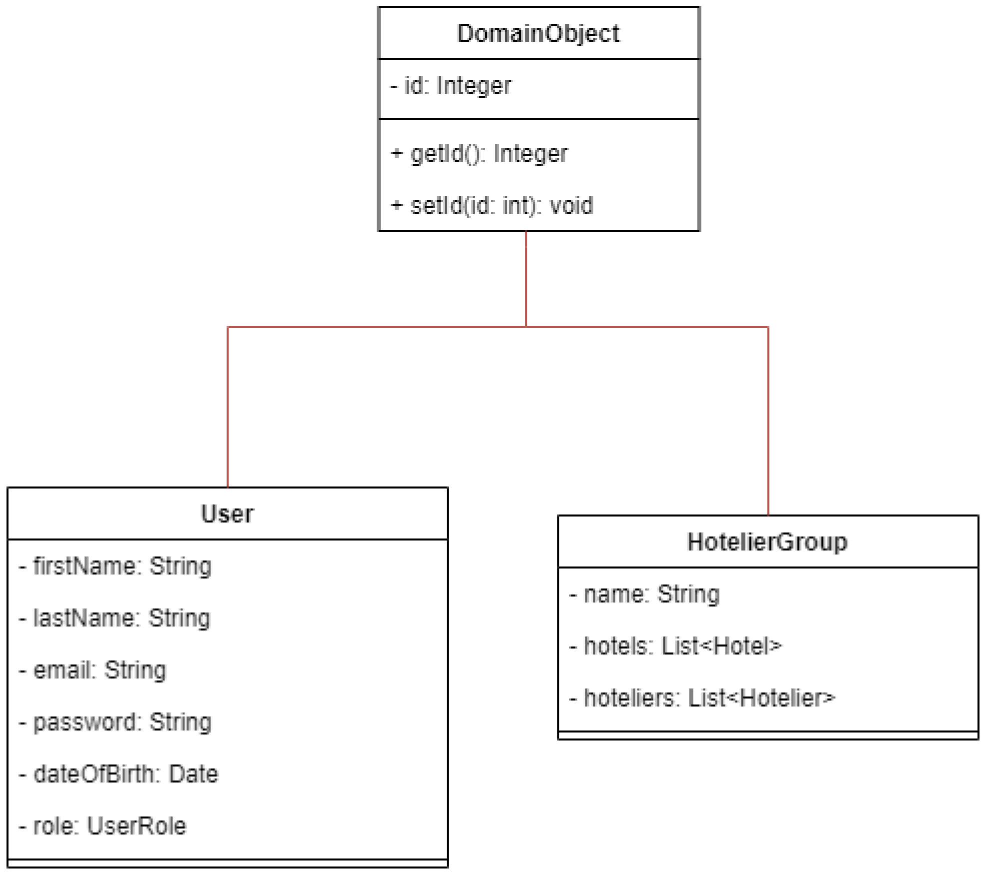
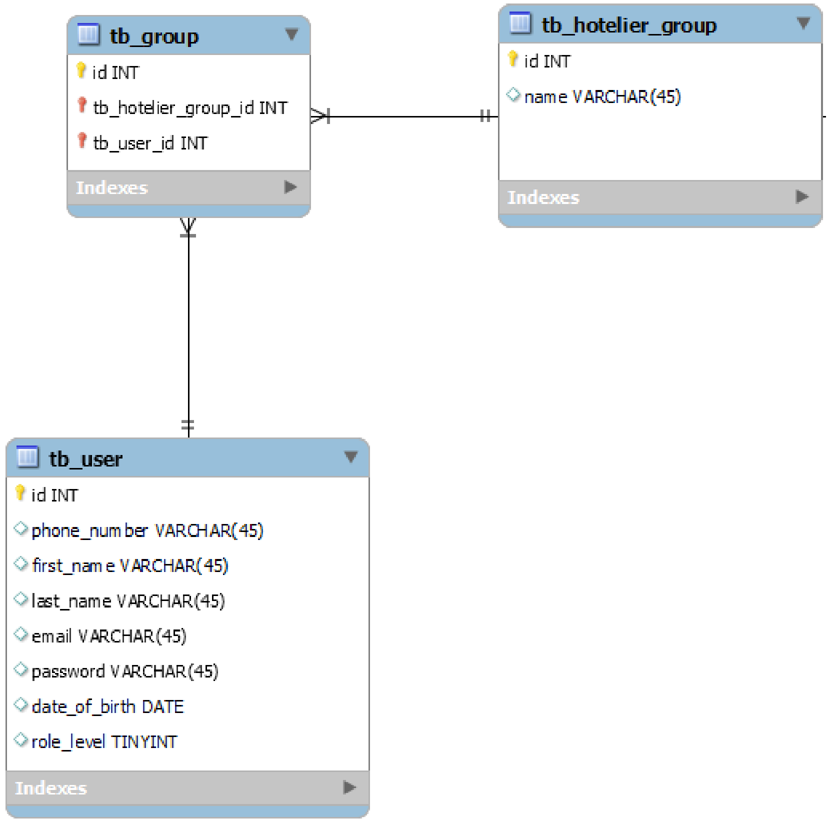


Figure 7.12: Association table mapping Sequence Diagram

7.7.3Entity Relationship Diagram

Note: To improve readability, Information of the Class and Database Diagram has been omitted

Class Diagram	Database Diagram
 DomainObject - id: Integer + getId(): Integer + setId(id: int): void User - firstName: String - lastName: String - email: String - password: String - dateOfBirth: Date - role: UserRole HotelierGroup - name: String - hotels: List<Hotel> - hoteliers: List<Hotelier>	 tb_group id INT tb_hotelier_group_id INT tb_user_id INT Indexes tb_hotelier_group id INT name VARCHAR(45) Indexes tb_user id INT phone_number VARCHAR(45) first_name VARCHAR(45) last_name VARCHAR(45) email VARCHAR(45) password VARCHAR(45) date_of_birth DATE role_level TINYINT Indexes

7.8 Pattern 8: Embedded value

7.8.1Description

We used *embedded value* to map the values of hotel address into the fields of hotels. When the hotel object is loaded or saved, the hotel's address is saved as well.

