Architecture Document

SWEN90007 Software Design and Architecture

Hotel Booking System
Team: Alphecca

In charge of:
Xiaotian Li 1141181 xiaotian4@student.unimelb.edu.au
Haimo Lu 1053789 haimol@student.unimelb.edu.au
Edison Yang 1048372 lishuny@student.unimelb.edu.au
Yifei Wang 1025048 yifewang2@student.unimelb.edu.au

Revision History

Date	Version	Description	Author
13/09/2022	02.00-D01	Initial draft	Xiaotian Li
13/09/2022	02.00-D02	added draft for pattern mvc, unit of work, identity map, lazy load, class diagram and descriptions	Haimo Lu
14/09/2022	02.00-D03	add more patterns, add diagrams	Xiaotian Li
14/09/2022	02.00-D04	add more patterns	Haimo Lu
17/09/2022	02.00-D05	add data mapper	Haimo Lu
18/09/2022	02.00-D06	Adding Domain Model diagram and its description	Edison Yang
19/09/2022	02.00-D07	fix pattern descriptions	Xiaotian Li
19/09/2022	02.00-D08	Finalizing Process View	Edison Yang
20/09/2022	02.00-D09	Adding use case diagram and the corresponding description.	Yifei Wang
20/09/2022	02.00-D10	Adding details.	Xiaotian Li
20/09/2022	02.00	Finalizing document formatting and now it's a final document	Edison Yang, Yifei Wang

Contents

1. Introduction

This document specifies the Alphecca hotel booking system's architecture, describing its main standards, module, components, *frameworks* and integrations.

1.1 Proposal

The purpose of this document is to give, in high level overview, a technical solution to be followed, emphasizing the components and *frameworks* that will be reused and researched, as well as the interfaces and integration of them.

1.2 Target Users

This document is aimed to be designed for the project team Alphecca, the teaching team of SWEN90007 and other end users who want to search uploaded hotels on a provided website. The document is written with a consolidated reference to the research and evolution of the system with the main focus on technical solutions to be followed.

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
Component	Reusable and independent software element with well defined public interface, which encapsulates numerous functionalities and which can be easily integrated with other components.
Module	Logical grouping of functionalities to facilitate the division and understanding of software.
Dao	Data Access Object. This object is the same as data mapper and it is in the Data source layer, which holds interfaces for database CRUD operations. It will be used by Blo objects.
Blo	Business Logic Object. This object is in the Service Layer, which holds all business codes for the system. It will be used by Controller objects, and may depend on different Blo objects and Dao objects.
Controller	MVC Controller Object. This object is in the Presentation Layer that takes responsibility to wrap/unwrap parameters, perform auth/security. It will use Blo objects to achieve business logic.
Entity	Entity Object is the Java object form of business domains. Also the entity object maps database table attributes from database.
Util	Objects that hold static methods for other components to use. Such as time/string operations, currency exchange calculation and so on.
IoC	Inversion of Control. We let the system create and maintain Java objects, rather than doing them by developers.
AOP	Aspect oriented programming. We extract reusable codes, and intercept it as an aspect to many workflows.
Alphecca Boot	This is the supporter developed by the team to speed up development for this project. Alphecca Boot implemented IoC, AOP, foundation

	MVC, embedded Tomcat Servlet container and SpringBoot-like annotations.
Bean	Java component / Objects. Usually has private fields, constructors.
Admin	Administer user of the system. It can see all users and all orders of the whole system
Client	Client user of the system. It will book a room from the hotel
Hotelier	The manager of the hotel, and provide rooms for clients to order
Transaction	The same as hotel bookings. One client can create one transaction, it includes target Hotel, target rooms and date info.

2. Architectural representation

The specification of the Alphecca hotel booking system follows the *framework* "4+1" [1], which defines a set of views, as shown in Figure 1. Each of these views approaches aspects of architectural relevance under different perspectives:

- The **logical view** shows the significant elements of the project for the adopted architecture and the relationship between them. Between the main elements are modules, components, packages and the application main classes;
- The **process view** shows the concurrency and synchronization aspects of the system, mapping the elements of the logical view to processes, *threads* and execution tasks;
- The **development view** focuses on aspects relating to the organization of the system's source code, architectural patterns used and orientations and the norms for the system's development;
- The **physical view** shows the hardware involved and the mapping of the software elements to the hardware elements in the system's environment.
- The **scenarios** show a subset of the architecturally significant use cases of the system.

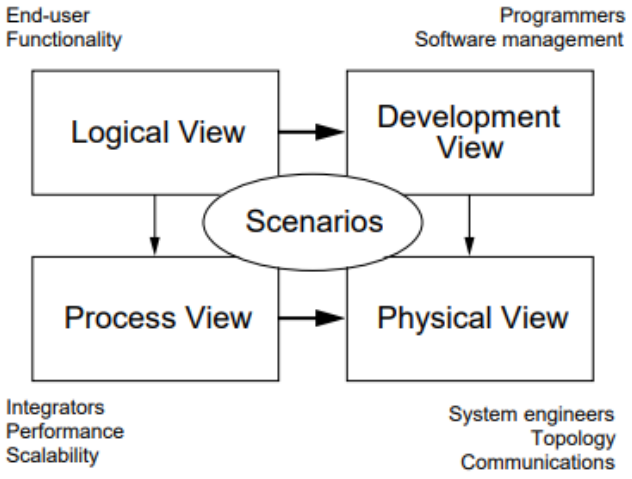

Figure 2 *Views of framework "4+1" (Kruchten, P.B. (1995))*

3. Architectural Objectives and Restrictions

3.1 Overview

The defined architecture's main objective is to make the system highly cohesive and low in coupling, so that makes it easy for developers to understand, maintain and extend the system.

For this system, we developed a supporter called Alphecca Boot, which is inspired by Spring Boot, and introduced several patterns such as IoC and AOP that were learned from Spring Boot. IoC is helpful to lower coupling by making an object free of creating objects, but only use them by Injection of Dependency. AOP can help developers to write less duplicated codes, such as authentication / authorization, Unit of Works, parameter validation and so on. Alphecca Boot also supports MVC annotations, which is useful for class scanning for the IoC container.

3.2 Objectives and Restrictions

3.2.1 Objectives

- We need to make the system highly cohesive and low in coupling. So that the system can be easily maintained and convenient for future extensions;
- We need to make the system fast in processing requests. Because the hotel booking system is a Business to Client product, high latency brings bad user experience;
- We need to make the system secure. Because booking a hotel will get lots of personal information from clients;
- We need to make the system fault tolerant. Internal errors won't crush and shutdown the whole server application;
- We need to make the system adapt to concurrency. Since there will be multiple users working on some shared data.
- We need to make the system easy to deploy. Since we separated the backend and the frontend, deploying 2 production servers should be easy.

3.2.2 Restrictions

- System complexity would increase if we want to implement a high cohesion low coupling system. This is because we need to introduce multiple design patterns and implement them with specific techniques. As a result the system complexity may increase and bring more cost for developers to learn and work.
- Making System fast will make it take more memory space, which is a common law of the software nature.
- Making the system secure will introduce extra modules into the system, which may bring technical risks for the development team.
- Some types of faults are inevitable, such as OS crashes, JVM crashes or power supply issues.
- Solving concurrency will result in decreasing performance. Synchronous locks are used on shared data.
- We deployed the application on the Heroku with the free plan, so the bandwidth has limitations.

3.3 Requirements of Architectural Relevance

This section lists the requirements that have impact on the system's architecture and the treatment given to each of them.

Requirement	Impact	Treatment
User authentication / authorization	Need fetch auth info from all requests, and perform identity check	A request filter will be implemented, and filter all incoming request
System need to be fast in performance	Some data can be cached, to reduce expensive database I/O	A cache can be implemented to cache result of method callings
System need to be secure	Need auth features, and incoming parameters should be validated	A JSR303 Validator should be implemented to validate incoming parameters. Passwords need to be encrypted before persistence
System need to be fault tolerant	Need to handle errors of the program and prevent shutting down	Exceptions need to be handled properly, and will be logged, and return exception messages for the front end.
System need to be working in concurrency	Need to prevent data loss, dirty read, phantom read and so on for the public resource.	• Using identity map and unit of work; • Synchronize lock should be introduced; • LockManager should be introduced for multiple requests;
System need to be easily deployed	Need to make the system not sensitive to the runtime environment. And need to be deployed on a cloud platform to be easily accessed	• Deploy on Heroku Platform; • Use Docker for deployment
System need to be easily maintained and extended	Need to reduce coupling, and reuse some utility modules to speed up development	• Implement patterns learned from the course; • Implement extra patterns such as IoC, AOP, and DI to speed up development; • Make use of 3-rd libraries.

4. Logical View

This section shows the system's organization from a functional point of view. The main elements, like modules and main components are specified. The interface between these elements is also specified.

4.1 Class Diagrams and Tables

4.1.1 Hotel Booking App

Figure 4.1 illustrates the UML class diagram of the hotel booking application system. For presentability and simplicity of the diagram, relationships with minor helper classes are

omitted. The light green represents Controller(Presentation Layer) package, the yellow represents package entity(domain), the light blue represents package Blo(Service Layer), the light pink represents package Dao(in charge of data). White represents all other helper packages like package constant or package param. For the high definition version, please see Github Repository.

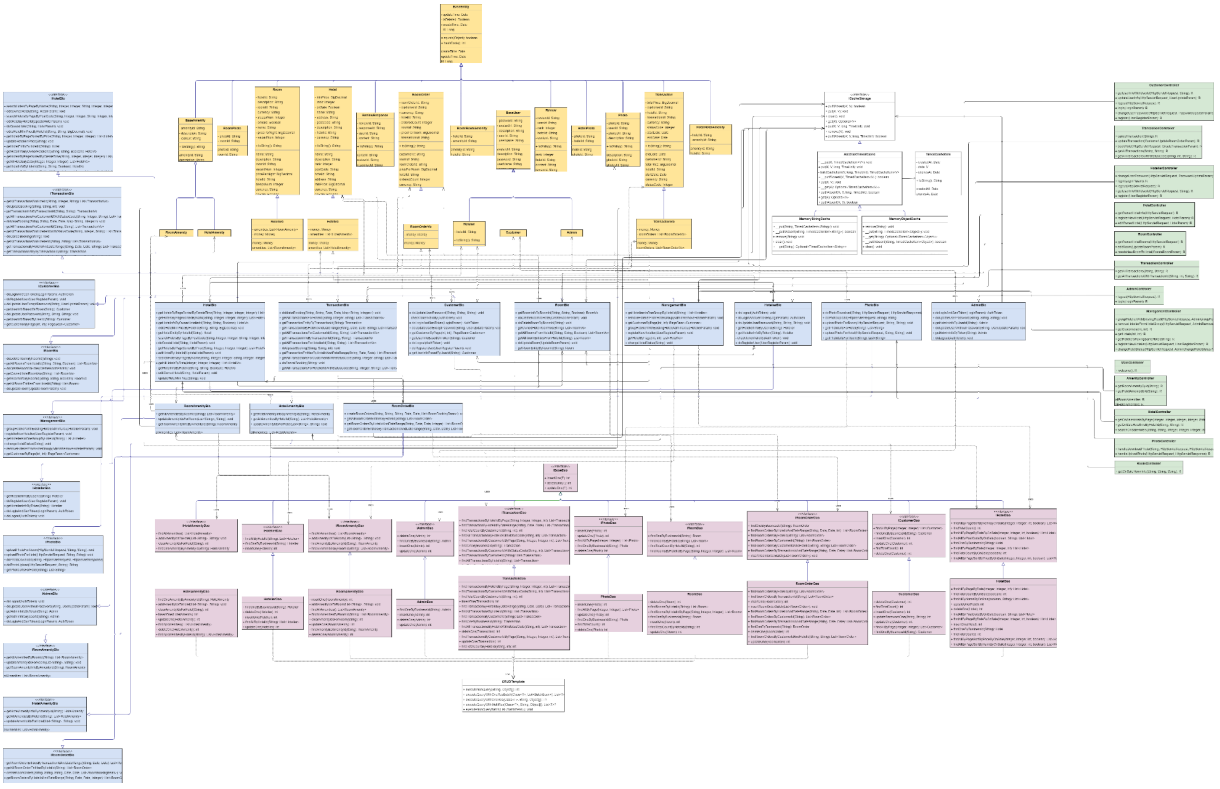

Figure 4.1.1.1 *UML Class Diagram*

(source: https://github.com/SWEN900072022/SWEN90007-2022-Alphecca/blob/main/docs/part2/images/UML_Class_Diagram.png)

According to the diagram, the system achieved the objective of high cohesion and low coupling:

1. Dependency Coupling occurs between Blos and Utils, Blo and Entities. This is because IoC and Dependency Injection make Blo and Controller objects free of object creation responsibility, only focusing on using them with Association relationships. Object creation is achieved by IoC container *BeanManager*, and all data bean instances are managed by the IoC container's BeanMap. The "use" dependency between Blo and Dao is necessary, because we use Lazy load for Dao: We call the *getLazyBeanByClass* method to lazy load a Dao instance inside methods of Blos, rather than instantiating Daos at fields when Blos are instantiated.
2. Each class has Single responsibility. As we can see, All controller classes take responsibility for handling incoming requests, but leave the business logic to Blos. As for Blos, All Blos focus on business logic for only one specific domain, and depend on the corresponding Dao. We can easily tell each class's responsibility, so that brings great advantages for developers in maintaining the project. Also we made use of GoF patterns such as Interface oriented programming, which is very useful for software engineering.

Table 4.1.1.2 describes the main classes of the hotel booking app:

Super Class / Interface	Class Name	Description
BaseUser	Admin	BaseUser is an abstract class of all roles in the HBS, users share fields among all roles in the system, while each subclass may have their own attributes or behaviors for future extension.
	Customer	
	Hotelier	
BaseAmenity	HotelAmenity	BaseAmenity is an abstract class of all amenities in the HBS, of which they share attributes among all amenities. Child classes may have their own attributes for future extension.
	RoomAmenity	
Room	RoomVo	RoomVo achieves embedded values when displaying price of the room in different currency
Hotel	HotelVo	HotelVo achieves embedded values when displaying minimum price of the hotel in different currency
RoomOrder	RoomOrderVo	RoomOrderVo achieves embedded values when displaying price of the room order in different currency
Review	/	domain for reviews
HotelPhoto	/	domain for hotel photo
Photo	/	domain for photo
Transaction	TransactionVo	Transaction holds data for a hotel booking, including room orders, date and money
IAdminBlo	AdminBlo	These classes are where business logics are implemented, the respective interfaces they inherit from are designed for extensibility of the system.
ICustomerBlo	CustomerBlo	
IHotelAmenityBlo	HotelAmenityBlo	
IHotelBlo	HotelBlo	
IHotelierBlo	HotelierBlo	
IPhotoBlo	PhotoBlo	
IRoomAmenityBlo	RoomAmenityBlo	
IRoomBlo	RoomBlo	
IRoomOrderBlo	RoomOrderBlo	
ITransactionBlo	TransactionBlo	
IAdminDao	AdminDao	These classes are where we construct the data source layer. The respective interfaces they inherited from are designed for extensibility of the system.
ICustomerDao	CustomerDao	
IHotelAmenityDao	HotelAmenityDao	
IHotelDao	HotelDao	
IHotelierDao	HotelierDao	

IPhotoDao	PhotoDao	
IRoomAmenityDao	RoomAmenityDao	
IRoomDao	RoomDao	
IRoomOrderDao	RoomOrderDao	
ITransactionDao	TransactionDao	
AbstractTimedCache	MemoryStringCache	MemoryStringCache stores String objects.
	MemoryObjectCache	MemoryObjectCache stores other generic objects

Table 4.1.1.2 *Classes table*

4.1.2 Alphecca Boot

Alphecca Boot is the framework created from draft by the development backend team. Since the module is at the framework layer of the system, it is not appropriate to discuss it in the end-user oriented logical view. Please refer to the Development view's Library and Frameworks to see more details.

4.2 Components

This section describes the main components of the Alphecca Hotel Booking system. Figure 4.4 shows the system component diagram.

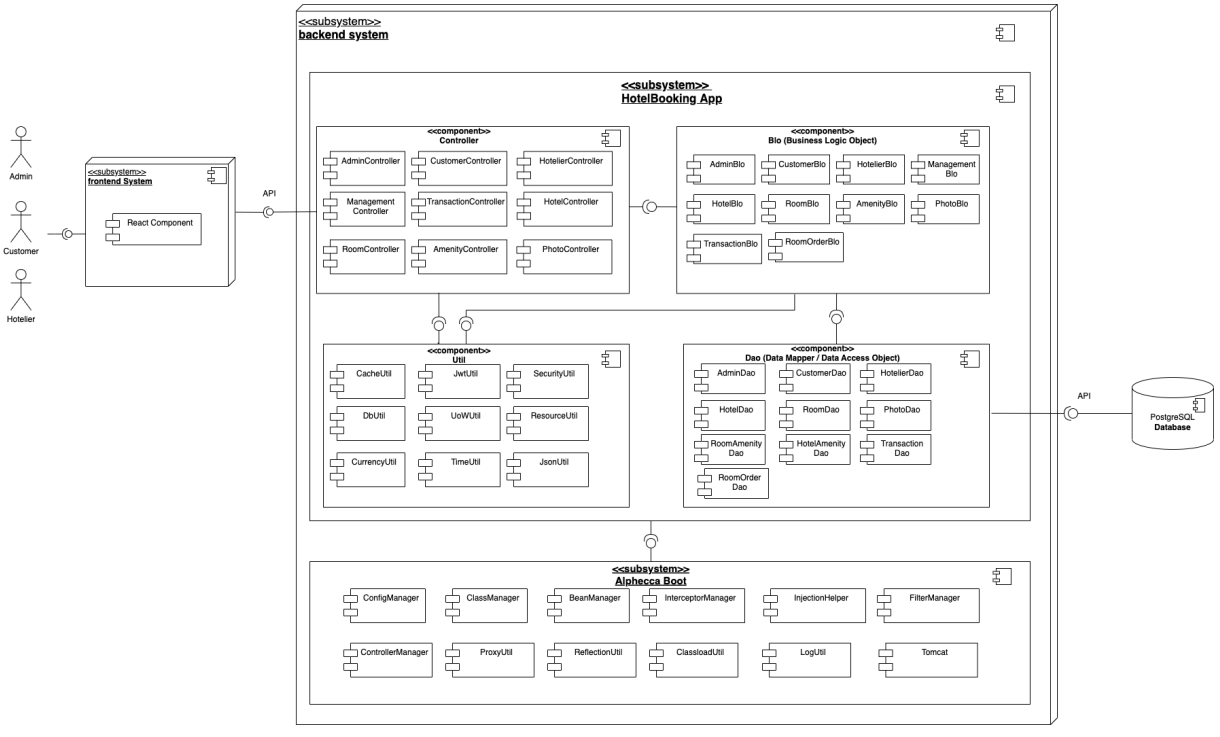

Figure 4.2 *System Component Diagram*

(source: https://github.com/SWEN900072022/SWEN90007-2022-Alphecca/blob/main/docs/part2/images/System_Component_Diagram.png)

In the following discussion, we will Introduce several important components of the system.

4.2.1 React Component

Responsibilities	Rendering UI elements, communicate with backend
Handled Requirements	All requirements that needs data I/O
Justification	Decision Making: 1. We divide the project into backend and frontend, and communication is based on RESTful API; 2. We need a framework to speed up frontend development; 3. Teammates familiar with React.js Framework.
Will it be reused?	Yes React components can be reused, because it can be called by other components in frontend subsystem, and can also adapt to different backend server if the backend follows the same API regulation
Source	react-18.2.0
Will it be reusable?	Yes This frontend react component can adapt different backend API implementations.

4.2.2 Controller Component

Responsibilities	Map the request URL and method; parse parameters; wrap response object; authentication; authorization
Handled Requirements	All requirements that needs data I/O
Justification	CIS standard. Controller is a compulsory component for MVC pattern.
Will it be reused?	No. Controller Components varies from different projects
Source	N/A
Will it be reusable?	It is technically reusable, but will not reuse it. Technically it is reusable. We can use the same controller with different Blo implementations, if the Blo method parameters are the same.

4.2.3 Blo Component

Responsibilities	Business logic support for the Controller layer.
Handled Requirements	Requirements that require Data persistence, calculation and sso on.
Justification	CIS standard. Blo Component is the Service Component, which is a compulsory component for MVC pattern.
Will it be reused?	No. Blo Service Components varies from different projects

Source	N/A
Will it be reusable?	No. Different Projects have different Business logic.

4.2.4 Dao Component

Responsibilities	Database CRUD operations
Handled Requirements	Requirements that require data persistence.
Justification	CIS standard. Dao Component is a key component for MVC pattern.
Will it be reused?	No. Dao Components vary from different projects, because different projects have different databases.
Source	N/A
Will it be reusable?	No. Different Projects have different databases

4.2.5 Util Component

Responsibilities	Support logic for Service layer
Handled Requirements	Requirements whose service layer uses reusable logics.
Justification	Decision Making: 1. Create Util to gather all reusable static logics; 2. Need some useful tools, such as Cache, Currency Exchanger, Date Calculator;
Will it be reused?	Yes. All Util Components can be reused for different projects. For example, TimedCache is a helpful thread-safe memory cache for a single machine.
Source	Latest
Will it be reusable?	Yes. Some utils are helpful, like DateUtil, ReflectionUtil, CacheUtil, and will be used in future projects.

4.2.6 Alphecca Boot Subsystem

Responsibilities	IoC Container, Servlet Container, Dependency Injector, Aop Proxy
Handled Requirements	All requirements that need the MVC web service.
Justification	Decision Making:

	3. Need a Supporter to speed up development; 4. Teammates know about Spring Boot; 5. Project can not use existing framework like Spring Boot; 6. Using Alphecca Boot can make projects with high cohesion and low coupling.
Will it be reused?	Yes. Alphecca Boot can be used as a 3-rd module in all kinds of Java MVC projects.
Source	Latest
Will it be reusable?	Yes. This Supporter is designed for all kinds of Java MVC projects, and totally coupling-free from the Hotel Booking App.

4.2.7 Postgres Database

Responsibilities	Data persistence
Handled Requirements	Login, register, booking, searching, viewing.
Justification	The only option: Based on project specification.
Will it be reused?	Yes. PostgreSQL Database is a 3-rd middleware for data persistence. And CRUDTemplate created by our team is suitable for all Java projects.
Source	Latest
Will it be reusable?	Yes. CRUDTemplate created by our team can be used in the future Java projects.

4.2.8 JWTUtil

Responsibilities	Generate or parse JWT token.
Handled Requirements	Authentication, Authorization.
Justification	Decision Making: 1. JWT RBAC is popular in industry; 2. JWT is more secure than Session; 3. JWT is more scalable, it can be used in multi-machine systems.
Will it be reused?	Yes. java-jwt is a third party Package, which is trusted by many users.
Source	java-jwt 3.19.2
Will it be reusable?	Yes. Will be used in the future Java projects

4.2.9 LogUtil

Responsibilities	Record runtime log into files and print to console
Handled Requirements	All
Justification	Decision Making: 1. We need a log system to track workflow; 2. System is deployed on heroku, can export log file
Will it be reused?	Yes. slf4j is a third party Package. It is maintained by trusted provider.
Source	slf4j-reload4j 1.7.36
Will it be reusable?	Yes. Will be used in the future Java projects

4.2.10 ReflectionUtil

Responsibilities	Perform Java reflection operations, like get class meta-data, get class loader, instantiate objects.
Handled Requirements	All
Justification	The only option. Since we use java.lang.reflect from JDK, there are no alternatives. Also, we need Reflection to fulfill IoC and AOP.
Will it be reused?	Yes. Reflection Util created by us is stable, and can be used in all Java projects
Source	JDK11
Will it be reusable?	Yes. Will be used in the future Java projects

4.2.11 ProxyUtil

Responsibilities	Dynamic proxy. Enhance methods, used for JSR303 validation, Unit of work.
Handled Requirements	Requirements needs JSONBody requests and database access
Justification	Decision Making: 1. For Class implemented from interfaces, we can use JDK proxy; 2. For Classes that do not have interfaces, we need to generate byte codes as super class of the original one, and enhance it with intercepted methods.
Will it be reused?	Yes.

	We use InvocationHandler from java.lang.reflect from JDK. And use Cglib for classes that do not have interfaces. ByteBuddy is still buggy.
Source	JDK11, cglib latest
Will it be reusable?	Yes. Will be used in the future Java projects

4.2.12 CacheUtil

Responsibilities	Holds thread safe K-V mapping
Handled Requirements	Identity Map; Token authentication with expiration time
Justification	Decision Making: 1. We need a thread safe k-v hashmap to implement Identity Map; 2. We need a map container that can configure expiration time.
Will it be reused?	Yes. We use CacheUtil both in Identity map and JWT.
Source	latest
Will it be reusable?	Yes. CacheUtil is a helpful thread safe HashMap with optional expiration time

4.2.13 Tomcat

Responsibilities	Holds servlets, handles web I/O
Handled Requirements	All
Justification	Decision making. Using embedded Tomcat is simple for development, we don't need to configure a lot, just instantiate a Tomcat object, and run it.
Will it be reused?	Yes. embedded Tomcat runs well.
Source	JDK11org.apache.tomcat.embed 8.5.76
Will it be reusable?	Yes. Will be used in the future Java projects

4.3 Packages

The Table below shows major packages for the hotel booking app:

Package	Class
app.model.entity	Admin
	Customer
	Hotelier
	HotelAmenity
	RoomAmenity
	RoomPhoto
	Room
	Hotel
	ReviewResponse
	RoomOrder
	RoomRoomAmenity
	Review
	HotelPhoto
	Photo
	Transaction
	HotelHotelAmenity
app.db.aop	UnitOfWorkInterceptor
app.model.bean	BatchBean
	PageBean
	UowBean
app.common.constant	DbConstant
app.db.helper	DbHelper
	UnitOfWorkHelper
app.db.resolver	BeanListResultSetResolver
	BeanResultSetResolver
	IResultSetResolver
app.db.util	CRUDTemplate
app.controller.admin	AdminController
	ManagementController
app.controller.customer	CustomerController
	TransactionController
app.controller.hotelier	HotelController
	HotelierController
	RoomController
	TransactionController
app.controller.shared	AmenityController
	HotelController
	MainController
	PhotoController
	RoomController
app.blo	AdminBlo
	CustomerBlo

	HotelAmenityBlo
	HotelBlo
	HotelierBlo
	PhotoBlo
	RoomAmenityBlo
	RoomBlo
	RoomOrderBlo
	TransactionBlo
app.dao	AdminDao
	CustomerDao
	HotelAmenityDao
	HotelDao
	HotelierDao
	PhotoDao
	RoomAmenityDao
	RoomDao
	RoomOrderDao
	TransactionDao

Table 4.2.3 *Package table*

For more project structure details, please refer to ***Source Code Directories Structure*** section.

5 Process View

The process view deals with the dynamic aspects of the system, explains the system processes and how they communicate, and focuses on the run time behavior of the system.The process view addresses concurrency, distribution, integrator, performance, and scalability. This document uses sequence diagrams to represent the process view as it shows the interaction logic between the objects in the system in a timely order that the interaction takes place.

5.1 Sequence Diagram

A sequence diagram shows process interactions arranged in time sequence in the field of software engineering. It depicts the processes involved and the sequence of messages exchanged between the processes needed to carry out the functionality. The sequence diagram includes the components as shown below:

a. Actor

Actor is an object that is located at the top of a sequence diagram and it represents an object interacting with the system.

b. Object

The object is the top of the sequence diagram shown as a rectangle. The sequence diagram is composed of a number of columns which is headed by an object.

c. Lifeline

Each object has a vertical dashed line drawn from the bottom of that object. When we move from the top of the diagram towards the bottom, we are moving in time as well. Something that is drawn below a particular event means that it occurs after it in the scenario.

d. Activation Bar

The activation bar is the box placed on the lifeline. It indicates that an object is active during an interaction between two objects. The duration of the objects staying active is represented by the length of the rectangle.

e. Message

An arrow from two objects specifies a message in a sequence diagram. A message can flow in any direction.

f. Combination Fragment

Combined fragments are logical groupings in sequence diagrams, represented by a rectangle that contains the conditional structures which affect the flow of message.