In our previous design, we separated the hotel address and hotel as two different tables in our database, and these two tables have one-to-one mapping. After refining the use cases, we found that these two objects will always be loaded and saved together. For instance, when a hotelier creates a hotel, the hotel's address is required along with the hotel's other information, such as name, hotelier group, and image URL. These data will be saved into the database at the same time. Additionally, when a customer searches hotels or a hotelier views his/her hotels, the hotel's address and hotel's information will also be loaded together and displayed to customers or hoteliers.

Based on these scenarios, we decided to group hotel and hotel addresses into a single table, as the entity relationship diagram shows.

7.8.2Sequence Diagram

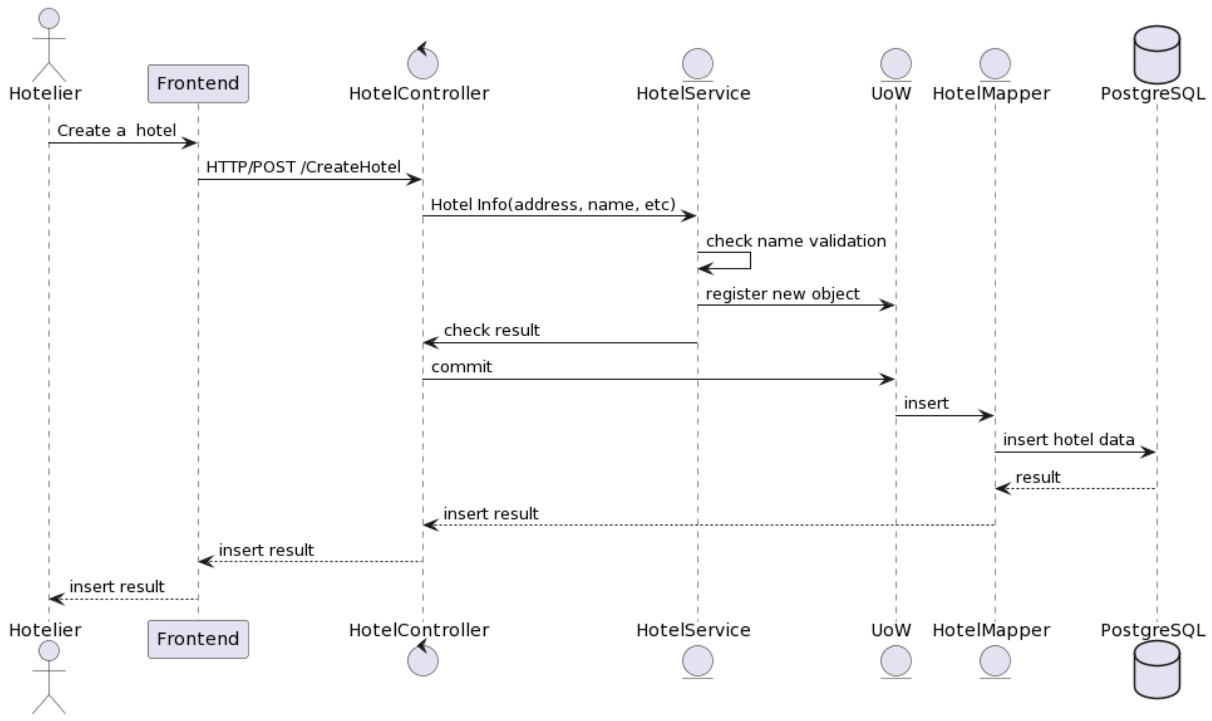
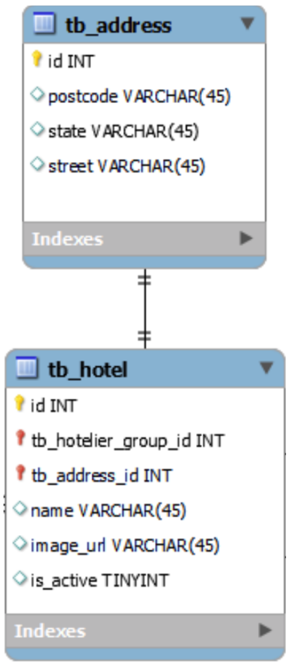
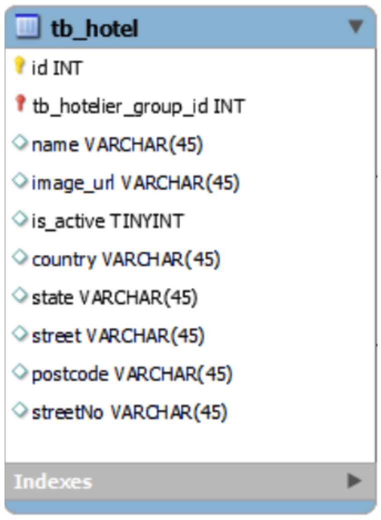


Figure 7.13: Embedded value Sequence Diagram

7.8.3Entity Relationship Diagram

Previous Database Design	**Updated Database Design with Embedded Value**
 tb_address: id INT, postcode VARCHAR(45), state VARCHAR(45), street VARCHAR(45). tb_hotel: id INT, tb_hotelier_group_id INT, tb_address_id INT, name VARCHAR(45), image_url VARCHAR(45), is_active TINYINT	 tb_hotel: id INT, tb_hotelier_group_id INT, name VARCHAR(45), image_url VARCHAR(45), is_active TINYINT, country VARCHAR(45), state VARCHAR(45), street VARCHAR(45), postcode VARCHAR(45), streetNo VARCHAR(45)
tb_address_id in hotel table points to address table	put country, state, street, streetNo, and postcode into single hotel table.