In our hotel booking system, there are three roles which are the Customer, Hotelier and the Admin. We have designed a sequence diagram for each of the roles as shown by Figure 5.1.1, Figure 5.1.2 and Figure 5.1.3 below.

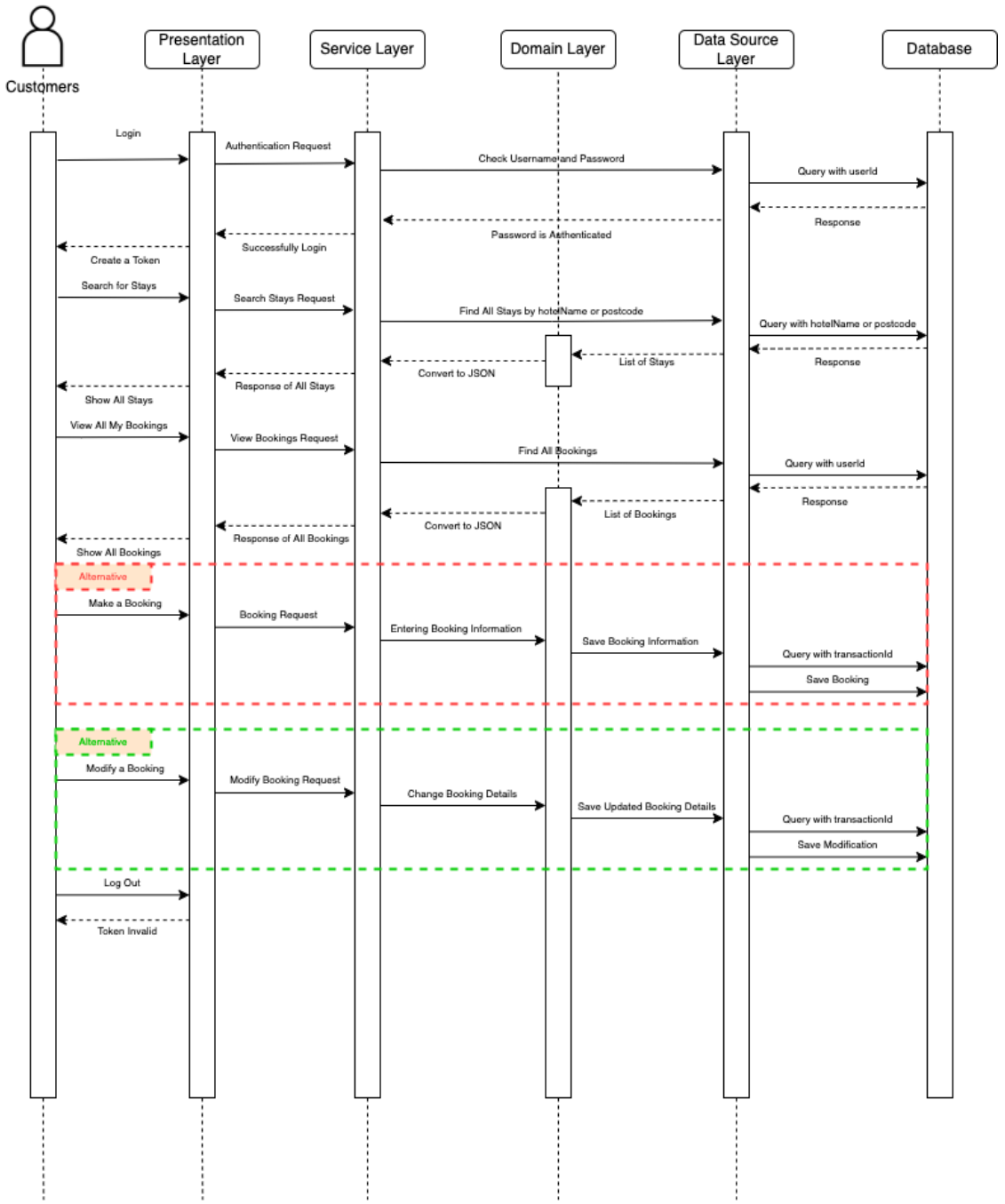

Figure 5.1.1 *Sequence Diagram for Customer*

(Source: https://github.com/SWEN900072022/SWEN90007-2022-Alphecca/blob/main/docs/part2/images/Sequence_Diagram_Customer.png)

A customer can log into the system. After logging in, the customer will be directed to the home page of the system then he/she can search for stays. In addition, the customer can view all the bookings he/she has. The customer is able to either make a booking or modify a booking. Finally, the customer can log out of the system.

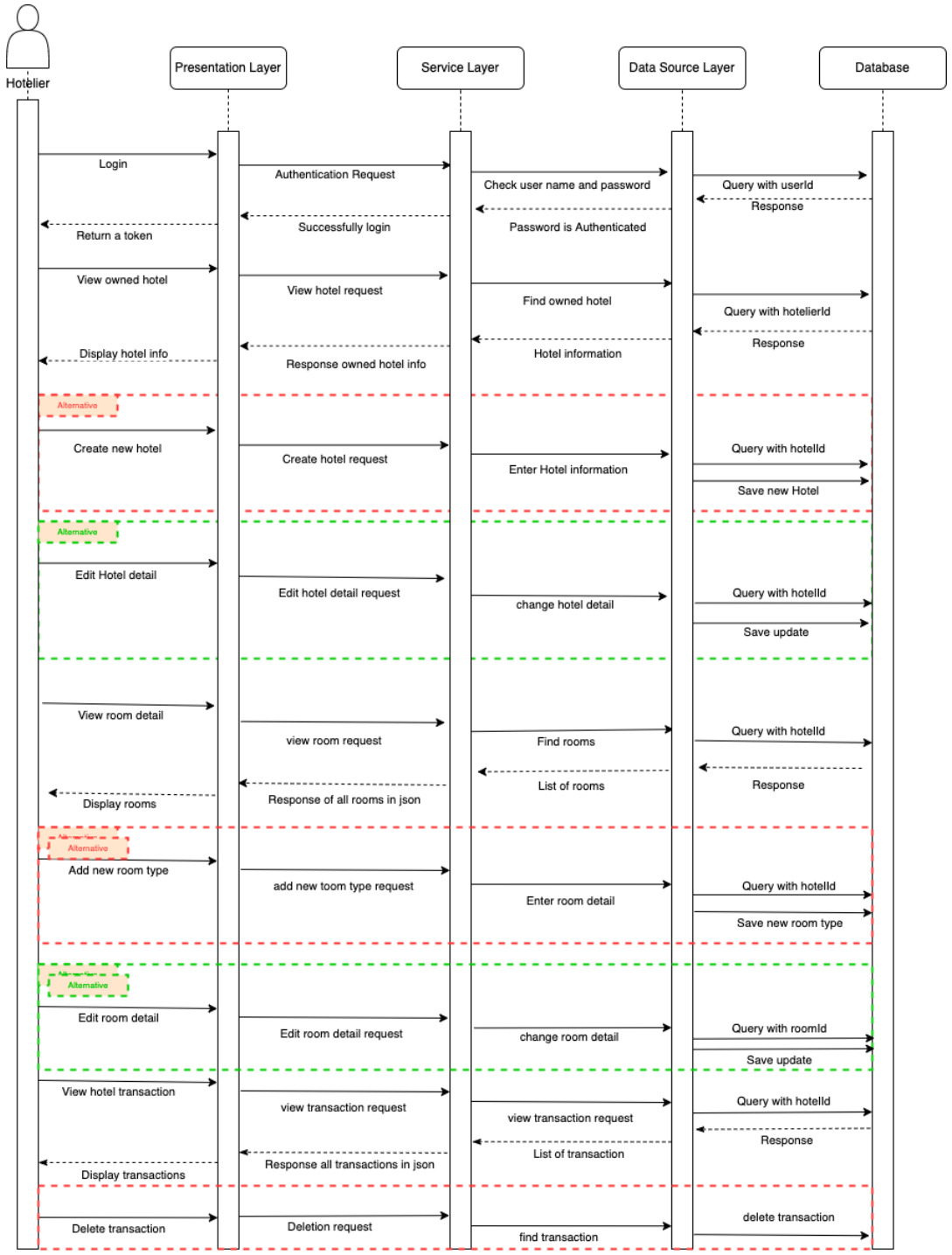

Figure 5.1.2 *Sequence Diagram for Hotelier*

(Source: https://github.com/SWEN900072022/SWEN90007-2022-Alphecca/blob/main/docs/part2/images/Sequence_Diagram_Hotelier.png)

A Hotelier firstly logs into the system as the role of Hotelier. The Hotelier is able to view his/her owned hotel. The Hotelier is able to either create a new hotel or edit the hotel details. The hotelier is able to view the room details in their hotel. The Hotelier can also either add a new room type of edit the existing room details. The Hotelier is able to view all of their hotel transactions. The Hotelier is able to delete a specific transaction.

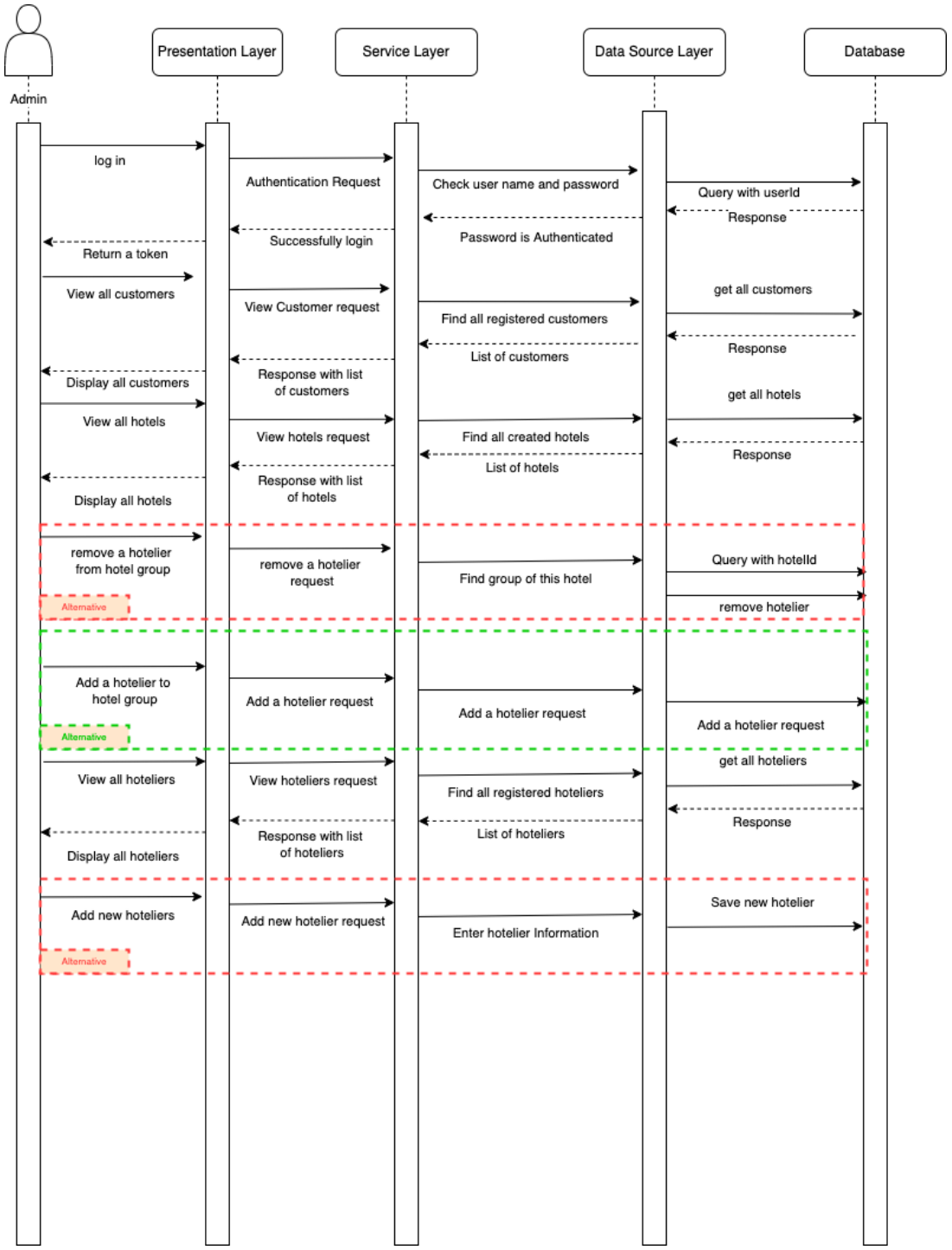

Figure 5.1.3 *Sequence Diagram of Admin*

(Source: https://github.com/SWEN900072022/SWEN90007-2022-Alphecca/blob/main/docs/part2/images/Sequence_Diagram_Admin.png)

The admin can login via an independent web page which has been split out from the main web page. The admin is able to view all the customers in the admin portal. The admin can also view all the hotels and the corresponding hoteliers in the portal. The Admin can either remove a hotelier from a hotel group or add a hotelier to a specific hotel group. The Admin can add new hoteliers.

6 Development View

This section provides orientations to the project and system implementation in accordance with the established architecture.

6.1 Architecture Patterns

An architectural pattern is a general, reusable solution to a commonly occurring problem in software architecture within a given context. In this section, we will be talking about domain model, data mapper, MVC, unit of work with dynamic proxy, identity map, lazy load, identity field, foreign key mapping, Association table mapping, embedded value, concrete table inheritance pattern as well as authentication and authorization in details.