7.9 Pattern 9.1: Single table inheritance

7.9.1Description

The inheritance patterns help us to map the fields in an inheritance class hierarchy to a relational database. These are necessary patterns to consider due to the aspect that relational databases don't support inheritance. For this, we used an inheritance pattern that allows us to map a database structure to the objects in an inheritance structure.

We decided to use **Single Table Inheritance** to minimise the code needed to load and save data back to the database. The Single table inheritance is being specifically used to help us represent the inheritance relationship between our User parent class and its children: Customer, Admin and Hotelier.

Furthermore, this design pattern helped structure every class in the inheritance hierarchy as a single table in the relational database, as **section 7.9.3** shows. In this sense, we needed to create a column called *role_level* in the database that helped us to recognise the role of an Object that persisted in the database, following a simple rule:

- Role 1: Customer
- Role 2: Hotelier
- Role 3: Admin

Due to the use of this pattern, we could take advantage of the following characteristics:

- In case of a modification in any of the classes involved in the inheritance hierarchy, we will need to only to modify a single table on the database.
- When fetching data from the database, will not be necessary to create complex queries such as joins queries to retrieve the required data.

Additionally, as a team, we considered the mitigate the trade-offs of using this pattern:

- This pattern could lead to having multiple columns which could be relevant or not, but in our implementation, we strictly limited the number of columns to the minimum number required to avoid any confusion.
- To avoid the number of possible empty spaces, we decided only to add one column called role_level to identify the type of User, using the platform.
- We were aware that this table could increase in size quickly and end up being too large, but for the purpose of this educational project, and due to its size, this was not a characteristic to consider at this phase of the product.