6.1.1 Domain Model

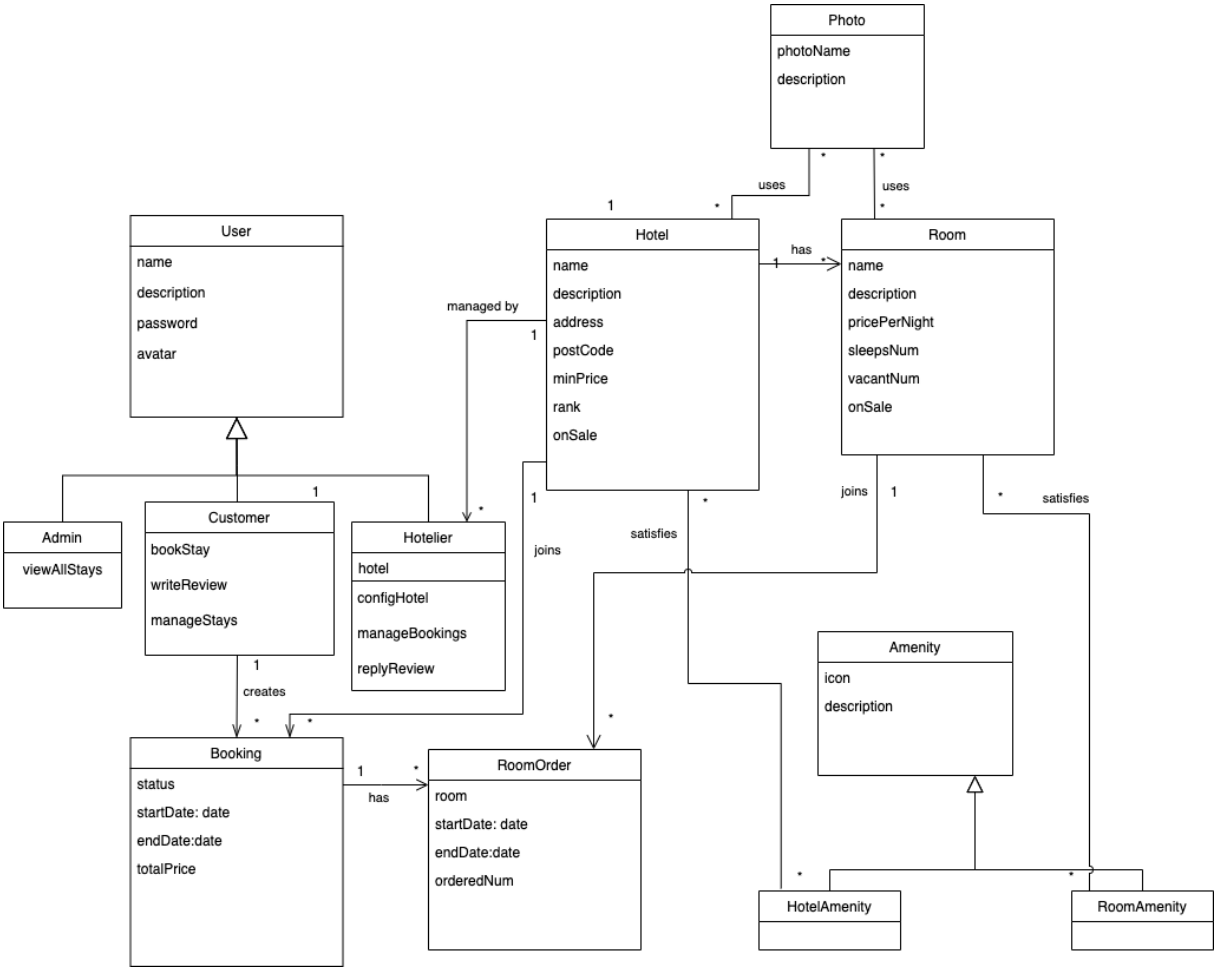

Figure 6.1.1 *Domain Model Diagram*

The domain model is a representation of meaningful real-world concepts pertinent to the domain that need to be modeled in software. The concepts include the data involved in the business and rules the business uses in relation to that data. A domain model leverages the natural language of the domain.

Using the Domain Model helps us to create a corresponding Java package called “domain”, and each object within the Domain Model is designed as Java class within that package. Table below shows the description of each object.

Object	Description
User	User is a general type of end-user in the system which is inherited into Admin, Customer and Hotelier.
Admin	Admin is one type of user in the system. Admin can view all the stays from the Customer.
Customer	Customer is one type of user in the system. Customers can create a Booking in the system, write reviews of a specific booking and manage their future bookings.
Hotelier	Hotelier is one type of user in the system. Hoteliers can create a hotel room by entering hotel details so that the customer can view it. They also have the authority to modify the booking from the customers, as well we manage the Hotel they have created in the system.
Booking	A Booking object contains the status of whether it’s a prior booking or a future booking. It also contains the check in date and check out date decided by the customer. The total price is included as well. The Booking object contains a RoomOrder object.
RoomOrder	It contains the room type of what the Customer has booked. It also has the check in date and check out date from the customer as well as the number of rooms that the Customer has booked.
Hotel	A Hotel is managed by the Hotelier. It contains its name, description, address, postcode, price, ranking and the information whether it’s on sale or not. A Hotel has a Room object.
Room	A room has its name, description, price per night, number of guests checked in, vacancy status and the information whether it’s on sale or not. It satisfies the RoomAmenity object.
Amenity	Facilities that a Hotel or a Room could have. It contains a corresponding icon and its description.
HotelAmenity	Inherited from the Amenity. It’s a facility that a Hotel can have.
RoomAmenity	Inherited from the Amenity. It’s a facility that a Room can have.
Photo	Photo object of Hotel and Room. It contains its name and the corresponding description.

6.1.2 Data Mapper

We use the data mapper pattern for the data source layer, this means we keep in-memory objects based on the database. The benefits of this pattern is that the database can be isolated with in-memory objects, and the in-memory objects will not need knowledge of the database schema.

We have a package called “dao” (data access object), where we implemented all the data mappers. All data mappers implement *IBaseDao* interface to make insertion, updating and deletion convenient for Unit of work. Concrete data mappers are shown below:

Package	Implementation	Functions
app.dao	AdminDao	findOneByBusinessId insertOne updateOne deleteOne
	CustomerDao	findTotalCount findAllByPage findOneByBusinessId insertOne updateOne updatePasswordOne deleteOne
	HotelAmenityDao	insertOne updateOne deleteOne findAllAmenities findOneAmenityByAmenityId findAllAmenitiesByHotelId addAmenityIdsToHotel clearAmenityIdsForHotel
	HotelDao	insertOne updateOne deleteOne findTotalCount findTotalCountByOnSale findAllByPageByDate findAllByPageByDateByOnSale findAllByPageSortByPriceByOnSale findAllByPageSortByRankByOnSale findAllByPostCodeByOnSale findAllByNameByOnSale findOneByBusinessId

	HotelierDao	insertOne updateOne deleteOne findTotalCount findAllByPage findOneByBusinessId findAllByHotelId
	PhotoDao	insertOne updateOne deleteOne findTotalCount findAllByPage findOneByBusinessId
	RoomAmenityDao	insertOne updateOne deleteOne findAllAmenities findAmenityBeAmenityId findAllAmenitiesByRoomId addAmenityIdsToRoom clearAmenityIdsForRoom
	RoomDao	insertOne updateOne deleteOne findOneByBusinessId findTotalCountByHotelId findRoomsByHotelIdByPage findRoomsByHotelId
	RoomOrderDao	insertRoomOrderBatch insertOne updateOne deleteOne findOneByBusinessId findRoomOrdersByTransactionId findRoomOrdersByCustomerId findRoomOrdersByHotelId findRoomOrdersByCustomerIdAndHotelId findRoomOrdersByTransactionIdAndDateRange findRoomOrdersByHotelIdAndDateRange

	TransactionDao	insertOne updateOne deleteOne findOneByBusinessId findTotalCountByHotelId findTransactionsByHotelIdByPage findAllTransactionsByHotelId findAllTransactionsByHotelIdWithStatusCode findTotalCountByCustomerId findTransactionsByCustomerIdByPage findTransactionsByCustomerId findTransactionsByCustomerIdWithStatusCode findTransactionByHotelIdByDateRange

6.1.3 MVC

The mvc describes a pattern in abstract that programs are arranged in hierarchies based on functions, where the "main " program invokes several components, which may in turn invoke still further components.

This components can be divided in three parts, as follows:

- **Model**: contains the core functionality and data
- **View**: displays the information to users
- **Controller**: handles inputs from users

In our project, The first components are implemented as multiple java packages that contain business logics and database data manipulation.

Package	Class
blo (service layer)	AdminBlo
	CustomerBlo
	HotelAmenityBlo
	HotelBlo
	HotelierBlo
	PhotoBlo
	RoomAmenityBlo
	RoomBlo
	RoomOrderBlo
	TransactionBlo

Package	Class
dao (data source)	AdminDao
	CustomerDao
	HotelAmenityDao
	HotelDao
	HotelierDao
	PhotoDao
	RoomAmenityDao

	RoomDao
	RoomOrderDao
	TransactionDao

The view components correspond to the frontend React components that render the html pages.

The controller component

Parent Package	Child Package	Class
Controller (Presentation Layer)	Admin	AdminController
		ManagementController
	Customer	CustomerController
		TransactionController
	Hotelier	HotelController
		HotelierController
		RoomController
		TransactionController
	Shared	AmenityController
		HotelController
		MainController
		PhotoController
		RoomController

6.1.4 Unit Of Work with Dynamic Proxy

The unit of work pattern describes a way to keep track of which domain objects have changed (or new objects created), so that only those objects that have changed need to be updated in the database. It maintains objects that are affected by a business transaction as a list, and is in charge of writing out of changes.

The pattern brings enhanced code maintainability, and less duplicated codes.

The unit of work pattern keep tracks of four list of objects:
new : a list of new objects that have been created and must be inserted into the database.
dirty : a list of existing objects whose attributes have changed values since they were read from the database.

clean : a list of existing objects that have not been changed since they were last read from the database.
deleted : a list of existing objects that need to be removed from the database.
Each object that is created from a query **must be in exactly one of these lists**. In practice, the clean list need not be created, as the set of clean objects is just the set of all objects not in the new, dirty, or deleted lists.

The table below describes the usage of the "Unit Of Work" pattern in our project. UnitOfWork is achieved by UnitOfWorkHelper class in package "db/helper", we use it by importing the object into respective classes.

Object	Operation	Design Rationale
CustomerBlo	Insert, update	Add new customer to the "new" list Add modified customer to the "dirty" list
HotelBlo	update	Add changed hotel to the "dirty" list
HotelierBlo	Insert, update	Add new hotelier to the "new" list Add modified hotelier to the "dirty" list
ManagementBlo	Insert, update	Add new hotelier to the "new" list Add modified hotelier to the "dirty" list

We implemented the pattern using **dynamic proxy** with Cglib Library to enhance the relevant Controller classes with units of work. The logical relationship as shown in the figure below:

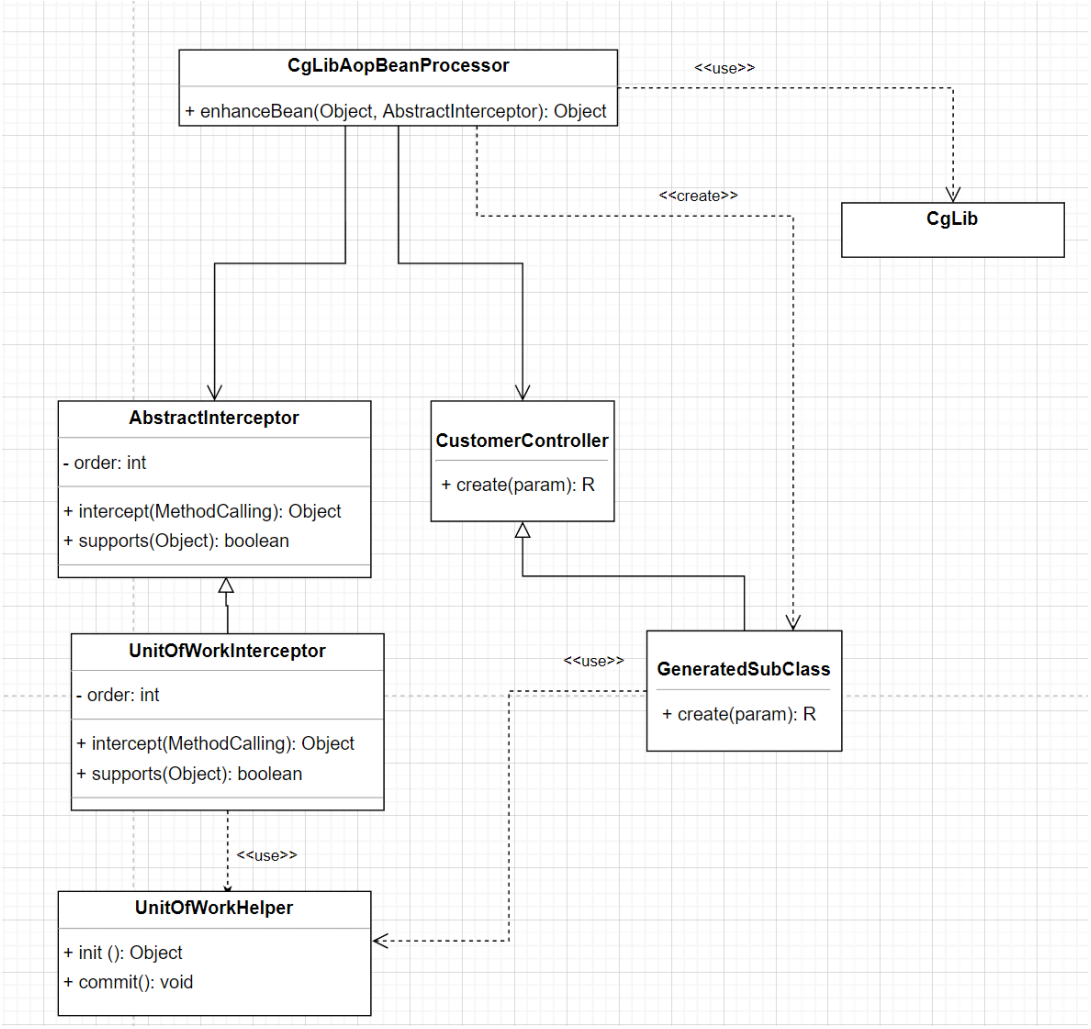

Figure 6.1.4 . *Unit of work with dynamic proxy class diagram*

(source: https://github.com/SWEN900072022/SWEN90007-2022-Alphecca/blob/main/docs/part2/images/Unit_of_Work.png)

Firstly, we defined *UnitOfWorkInterceptor*, which extends from *AbstractInterceptor,* where we wrap the target controller object with *UnitOfWorkHelper*'s init and commit method.