7.9.2Sequence Diagram

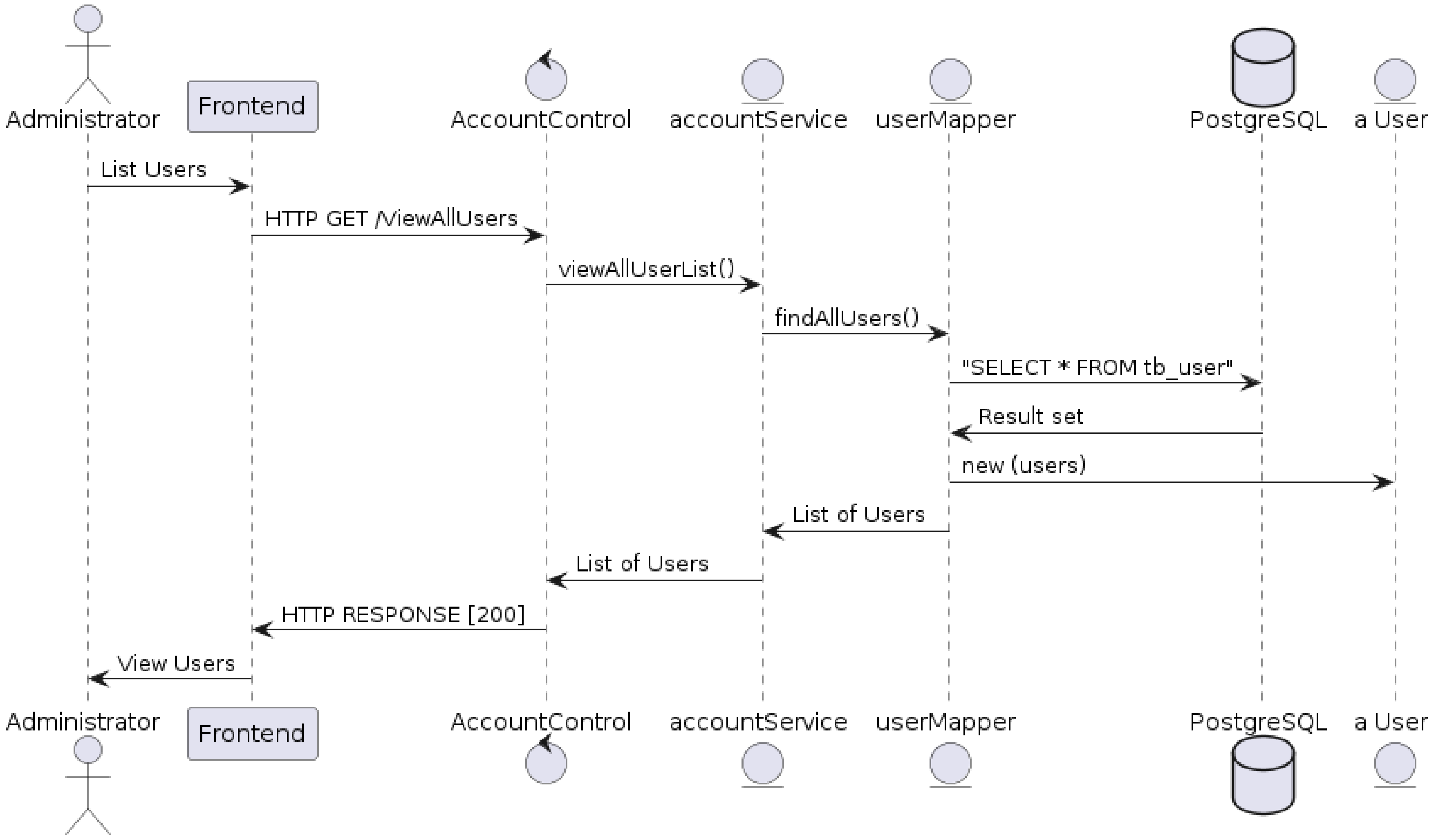
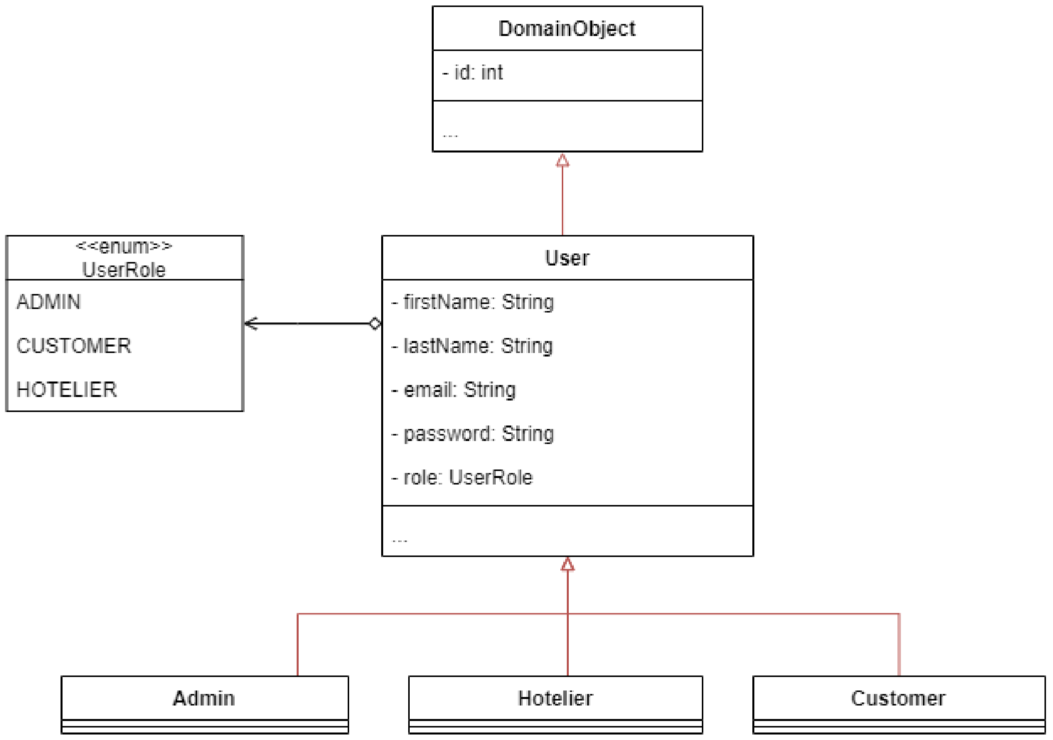
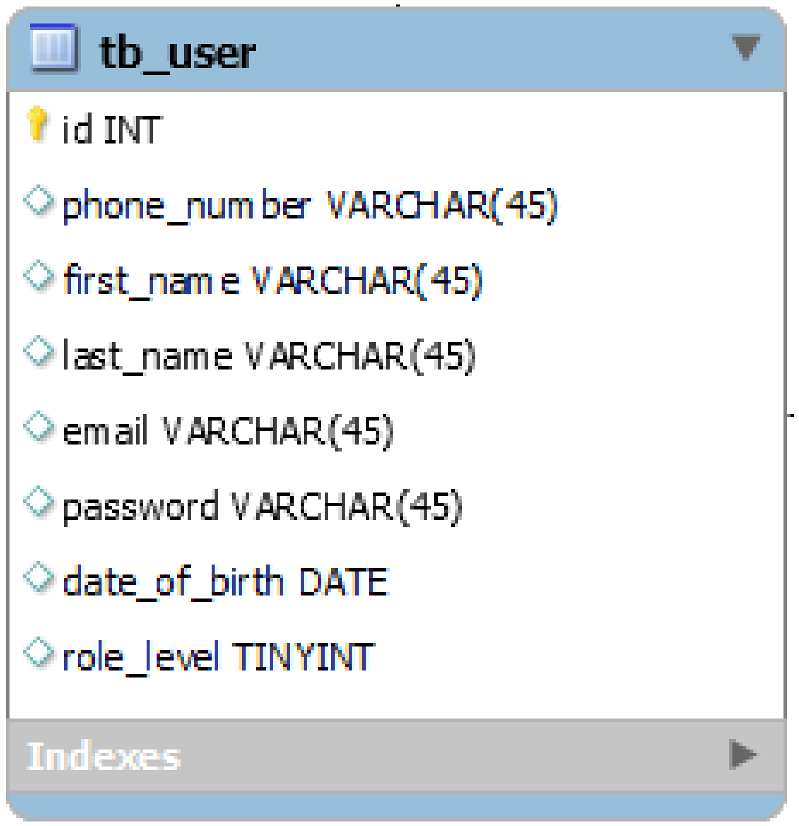


Figure 7.14: Single table inheritance Sequence diagram

7.9.3Entity Relationship Diagram

Note: To improve readability, Information of the Class and Database Diagram has been omitted

Class Diagram	Database Diagram
 DomainObject - id: int ... <<enum>> UserRole ADMIN CUSTOMER HOTELIER User - firstName: String - lastName: String - email: String - password: String - role: UserRole ... Admin Hotelier Customer	 tb_user id INT phone_number VARCHAR(45) first_name VARCHAR(45) last_name VARCHAR(45) email VARCHAR(45) password VARCHAR(45) date_of_birth DATE role_level TINYINT Indexes

7.10 Pattern 9.2: Concrete Table Inheritance

7.10.1 Description

In addition of the Single Table Inheritance pattern, the team also decided to implement the concrete table inheritance into the system by representing the inheritance hierarchy between the Amenity Parent class and its children: HotelAmenity and RoomAmenity, as shown in **section 7.10.3**.

The use of this design pattern helped us to use one table per concrete class in the inheritance hierarchy as seen in the Database Diagram in **section 7.10.3**. This table contains columns from its corresponding concrete class and the fields from its ancestor (Amenity).

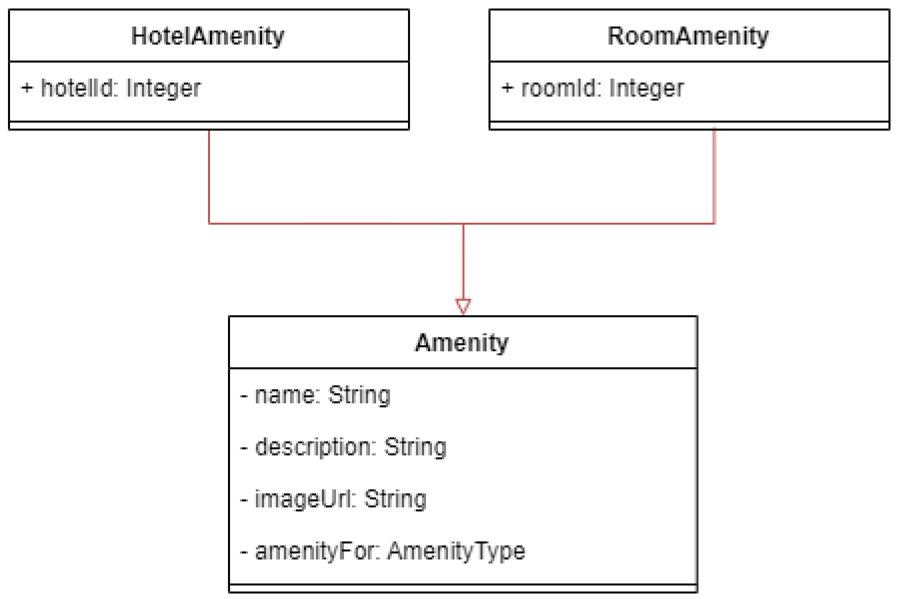
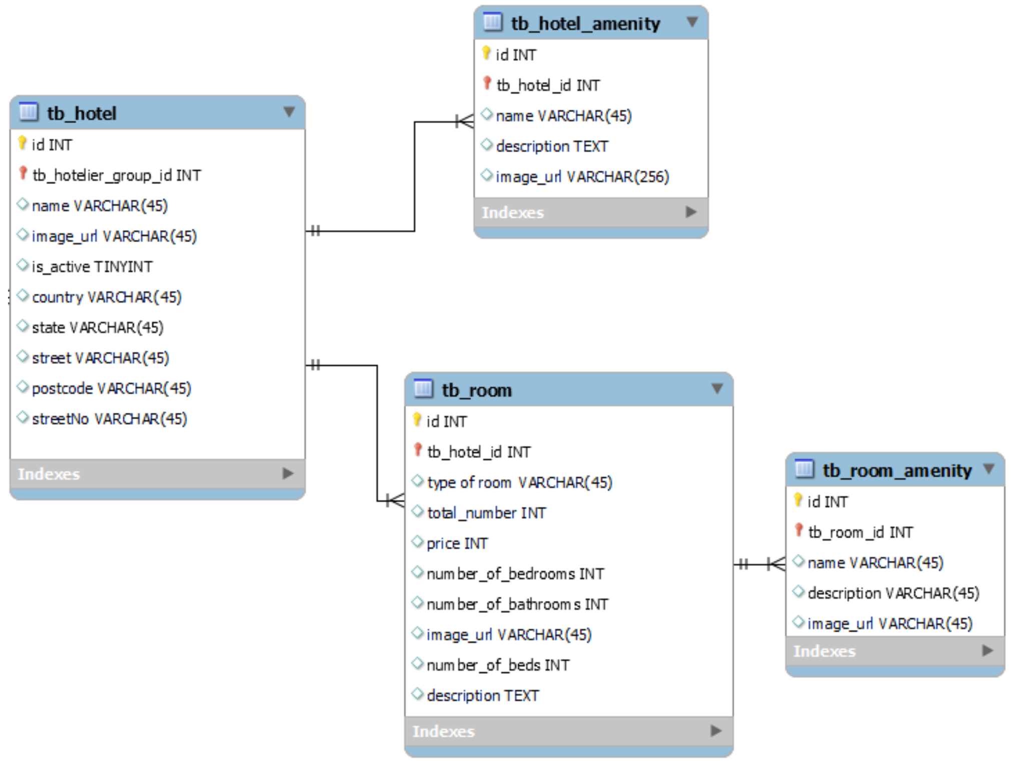
Additionally, this pattern helped us to reduce the persistence of irrelevant empty fields in the database in comparison with the single inheritance design pattern while keeping some of the single inheritance advantages such as there is no necessity to join to read the data from the HotelAmenity or RoomAmmenity. Furthermore, each table will be accessed only when each individual concrete class requires it.