Then, during the class scan stage, the Alphecca Interceptor manager will scan all classes that extend *AbstractInterceptor,* spot *UnitOfWorkInterceptor*, then Alphecca Interceptor will put it into the interceptor list and bean map.

During the bean instantiate stage, interceptor list and bean map will be traversed, Since the *CustomerController* supports the *UnitOfWorkInterceptor*, *CgLibAopBeanProcessor* will be created. *CgLibAopBeanProcessor* will use the Cglib library to generate one subclass of the *CustomerController* with enhanced methods by *UnitOfWorkHelper.*

Then, Controller Manager will map the Generated subclass from *CustomerController* to the servlet in the run-time. As a result, all methods of this Controller will be wrapped with Unit of work logic.

Sequence diagrams will be shown in the ***Libraries and Frameworks*** section later.

6.1.5 Identity Map

The identity map pattern is an elegant way of maintaining data integrity. This pattern makes sure that the same data is only loaded into one object only. This pattern prevents cases when the same data is loaded into different objects, and if those objects are then changed separately, this will create clashes of different values, so we will not be able to determine the correct values to update our database.

The pattern simply creates mapping from unique business IDs derived from the database, with the object instances associated with that ID. So when an object is needed, the system first checks with the identity map to see if it already contains it, if it has, the identity map will simply return that object, without looking through the database. If the object is not in the map, a database will be query for the object, and it will be added to the identity map.

We designed a thread-safe key-value memory storage with optional expiration time, and made it a reusable module for 2 features of the project: JWT authentication's expiration logic, and Identity Map's identity-object mapping. As is shown in the figure below.

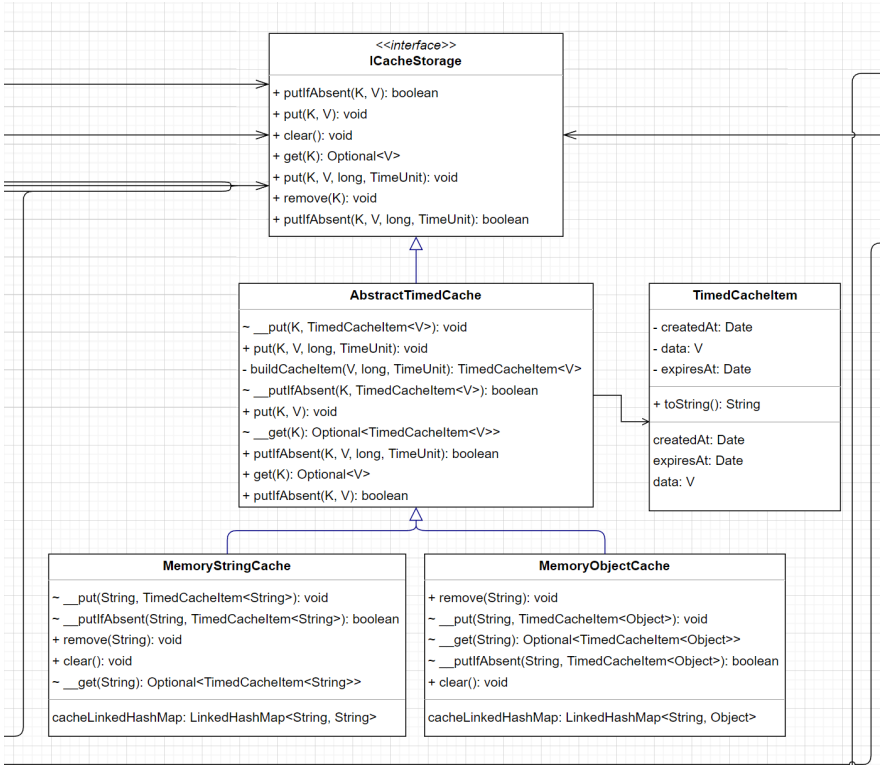

Figure 6.1.5.1 *"Cache" module class diagram*

The reusable module's main data structure is a concurrent hash map. We can use it to implement Identity map by mapping a domain object with its identity id, so that to make sure multiple database queries from one id only produce one singleton domain instance. The package has an interface called "ICacheStorage", which is implemented by an abstract class called "AbstractTimedCache". The abstract class is further extended by two concrete classes called "MemoryObjectCache" and "MemoryStringCache", with the first stored objects and

the second stored string. We design the map so that an object can be configured to expire after a designated amount of time by default if we do not manually remove it upon the business transaction. So that the "Cache" with expiration time can also be reused for token authentication.

The identity map should have one map per database table**.** The map uses a predefined prefix to identify the table, and the unique primary ID in the database to identify the entry of the table. The predefined strings are stored in CacheConstant Interface.

Here's an example of the key format:

Table Name	Key-prefix	key
hotel_amenity	e-hotel-am-	e-hotel-am-**ID**

As a result, we logically make a singleton map object into multiple Identity Maps, and each of the Identity Map only represents one Table.

Note: This "Cache" module used for identity maps is **NOT** aiming to reduce database I/O. This "Cache" utility maps identity id with the one object from the database, so that to prevent multiple instances for one identity. We call this utility "Cache" because this module is reusable, and used in token authentication logic.

Data Consistency consideration: According to the course material, the identity map should be attached to request context. However, the reusable "Cache" utility crafted by us is not thread-specific in this project, because we reused this module for auth token storage and made it singleton for the server process. As a result, concurrency issues should be resolved in this case.

We tried to implement an "Identity Map" for shared threads. For different modules in one request thread, they share the object from the identity map to merge changes, there is no difference with the thread-specific map. For different requests, we need to make changes visible and prevent outdated data, after updating commits in a request context, the value in the shared identity map is evicted, so that the other request can get updated data, and can merge changes.

Moreover, for scalability objectives, eventual consistency for distributed backend servers is also taken into consideration for the thread-shared scheme. For example, if the backend system extends to multiple processes/servers, they all have their own singleton map objects on their application runtime process, and can not know whether the database is changed by the other backend node. As a result, we need expiration time to make sure that different identity maps from different backend nodes don't have out-dated data. Also, the expiration time for data is randomized to prevent an avalanche of outdated data, expiration time is shown below:

Type	Max expiration time
Data requires high consistency (Search, Orders, admin)	10 seconds max
Data requires consistency (Index hotels)	30 seconds max
Normal Data (User Info)	5 minutes max

Static Data (Photo Meta, Amenities)	60 minutes max

All expiration time can be modified in a configuration constant file so that it is convenient to modify according to the requirements in the future. To prevent null pointer issues, we use lazy clean for the map.

One important note is that if one domain object is modified (update, deletion), the corresponding Identity Map record should be cleaned after commits. We can do this by calling the “cache.remove” method, and also, Unit of Work’s “Commit” method has already implemented the clean logic. We use lazy deletion, so it won’t cause null pointer exceptions.

Also, it is important to update the database first, and then clean the Identity Map, to prevent dirty data from outdated databases. However, there are still data consistency issues, such as multi-node database synchronization, network latency. It is suggested to double clear the Identity Map cache later in some random time period to make sure data consistency.

Discussion: It is **NOT** convenient to make the identity map be shared with different request contexts, because we need to handle retrieval data consistency issues, isolation issues and visibility issues within requests in a distributed system. It is the best practice that the identity map should be thread-specific to prevent problems, as is taught in the course.

A possible simple improvement of our system is that we can attach thread ID to the key. Since the map object is thread-safe, we can isolate the map to identity maps for different request threads.

Example of use: The code snippet below shows a common query example use with the Identity Map:

```
@Override
public Customer getUserInfoBasedByUserId(String userId) {
    Optional<Object> cacheItem = cache.get(CacheConstant.ENTITY_USER_KEY_PREFIX + userId);
    Customer customer = null;
    if (cacheItem.isEmpty()) {
        synchronized (this) {
            cacheItem = cache.get(CacheConstant.ENTITY_USER_KEY_PREFIX + userId);
            // if the result is still empty
            if (cacheItem.isEmpty()) {
                ICustomerDao customerDao = BeanManager.getLazyBeanByClass(CustomerDao.class);
                customer = customerDao.findOneByBusinessId(userId);
                // if no such customer
                if (customer == null) {
                    throw new RequestException (
                            StatusCodeEnume.USER_NOT_EXIST_EXCEPTION.getMessage(),
                            StatusCodeEnume.USER_NOT_EXIST_EXCEPTION.getCode());
                }

                // use randomed expiration time to prevent Cache Avalanche
                cache.put(
                        key: CacheConstant.ENTITY_USER_KEY_PREFIX + userId,
                        customer,
                        RandomUtil.randomLong(CacheConstant.CACHE_NORMAL_EXPIRATION_PERIOD_MAX),
                        TimeUnit.MILLISECONDS
                );
            } else {
                // data is put by other thread, just get from cache.
                customer = (Customer) cacheItem.get();
            }
        }

    } else {
        customer = (Customer) cacheItem.get();
    }
    return customer;
}
```

Figure 6.1.5.2 *example for identity map*

According to the example code snippet, firstly, the system will check the existence of the domain object with an Identity from the cache map. If there is nothing in the Identity Map, perform a database query, and then put the result into the Map. If the object exists, get from the map directly.

We also introduced synchronized lock here to make sure only one thread can access the database if the domain object is null, and then put the result to map. Otherwise, multiple threads will perform duplicated database queries, it is unacceptable because it breaks the uniqueness of the object from the identity map.

6.1.6 Lazy load

The lazy load pattern requires us to delay loading of an object until the point we need to use it. Lazy load pattern loads some dummy objects that contain no data, and only loads the corresponding data when we try to access the data. The lazy load pattern improves the performance of the system as some objects are postponed to load.

We implemented the lazy load for all *@Lazy* annotated component classes in the BeanMap, and mainly used in Dao objects.

At the beginning, all component classes are scanned by ClassManager and map in the BeanManager when the system starts. If some classes are annotated as *@Lazy,* they will not be loaded and assigned as null in the BeanMap. If method *getLazyBeanByClass* from BeanManager is called for the first time, it will perform a singleton lazy load to instantiate the lazy object, and the method returns previous-initialized for all future calls. We mainly used this technique for data access objects.

Example of use: As is shown in the figure below:

```
/**
 * called by outside to get lazy beans
 * @param clazz target class
 * @return lazy bean instance
 */
/unchecked/
public static <T> T getLazyBeanByClass(Class<T> clazz) {
    Object instance = getBeanFromBeanMapByClass(clazz);

    // check existence first without lock to guarantee performance
    if (instance == null) {
        synchronized (BeanManager.class) {
            instance = getBeanFromBeanMapByClass(clazz);
            if (instance == null) {
                instance = ReflectionUtil.genNewInstanceByClass(clazz);
                BEAN_MAP.put(clazz, instance);
                LOGGER.info("Lazy loaded: [{}]", instance.getClass().getName());
            }
        }
    }

    return (T) instance;
}
```

Figure 6.1.6 *Core implementation of lazy load*

As is shown in the figure, when the method is called, firstly it will try to get an instance from the bean map, if the instance is not instantiated, *ReflectionUtil* will instantiate it and put it into the bean map. The reason why we detect "instance == null" twice is to guarantee performance. First detection will not lock the method and return result immediately, the second one is locked, to prevent multiple threads rushing in and generating multiple instances.

This *getLazyBeanByClass* method can be reused in any other component class in the system to achieve lazy loading. Not only Dao objects, but all components classes with *@Lazy* annotation.

Discussion: We implemented the Lazy Load pattern for component objects, but we didn't use the lazy load pattern for domain object fields in the system, because it may attract multiple expensive database I/Os for one domain object in some cases, so called *Ripple Loading*. Since we don't have large data records with many columns from the database, eager loading the whole object from the database is easy to unify data access layer logic, and then implementing a cache technique can reduce the database I/O as an alternative. We plan to try the cache later, and have a performance discussion in the future.

6.1.7 Identity field

For the key of tables, we decided to use **Meaningful keys** for business ID (user_id for user table, hotel_id for hotel table), and use **meaningless keys** for Primary Keys as the "id" field in the table. All keys are **database-unique keys.**

The reason we have meaningful business keys is that these keys can be read by developers and database administrators. Sometimes, business keys can be displayed for the end-user, such as the unique user login ID.

However, human-readable keys tend to be Char type, and have redundant information, it is not suitable to make them Primary Keys for the DBMS. Also, char type primary keys with long size can not be fast indexed. So we also need meaningless numeric keys generated by the system to serve as Primary keys, which can be sorted, and speed up table indexing.

We have to use database-unique keys, this is because we are using Identity Map pattern. Each domain object must have a unique identity field.

For auto generated meaningless primary keys, we used the Snow-Flake algorithm for generating the unique ID. The Snow-Flake algorithm can generate monotone increasing IDs, and these IDs can be used in multi-server systems, which fulfill the scalability objective.

To simplify the project, we make the business ID the same as the primary key ID in most tables. It is suggested that developers should design some regulation for meaningful keys for these business IDs.

6.1.8 Foreign key mapping:

A foreign key is a column that link with another table in the database, when some objects in our domain requires references to another object in the domain, using foreign key mapping could help us preserve such relationships in the database. In practice, this is achieved by having a column in the table

corresponding to an object, that is a foreign key field referring to another object where the foreign key mapping is required.