7.10.2 Sequence Diagram

In **section 7.5.2** and **section 7.3.2** we can observe the communication between the different entities in when interacting with objects that fetched Ammeneties information from the database.

7.10.3 Entity Relationship Diagram

Note: To improve readability, Information of the Class and Database Diagram has been omitted

Class Diagram	Database Diagram
 HotelAmenity + hotelId: Integer RoomAmenity + roomId: Integer Amenity - name: String - description: String - imageUrl: String - amenityFor: AmenityType	 tb_hotel id INT tb_hotelier_group_id INT name VARCHAR(45) image_url VARCHAR(45) is_active TINYINT country VARCHAR(45) state VARCHAR(45) street VARCHAR(45) postcode VARCHAR(45) streetNo VARCHAR(45) Indexes tb_hotel_amenity id INT tb_hotel_id INT name VARCHAR(45) description TEXT image_url VARCHAR(256) Indexes tb_room id INT tb_hotel_id INT type of room VARCHAR(45) total_number INT price INT number_of_bedrooms INT number_of_bathrooms INT image_url VARCHAR(45) number_of_beds INT description TEXT Indexes tb_room_amenity id INT tb_room_id INT name VARCHAR(45) description VARCHAR(45) image_url VARCHAR(45) Indexes

7.11 Pattern 10: Authentication and Authorization

7.11.1 Description

We take JWT and web filters as security patterns, which are close to Intercepting validator patterns.

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed by using some cryptography algorithm. And according to our project structure, we developed and deployed the front-end and back-end separately, using JSON objects as communication data format, so JWT is appropriate for us to implement in the authentication and authorization pattern. Additionally, we are using the HMAC256 encryption algorithm to generate the token from the user's personal information and encrypt the password.

However, JWT is only a kind of standard and method for encapsulating authentication information, we have to implement a security framework that makes JWT work in an effective and appropriate way. Based on the controller pattern we choose and the number of Restful API we have implemented, it is better to use the intercepting validator. Because different roles will have different permissions for different applications, and we can parse the role from the token sent from the client, once we maintain a good permission list, it is effective to filter every request before it reaches the controller.

Therefore, we implement JWT and intercepting validator in four steps:

1. Every user needs to login with an existing account. Once login success, the user will get the token based on their personal information: id, email, and role. This token will become the authorization of the user.
2. Maintain a permission list for each role based on the requirements and use cases, and it could be stored in the database or in the settings file.
3. Next, we implement the interceptor which will be initialised with permission lists when startup the system. And then this interceptor will invoke JWT to verify every token when every request reaches the back-end
4. In the last step, the validator could use the permission list to filter with user information verified and parsed by JWT to achieve the goal of authentication.

   Thus, the client needs to login first to get a token based on user id, email and role, and then use it as authorization every time accessing the system; if authorising successfully, the information parsed by JWT will be used in the authentication.

7.11.2 Class Diagram

As shown in figure 7.15, the class diagram shows which classes are used for generating a token through JWT. And the loginHttpRequest will be the one in the whitelist. So it is unnecessary to put the interceptor class in the diagram.

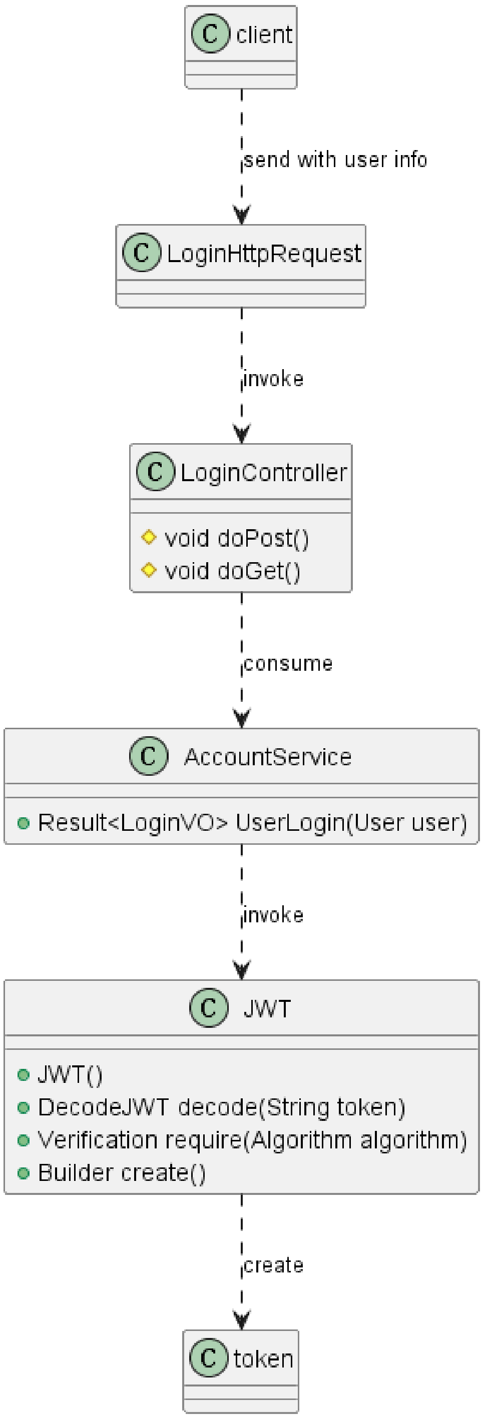


Figure 7.15: Authentication and Authorization - 1

As shown in **figure 7.16**, Classes are used for verifying a token through the pattern of the interceptor validator. The SecurityFilter class will be automatically triggered when every request reaches. And it will first verify the token, if the token is valid, it will then filter the permission of the user.