Here are some examples of one-to-many relationship foreign key mapping in our project:

One side	Many Side	Database Scheme
Hotel	Hotelier	The “hotelier” table has a foreign key “hotel_id”, reference the primary key in table “hotel”, that shows which hotel a hotelier has.
Hotel	Room	The “room” table has a foreign key “hotel_id”, reference the primary key in table “hotel”, that shows which hotel a room belongs to.
Hotel	Transaction	The “transaction” table has a foreign key “hotel_id”, reference the primary key in table “hotel”, that shows which hotel a transaction belongs to.
Transaction	RoomOrder	The “room_order” table has a foreign key “transaction_id”, reference the primary key in table “transaction”, that shows which transaction a room order belongs to.
Customer	Transaction	The “transaction” table has a foreign key “customer_id”, reference the primary key in table “customer”, that shows which customer a transaction belongs to.

There are also many-to-many foreign key mappings, which are shown in the next section.

6.1.9 Association table mapping

The difference between one-to-many relationship and many-to-many relationship is that one-to-many relationship can be achieved by using single-valued end of the association, because objects in many side only has one reference to the objects in one side, while in many-to-many relationship, one object at one side could hold multiple references to the other side. To solve this, we create a new table to represent the mapping.

A generic way to represent this is a table containing foreign keys of both sides that are linked with the association. Here’s an example from one implementation in our project:

This is two hotels from hotel table

<Table> hotel	
Hotel_id	Name
1562766569625649152	Ali demo hotel
1568962476380049408	Snowman

These are two amenities from hotel_amenities table

<Table> hotel_amenity	
Amenity_id	Description
1	Swimming Pool
3	Gym

Hotel with id 1562766569625649152 can have many amenities, amenity with id 1 can have multiple hotels

<Table> hotel_hotel_amenity	
Hotel_id	Amenity_id
1562766569625649152	1
1562766569625649152	3
1568962476380049408	1

6.1.10 Embedded value

In some domain models, it makes sense to have small objects for recording information that do not make sense to represent in a database.

In the system there are 2 main examples of the embedded value pattern: Money and Amenities.

Money:

To meet International objectives from the project specification, currency exchange is needed for the system. For the currency exchange, we need to define a class called *Money*. As is shown in figure.

```
public class Money implements Comparable<Money> {

    5 usages
    private BigDecimal amount;
    5 usages
    private String currency;

    996Worker
    @Override
    public int compareTo(Money other) {
        BigDecimal absoluteAmount = CurrencyUtil.convertCurrencyToAUD(this.currency, this.amount);
        BigDecimal otherAbsoluteAmount = CurrencyUtil.convertCurrencyToAUD(other.currency, other.amount);

        return absoluteAmount.compareTo(otherAbsoluteAmount);
    }
```

Figure 6.1.10.1 *Money class code snippet*

The requirement is that the frontend parameter will contain currency name, for example RMB, and then backend will return to frontend with corresponding money information, as is shown in the figure.

Figure 6.1.10.2 *Backend response snippet*

In the figure, we can see there is a field called "money", which is used for the front-end to render the correct currency info. It is easy to tell that it is not appropriate to store a "money" field in the database table, since the exchange rate is dynamic, and also there are too many currency names in the world.

To resolve the problem, we need to create the Money object based on dynamic currency from the backend, and map the Money object into the field of another object's table, for example, hotel object from the database. As a result, we can return the hotel object with an embedded money object to frontend. The example code is shown below:

```
Hotel hotel = getHotelEntityByHotelId(hotelId);
if (hotel == null || (!showNotSale && !hotel.getOnSale())) {
    throw new RequestException(
            StatusCodeEnume.HOTELIER_NOT_HAS_HOTEL.getMessage(),
            StatusCodeEnume.HOTELIER_NOT_HAS_HOTEL.getCode()
    );
}

// generate hotel vo
hotelVo = new HotelVo();
// copy properties
BeanUtil.copyProperties(hotel, hotelVo);

// embedded value
Money money = new Money();
money.setCurrency(currencyName);
money.setAmount(CurrencyUtil.convertAUDtoCurrency(currencyName, hotel.getMinPrice()))
hotelVo.setMoney(money);

// list amenities
List<HotelAmenity> amenities = hotelAmenityBlo.getAllAmenitiesByHotelId(hotelId);
hotelVo.setAmenities(amenities);

return hotelVo;
```

Figure 6.1.10.3 *embedded value example*

Amenities:

What's more, according to the code snippet, we can also find that the list object of hotel amenities is another embedded object to the hotel object.

The original *hotel* table doesn't have the field that holds the list of hotel amenities, which violates the atomicity law of database design. So we query the list of hotel amenities from the *hotel_amenity* table, and then wrap it into a list object, and then embed it into the hotel object based on the *hotel* table.

6.1.11 Concrete Table Inheritance Pattern

Concrete table inheritance represents an inheritance hierarchy of classes with one table per concrete class in the hierarchy.

For example, domain objects of user are using such a inheritance patterns, as is shown in the figure:

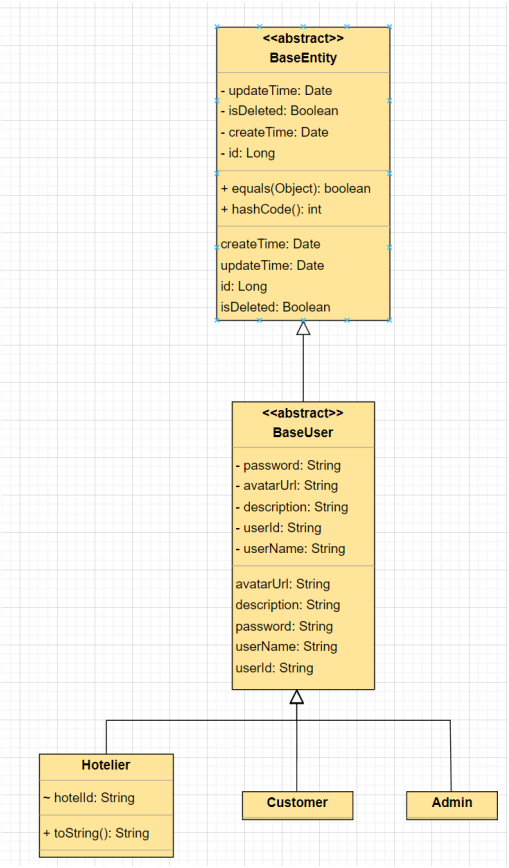

Figure 6.1.11 *entity class diagram for users*

(source: https://github.com/SWEN900072022/SWEN90007-2022-Alphecca/blob/main/docs/part2/images/Entity_Class_Diagram_user.png)

To prevent duplicated codes, an abstract entity class is created to map basic data such as CRUD time. An abstract class extended from the base abstract entity is created to map common data for users, such as username, password and so on. Then there are three concrete classes representing three type of users in the system. This class hierarchy is open for extension, and preventing errors caused by coping duplicated codes.

For three concrete user classes, we created three tables (hotelier, customer, admin) in the database to map them. Reasons are as follows:

- **Business considerations**: By dividing 3 tables, one user can login as customer, admin or hotelier with one same Email account. For example, if the hotelier is on vacation, he can also register a customer account with the same email to the system to book hotels. On the contrary,

if we have only one user table, since login email is unique, one user needs to prepare 3 different Emails for three types of users.

- **System considerations**: Concrete table inheritance is straightforward and simple: one concrete class to one concrete table, which is easy for developers to understand. Secondly, we don't need a join query to get full user data.

As a result, concrete table inheritance is used for the backend system.

6.1.12 Authentication and Authorization

To meet objectives including security and scalability, we decide to use **JSON Web Token** for the system authentication. What's more, since there are 3 types of users, and their responsibilities are divided clearly, we use **Role Based Access Control** in authorization.

For code implementation, we use the request filter to perform authentication/authorization logic ahead of all incoming requests at framework level. The class diagram is shown below:

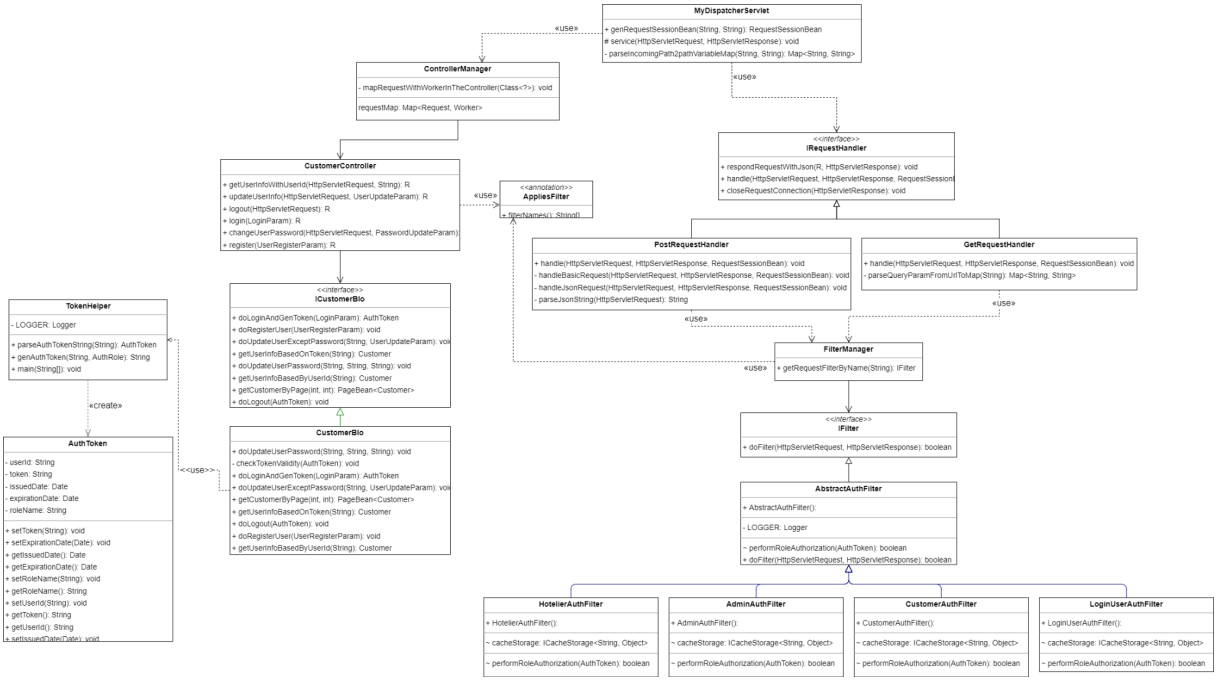

Figure 6.1.12 *security module class diagram*

(source: https://github.com/SWEN900072022/SWEN90007-2022-Alphecca/blob/main/docs/part2/images/Security_Model.png)

As is shown in the figure above, we defined multiple *AuthFilter* by strategy pattern. All concrete authentication filters are derived from the IFilter interface. At the starting stage of Alphecca Boot Framework, *ClassManager* and *FilterManager* will scan all filter classes, and *MyDispatcherServlet* will run all filter logic in *RequestHandler* when incoming requests occur.

In the *AuthFilter*, the token from the header will be parsed and checked for validity. Also the token will be checked whether it is expired or not by CacheUtil. If the *AuthFilter* is passed, the request will be handled. Otherwise, the 403 status code will be raised.

As for the token, all requests to the secured API will carry a header field "Authorization", which contains an encrypted token string. The token payload will be userId, role and expiration time. Filters that we mentioned before will validate the token, and grant access based on the role.

6.1.13 IoC, DI and AOP

Out of interest and to speed up development, we crafted a framework called Alphecca Boot, which is inspired by Spring Boot and other open source projects. This homemade framework introduced extra patterns including IoC (Inversion of Control), DI (Dependency Injection) and AOP (aspect oriented programming):

- **Inversion of Control and Dependency Injection**: Inversion of Control aims to reduce coupling of the system, and one way to achieve this is Dependency injection. When an object is created, it is passed references to the objects it depends on by some external entity that regulates all objects in the system. We implemented *ClassManager, BeanManager* and *InjectionHelper* to achieve IoC and DI. This pattern is used for all important components such as *Controller, Blo, Dao, Interceptor, Filter* and some utils.
- **Aspect Oriented Programming**: AOP uses dynamic proxy pattern to reuse logics like aspects, and enhance a target method with these aspect logic. We implemented AOP proxy in 3 ways: JDK proxy (used for classes implementing interfaces), Cglib proxy (used for classes without super classes or interface), and Bytebuddy(deprecated, still buggy in chain proxy). AOP helps a lot in the project:
    - JSON body parameter validation: We use an interceptor class to enhance all methods from controllers that contain *@RequestJsonBody* annotation. As a result, parameter objects will be validated with JSR303 standard.
    - Unit of work: We used *UnitOfWorkInterceptor* to enhance all methods from controllers, so that all controller methods will be wrapped with unit of work init and commit.

For more information and diagrams, please refer to **6.2.1 Libraries and Frameworks** section.

6.2 Source Code Directories Structure

Source code directories main structure is shown in the screenshot below:

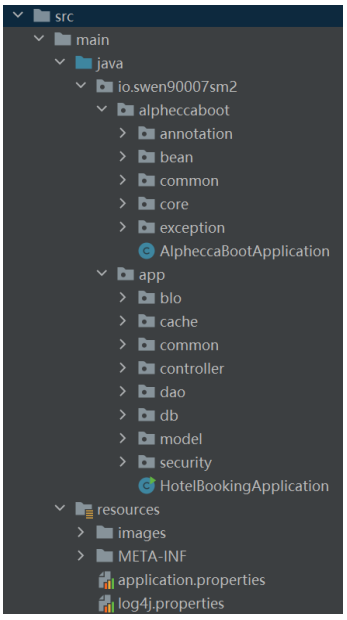

Figure 6.2.1 *Main source code structure*

io.swen90007sm2.alpheccaboot: This Directory holds all components of the Alphecca Boot Framework. This package is independent from the hotel booking application, and can be imported as a third-party module to other Java projects

io.swen90007sm2.app: This directory holds all components of the hotel booking application, including MVC objects, domains and utils. The detailed structure is shown below:

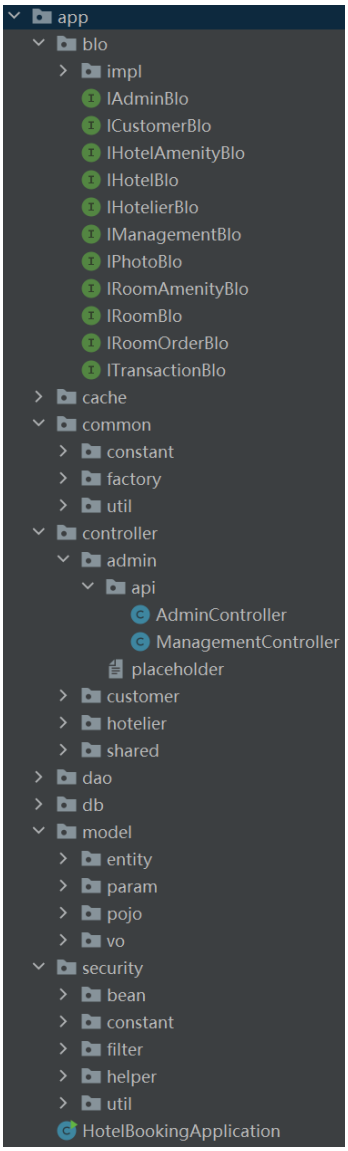

***Figure** 6.2.2 app source code structure*

For package details, please refer to the *Component* chapter.

resources: This folder contains two important configuration files:

- **application.properties**: configuration file, defines target base package, server metadata and database metadata;
- **log4j.properties**: configuration for log system. Can display logs in console and in files.

6.2.1 Libraries and *Frameworks*

This section describes libraries and *frameworks* used by the hotel booking system.

For the project, we developed our own framework from draft. The Alphecca Boot framework speeds up the backend development. This framework is inspired by Spring Boot Framework, and it includes some features such as IoC container, aspect oriented programming support, MVC annotation support and embedded Tomcat servlet container.

For the Alphecca Boot Supporter layer, according to the diagram, We used multiple design patterns including Strategy Pattern and Factory Pattern to make our system convenient for development.

The figure illustrates the UML class diagram of the Alphecca Boot framework layer for the system. For the high definition version, please see Github Repository.

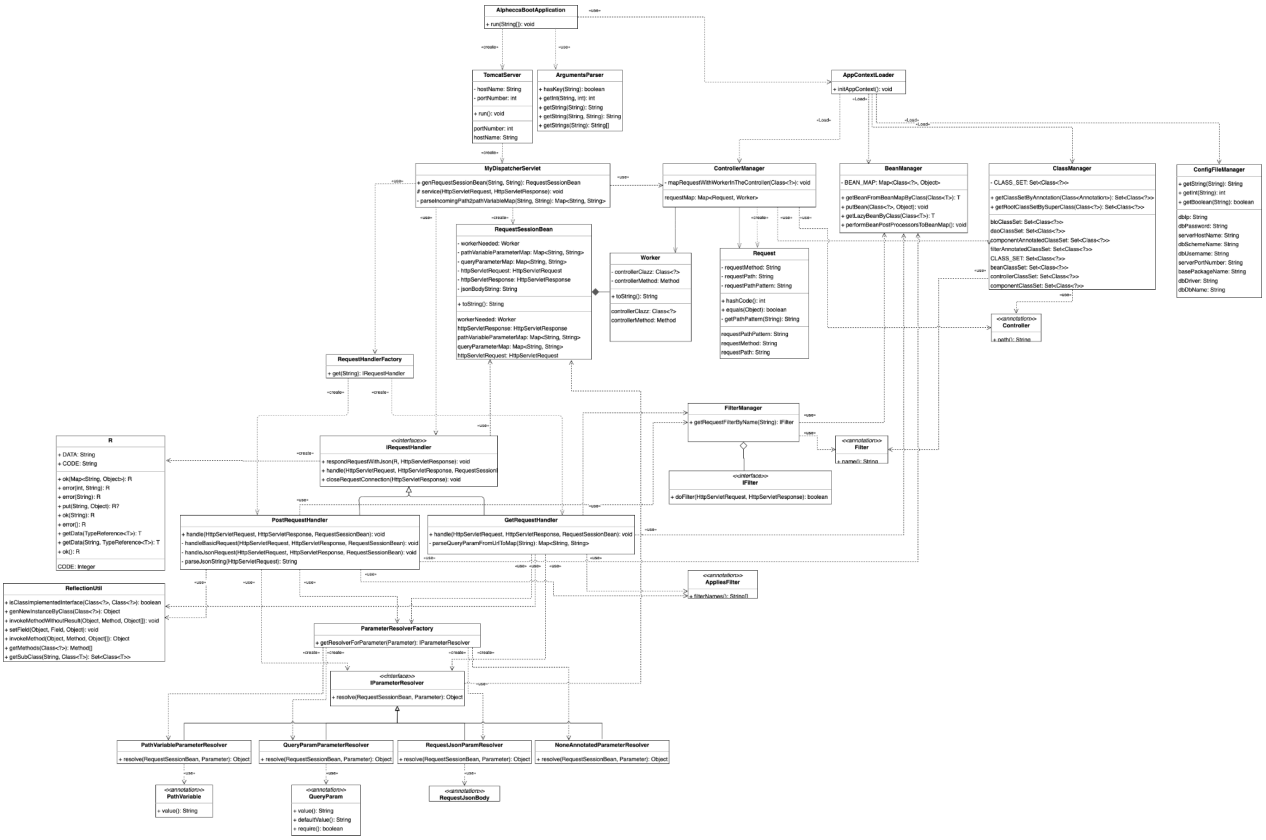

Figure 6.2.3 *Alphecca Boot Web Class Diagram*

(source: https://github.com/SWEN900072022/SWEN90007-2022-Alphecca/blob/main/docs/part2/images/Web.png)

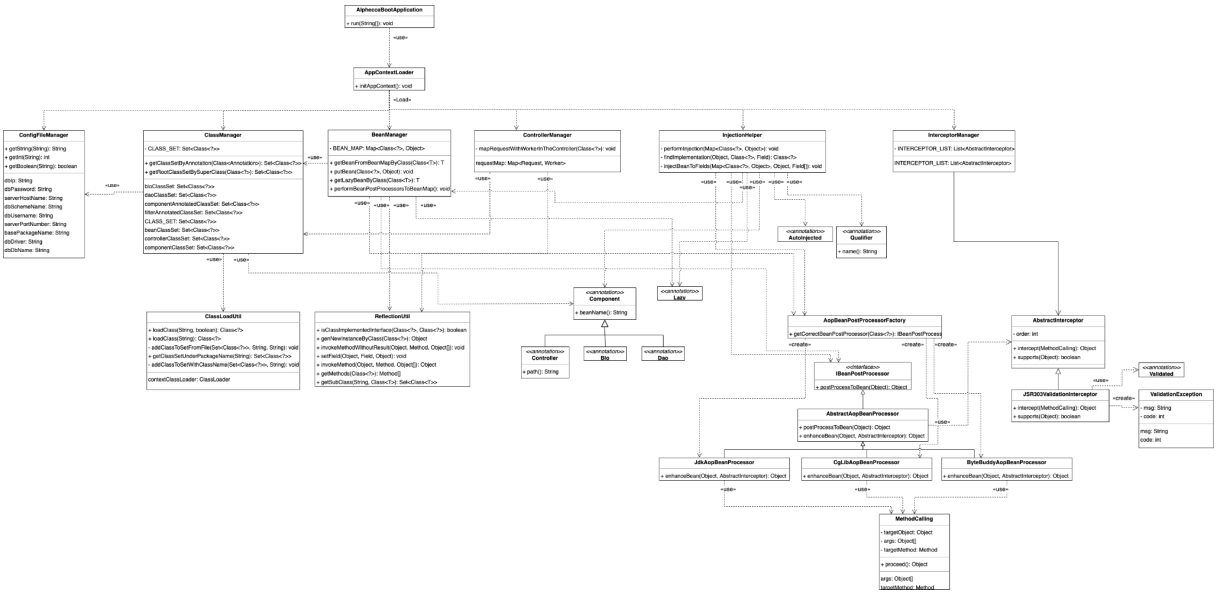

Figure 6.2.4 *Alphecca Boot IoC and AOP Class Diagram*

(source: https://github.com/SWEN900072022/SWEN90007-2022-Alphecca/blob/main/docs/part2/images/IoC%26AoP.png)

As we can see from figures, the Alphecca Boot Framework have 3 main functionalities:

- **Web MVC**: Alphecca Boot contains an embedded Tomcat Servlet Container, and handles http requests with *MyDispatcherServlet.* In *MyDispatcherServlet* 's *service* method, Factory Pattern and strategy pattern are used to perform request handling and parameter parsing. At each request handler, the *Filter* is applied for all requests, where we implement our authentication and authorization logic. After passing the filter, Alphecca Boot will map the request to the corresponding Controller method based on URL and request method, and then respond with the JSON body.
- **IoC & AOP:** Alphecca Boot uses *ClassManager* to scan all classes at the package path defined by configuration file, and automatically instantiate important component classes into *BeanMap* at *BeanManager*. So that, we make Alphecca Boot take responsibility for creating objects, and let the other business domains focus on their own logic. AOP is achieved by dynamic proxy. For classes that implemented interfaces, Alphecca Boot uses JDK proxy to enhance target classes; as for other conditions, Cglib library is used to generate enhanced byte code of the target class. AOP is helpful to deal with duplicated codes, we use AOP to achieve JSR303 parameter validation and Unit of Works logic.
- **Dependency Injection:** Dependency can be easily achieved by annotation *@AutoInjected.* For all *@Component* annotated classes, *InjectionManager* will scan their methods and find *@AutoInjected* annotation, then perform injection from *BeanMap*. *@Lazy* annotated component objects can not be auto injected and raise exceptions.

The diagram below shows the starting up workflow of Alphecca Boot:

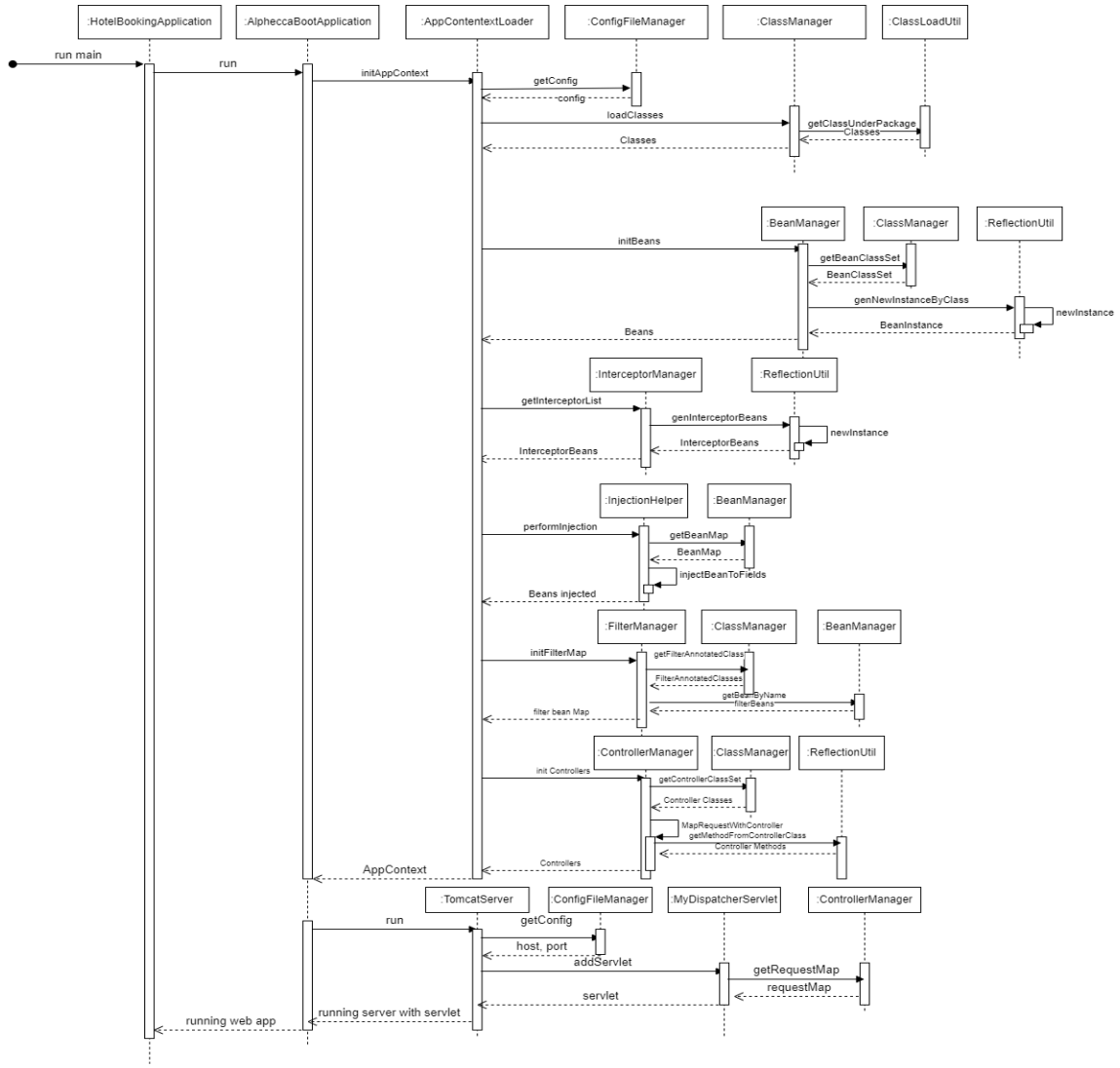

Figure 6.2.5 *Alphecca Boot starting up sequence diagram*

(source: https://github.com/SWEN900072022/SWEN90007-2022-Alphecca/blob/main/docs/part2/images/Sequence_Diagram_Framework.png)

According to the figure, We can divide starting up into several steps:

1. Parse configuration files;
2. Scan all classes from target base package path;
3. Instantiate *@Component* annotated classes, put objects in Bean Manager;
4. Enhance objects in Bean Manager with Interceptor by dynamic proxy;
5. Dependency Injection on *@AutoInjected* fields;
6. Mapping Controller with URL pattern and request methods;
7. init Tomcat server;
8. Attach Controller Mapping info with Servlet;
9. Application started.