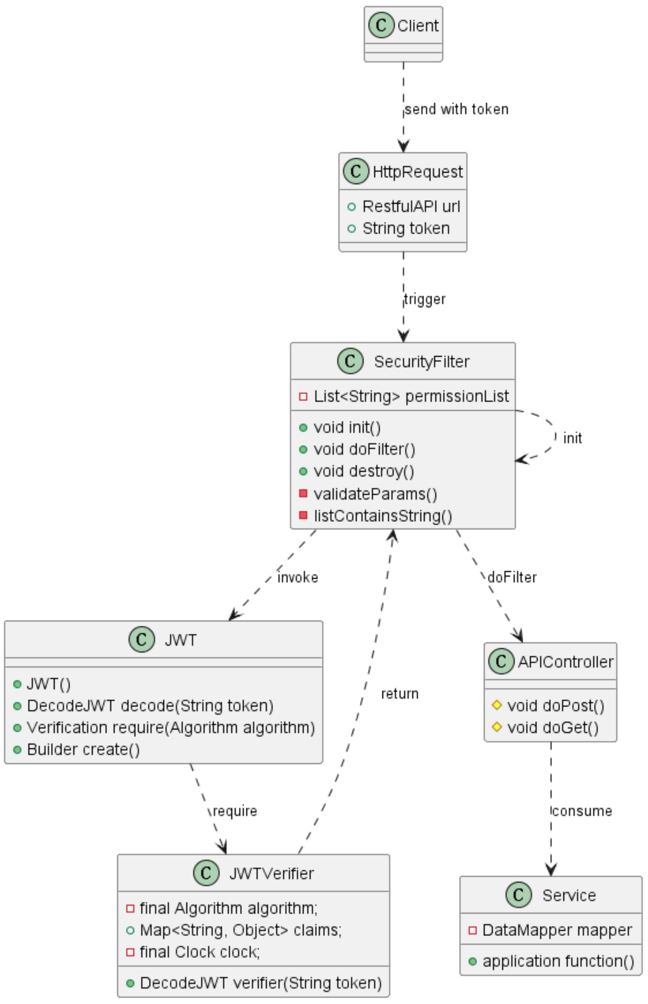


Figure 7.16: Authentication and Authorization - 2

7.11.3 Sequence Diagram

The first sequence diagram, **figure 7.17**, is to show the logic flow when users try to login to the system, and how users can get the token. The login check is to guarantee that the user has already signed up in the system and login with the right user. After login successfully, the user will get an authorised token.