After starting up, Alphecca Boot's Tomcat servlet container will listen to incoming requests and map them to correct Controllers.

What's more, Alphecca Boot has no coupling with the hotel booking application. As a reusable module, Alphecca Boot can be used in any Java MVC application with correct configuration files.

The table below describes other third-party frameworks and libraries for this project.

Library / *Framework*	**Reason**	**Version**	**Environment**
commons-dbutils	Decision Making: It is a popular library for JDBC connection pool. We should use the connection pool rather than write sql communication by hand	1.7	All
postgresql	The only option: PostgreSQL Java Driver for JDBC	42.4.1	All
javax.servlet-api	The only option: Java standard servlet library	4.0.1	All
tomcat-embed-core	Decision Making: It is convenient to use embed servlet container to simplify development	8.5.76	All
commons-lang3	Decision Making: helpful utils collection from Apache	3.12.0	All
fastjson	Decision Making: popular JSON encode/decode library	2.0.7	All
reflections	The only option: Java Reflection api	0.10.2	All
httpcore	The only option: implementation of servlet	4.4.15	All
hibernate-validator	The only option: JSR303 validation tool	6.2.0	All
cglib	Decision Making: popular dynamic proxy	3.3.0	All
java-jwt	Decision Making: popular jwt token util	3.19.2	All
jbcrypt	Decision Making: popular encrypter	0.4	All
hutool-core	Decision making: popular reusable utils library	5.8.5	All
bootstrap	Decision making: Easy to use and many useful components ready to use for the front end of the project.	5.2.0	All
Nginx (Docker)	Decision making: Popular Server for frontend app deployment in production. Will use Nginx Docker image.	alpine	Production

6.2.2 Development Environment

Name	Description
IntelliJ IDEA 2022.2 Ultimate Edition	Java IDE, support database operation
JDK 11	JDK version 11 is Heroku friendly, and also support Cglib library
React.js	Framework for frontend to speed up development
Vscode	Code Editor for frontend.

7 Physical View

This section describes the hardware elements of the system and the mapping between them and the software elements.

7.1 Deployment Diagram

The Alphecca Hotel Booking system deployed the frontend and backend to the Heroku platform. As for data persistence, Heroku Postgres service is used, and has been attached to backend applications. As is shown in the diagram below.

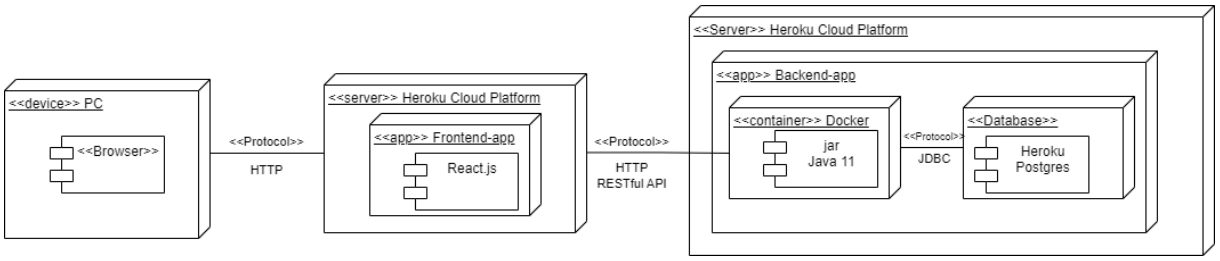

Figure 7.1 *System deployment diagram*

(source: https://github.com/SWEN900072022/SWEN90007-2022-Alphecca/blob/main/docs/part2/images/System_Deployment.png)

According to the diagram, Http RESTful APIs are used for communication between frontend and backend, which can be convenient for developers to set up regulations and API reference.

Docker is used for the deployment of the system, this is because docker can provide unified runtime environment for applications, and it is convenient to deploy the application to multiple devices easily.

7.2 Development Environments

This section describes the development and production environment.

Environment	Description
development env	• Run with IDEA IDE from *io.swen90007sm2.app.HotelBookingApplication* entrance class • Dependency is managed by Maven • Database is declared in *application.properties* configuration file • Can be run in any platform that implemented Java Virtual Machine
production env	• Compiled and run with Docker.

7.2.1 Hardware

Hardware requirements are discussed below:

- Backend projects can run on any device/platform that implements Java Virtual Machine.
- Frontend projects can run on any device/platform that supports Node.js.

As for detailed hardware requirements, see JVM and Node.js documentation.

7.2.2 Software

- Backend and frontend projects can run with IDEs of developers.
- Before starting the application, dependencies should be installed. Backend is using Maven, Frontend is using Npm.

Note:

The project can run without problems in Java runtime version 11.

For later version Java, please add JVM options when run the project source code: *--add-opens java.base/java.lang=ALL-UNNAMED*.

For example:

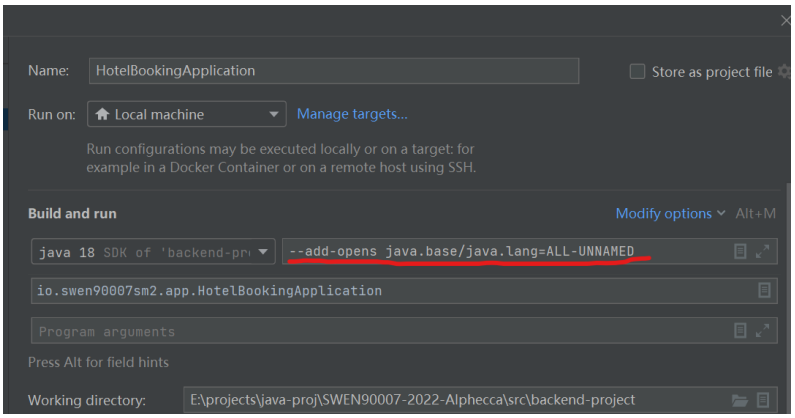

Figure 7.2.2 *JVM option for JREs that version*

This is because in Java 16+, internal Java APIs are blocked due to JEP 396 standard from Java 16. This restriction will cause CgLib dynamic proxy to throw *protected classes* exception, and then shutdown the JVM.

For Heroku and CgLib friendly, Java 11 is recommended for Project JDK and Application JRE.

7.3 Development Environment

The backend and frontend application will be compiled into Docker images, and be released to Heroku platform as a runtime container.

For the backend project, using *maven:3.8.6-jdk-11-slim* docker image to compile and build the jar with Maven, and then run the application with *adoptopenjdk/openjdk11:jre-11.0.9_11.1-alpine* image. The port mapping follows Heroku documentation. Then we can release the Docker Image to Heroku with the .sh script created by the backend team.

Important Scripts are shown as follows:

Script	Description
src/backend-project/Dockerfile	Backend Docker building
src/backend-project/deploy-heroku.sh	Automatically deploy the docker image to the Heroku Platform

The Frontend project has the same Script structure as we have discussed above.

7.4 Links

Github Repository:

- https://github.com/SWEN900072022/SWEN90007-2022-Alphecca

The backend API:

- URL: https://swen90007-alphecca-backend-app.herokuapp.com/

The RESTful API documentation:

- URL: api.996workers.icu
- Username: 90007-visitor@996workers.icu
- Password: qD8EjKzS6sPJ8BeVcMNF

The frontend address:

- URL: https://swen90007-alphecca-frontend.herokuapp.com/

The admin address:

- URL: https://swen90007-alphecca-frontend.herokuapp.com/adminLogin

All the data sample has been included in the github repository under docs/data-samples.

8 Scenarios

The description of an architecture is illustrated using a small set of use cases, or scenarios. The scenarios describe sequences of interactions between objects and between processes.

8.1 Use Case Diagram

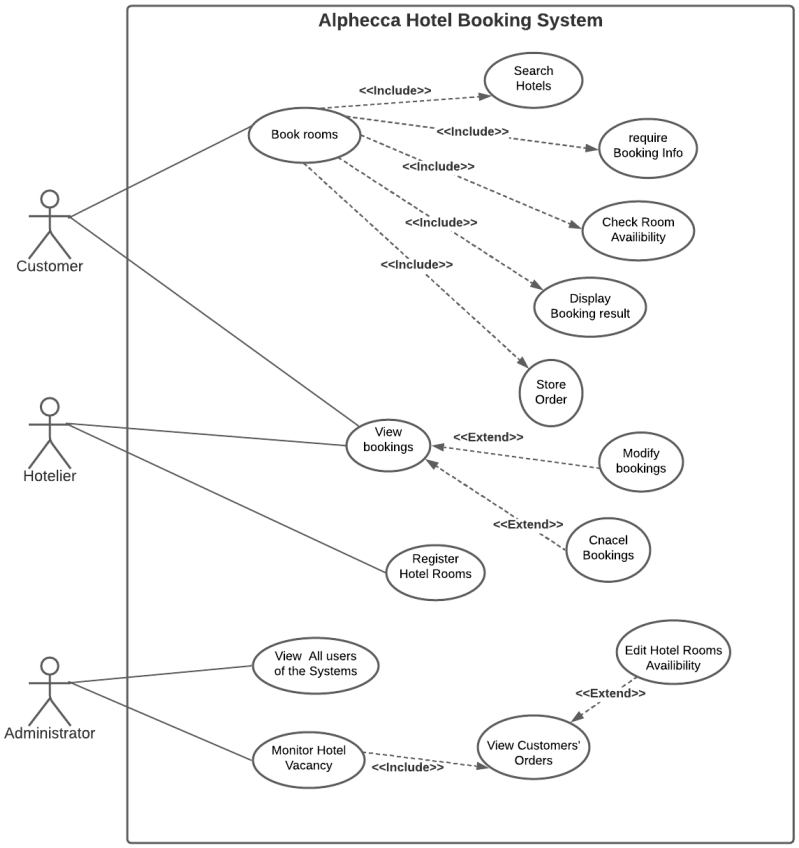

Figure 8.1 *Use Case Diagram*

(source: https://github.com/SWEN900072022/SWEN90007-2022-Alphecca/blob/main/docs/part2/images/hotel%20use%20case.png)

9 User Manual

9.1 Log in as an Customer

- Enter https://swen90007-alphecca-frontend.herokuapp.com in the web browser (Chrome preferred).
- The user can search hotels or enter the user portal straight away.
- Search Hotel
    - For now, the user can only search hotels by either hotel name or post code. Then the user can select preferred check in and check out dates, corresponding residents and rooms.
    - For testing purposes, the user can enter “demo” in the hotel name field or “3003” in the postcode field.

    - Clicking the search button, the user will be directed to a new page showing all the results.
    - Clicking on the Check Details button, the user will be directed to a new page showing the details of the hotel and the types of room that the hotel offers.
    - The user is able to reserve a hotel room by clicking on the "Book Now!" button. If the user is signed in, the booking is reserved under the user's account. Otherwise, the user will be directed to the signin page.
- Enter the User Portal
    - Before entering the user portal, the user needs to be signed in.
    - The user can login by clicking the "Sign in/Sign up" button in the home page.
    - For testing purposes, the user can enter "edisonTest@gmail.com" in the Email Address field with the corresponding password "yls123456" to log into the user portal.
    - When the user is logged in, the user will be directed to the home page.
    - The user can click on the user icon located on the top right of the webpage to enter the user portal.
    - In the user portal, the user can be able to view his/her prior bookings or future bookings by clicking on the "Prior Booking" or "Future Booking" tabs correspondingly.
    - The user is able to edit the future bookings in the "Future Booking" tab by simply clicking on the "Edit" button. The user is able to choose a new check in and check out dates and change the number of people visiting. By clicking on the "Reserve Now!" button, the new booking information will be recorded.
    - The user is able to cancel the future booking by simply clicking on the "Cancel" button.

9.2 Log in as an Hotelier

- Enter https://swen90007-alphecca-frontend.herokuapp.com in the web browser
- User can press the sign in/up button and choose "I'm a Hotelier" role card to login
- Create new hotel (login needed)
    - For hoteliers who are new to our website, there will be a button on the hotelier home page to allow them to register a new hotel.
    - To create a hotel, some information is required. A window will pop up asking hoteliers to enter the hotel name, address, post code etc after the register button is clicked. Make sure all information is correctly entered, then the hotel information will be uploaded to the system after they press the submit button.
- Edit hotel information (login needed)
    - Hoteliers can always update details of owned hotels by clicking the edit button.

- Add room types and edit room (login needed)
    - A new Hotel will not have any room and room details exist. Hoteliers need to create a new room type by clicking the add room button. and a pop-up window will allow them to enter further required information for a room
    - Room types created for this hotel will be displayed in the room detail section. if there is more rooms available for booking, hotelier can edit it and upload to the hotel system

- View transactions (login needed): hoteliers can view all transactions of this hotel by clicking the transaction button. All transactions will be presented on the transaction page. Each transaction shows the details of the customer and the status of the transaction. if customer

canceled the booking, the status would be “cancel” on the page, while “process” status means that the booking is still pending.

9.3 Log in as an Admin

- Admin can login to their portal by entering the following link.
  https://swen90007-alphecca-frontend.herokuapp.com/adminLogin
  A existing admin account: admin@demo.com password: 123456
- The website will direct to the admin portal after successful login. the first page of the portal lists all customers of the system
- To view all hotels registered in the system, admin needs to click the “Hotel” tab on the nav bar.
  - On this page, the admin can change the status of the hotel by clicking the open/close buttons under the status column. if the hotel is open, this hotel would be closed by clicking the status button.
  - To add/remove hoteliers to one hotel, admin can click the corresponding edit button in the row. It will have a pop-up window open to show the user id of existing hoteliers who manage this hotel. Press the red remove button to delete this hotelier from this group. Click the add button, an input field will appear and the admin can enter the user id to add a new hotelier.

- Hotelier tab on the nav bar will navigate to the page that displays basic details of all hoteliers. admin can add a new hotelier by entering the user name, email address and password.

10 References

Kruchten, P. B. (1995). The 4+ 1 view model of architecture. *IEEE software*, *12*(6), 42-50.

“4+1 architectural view model”, En.wikipedia.org, 2020. [Online]. Available: https://en.wikipedia.org/wiki/4%2B1_architectural_view_model.