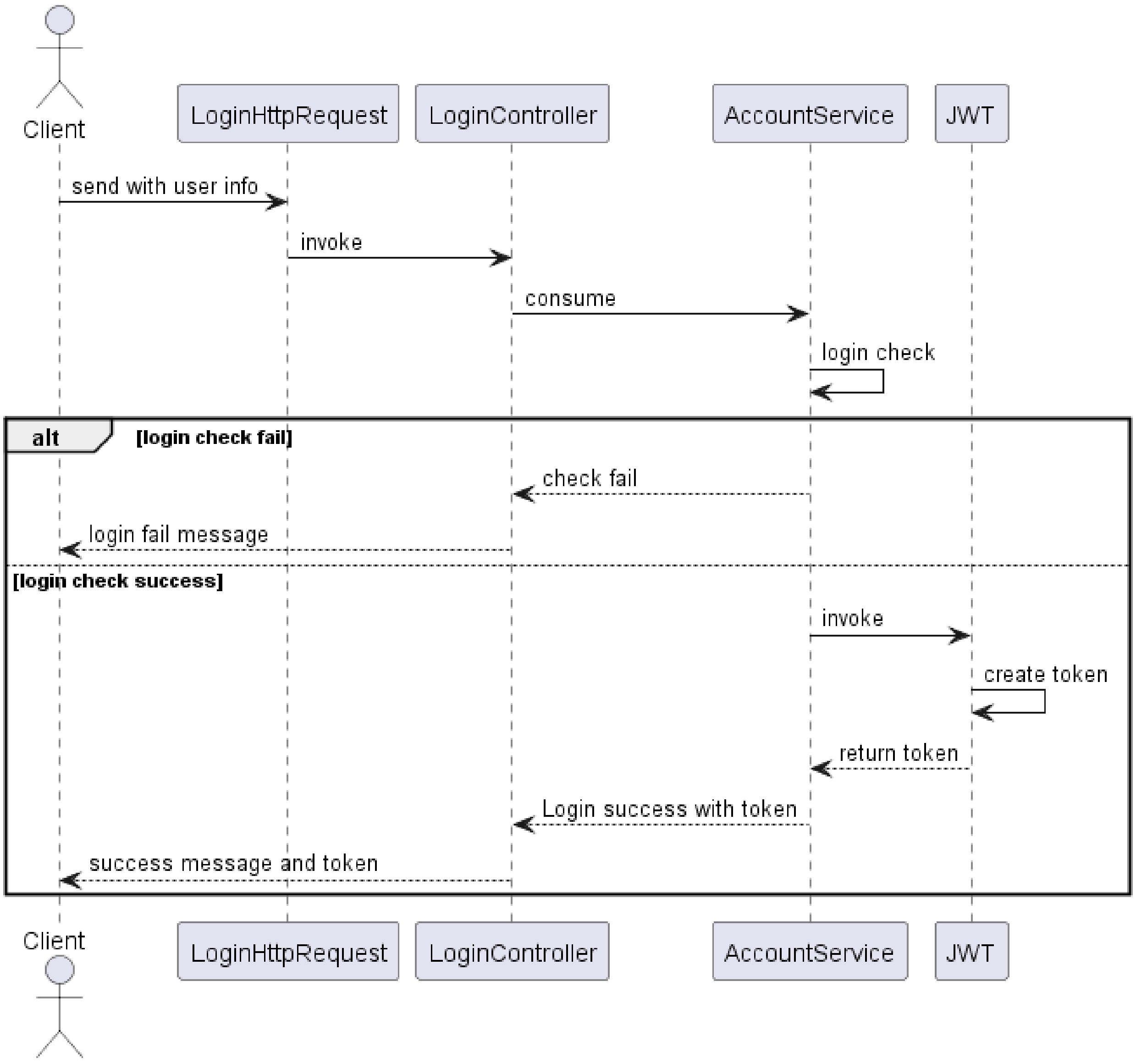


Figure 7.17: Authentication and Authorization Sequence diagram - 1

The second sequence diagram, **figure 7.18**, is to show the logic flow when users try to use the system, and how the interceptor works to verify the token and filter the user permission. It will first check if the token is valid, second, it will filter the permission of the user, and the request will reach controller

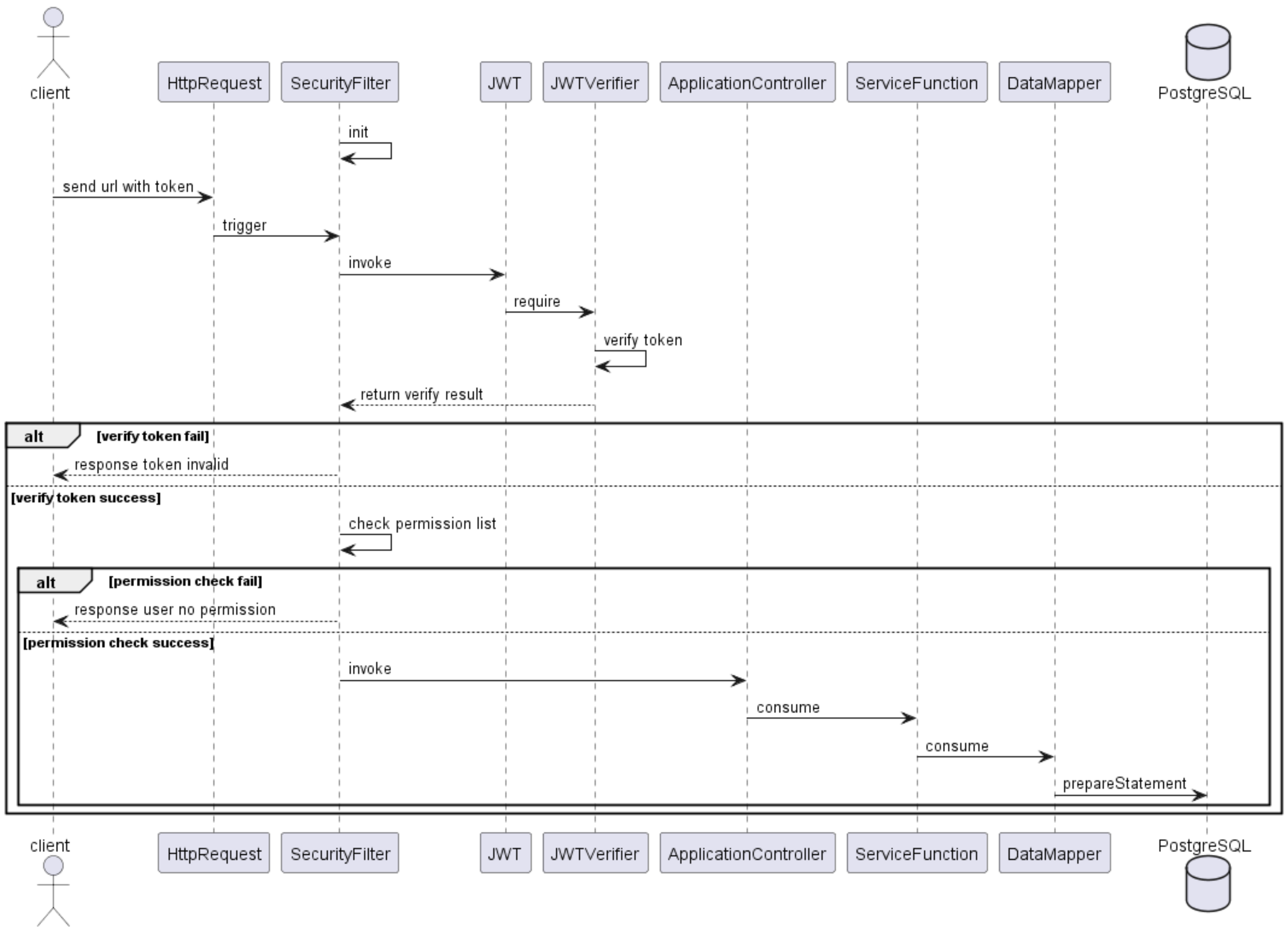


Figure 7.18: Authentication and Authorization Sequence diagram - 2

8. References

[1] Larman, C. (2005). *Applying UML Patterns: an introduction to object-oriented analysis and design and iterative development (3rd ed.).* Pearson Education, Inc.

[2] Cockburn, A. (2016). *Writing effective use cases.* Boston Addison-Wesley

[3] PlantUML. *Domain Model Tool Creator*. https://plantuml.com/commons

[4] Visual Paradigm Online.
https://online.visual-paradigm.com/diagrams/solutions/free-visual-paradigm-online/