# Architecture Document

**SWEN90007**

**Travel Reservation System**

Team: Four Aces

**My Tien Hinh** - 923427 - mhinh@student.unimelb.edu.au
**Xueqi Guan** - 1098403 - xueqig@student.unimelb.edu.au
**Yiyuan Wei** - 793213 - yiyuanw1@student.unimelb.edu.au
**Yuxiang Wu** - 1006014 - yuxiang2@student.unimelb.edu.au

**Revision History**

| Date | Version | Description | Author |
|---|---|---|---|
| 21/09/02 | 02.00-D01 | Adjust and add domain model | Everyone |
| 21/09/11 | 02.00-D02 | Data mapper | Xueqi |
| 21/09/11 | 02.00-D03 | Front Controller and Domain model | My Tien |
| 21/09/11 | 02.00-D04 | Lazy Load | Yiyuan |
| 21/09/11 | 02.00-D05 | Unit of Work | Yuxiang,Xueqi |
| 21/09/16 | 02.00-D06 | Component Diagram<br>Component Diagram Class table | Xueqi,Yuxiang |
| 21/09/20 | 02.00-D07 | Data Mapper Sequence Diagram | Xueqi |
| 21/09/20 | 02.00-D08 | Unit of work Sequence Diagram | Yuxiang |
| 21/09/21 | 02.00-D09 | Lazy Load Sequence Diagram | Xueqi |
| 21/09/23 | 02.00-D10 | Database Diagram | Xueqi, Yuxiang |
| 21/09/23 | 02.00-D11 | Identity Field, Foreign key mapping | Yuxiang,Xueqi |
| 21/09/24 | 02.00-D12 | Association Table Mapping | Xueqi,Yuxiang |
| 21/09/26 | 02.00-D13 | Inheritance Pattern | Yuxiang,Xueqi |
| 21/09/26 | 02.00-D14 | Authentication description<br>Inheritance Pattern, Deploy diagram | Yuxiang |
| 21/09/27 | 02.00-D14 | Finalise the whole document | Everyone |

## Contents

# Introduction

This document specifies the architecture of the travel reservation system, describing its main standards, modules, components, frameworks and integrations.

## 1.1 Proposal

This project simulates a Travel Reservation System, a commercial platform for customers to search for and book flights across Australia or internationally. Customers can easily find information about their bookings from the system. The system also allows airlines to manage their flights, ticket options and reservations from their customers. All users and activities are managed by an administrator to ensure the system is working correctly.

The purpose of this document is to provide an overview of the system. High level design and architecture is captured in the use cases (Section 3), domain model description and diagram (Section 4). This serves as a common ground of communication among the development team members, as well as among the team, SWEN90007 teaching team and clients.

## 1.2 Target Users

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

### 1.1 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 |
|---|---|
| Airplane | The class encapsulating details of the airplane used in the flight |
| Flight | The trip on the airplane |
| Passenger | Customers book tickets for passengers who will take flights. |
| Ticket | A ticket contains a passenger's information and seat number. |
| TRS | Travel Reservation System |

# Architectural representation

The specification of the travel reservation system architecture 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.

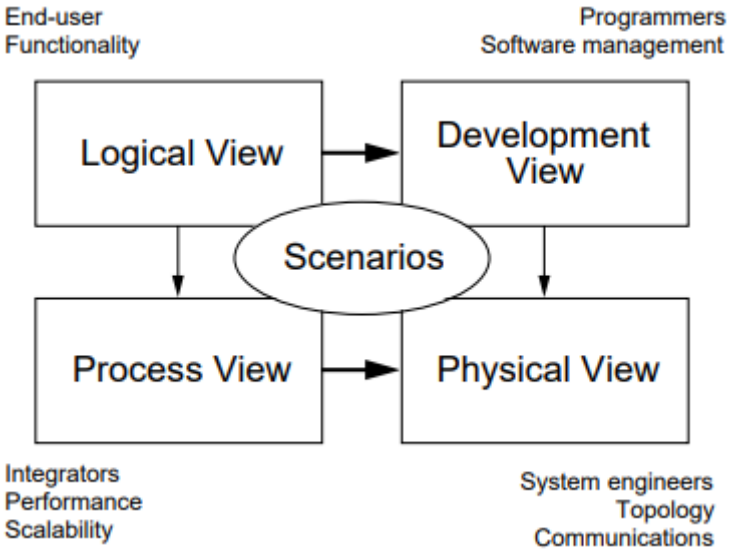


**Figure 1.** Views of *framework* "4+1"

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

## 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.

### 3.1 Class Diagram

Below is the class diagram of our system:

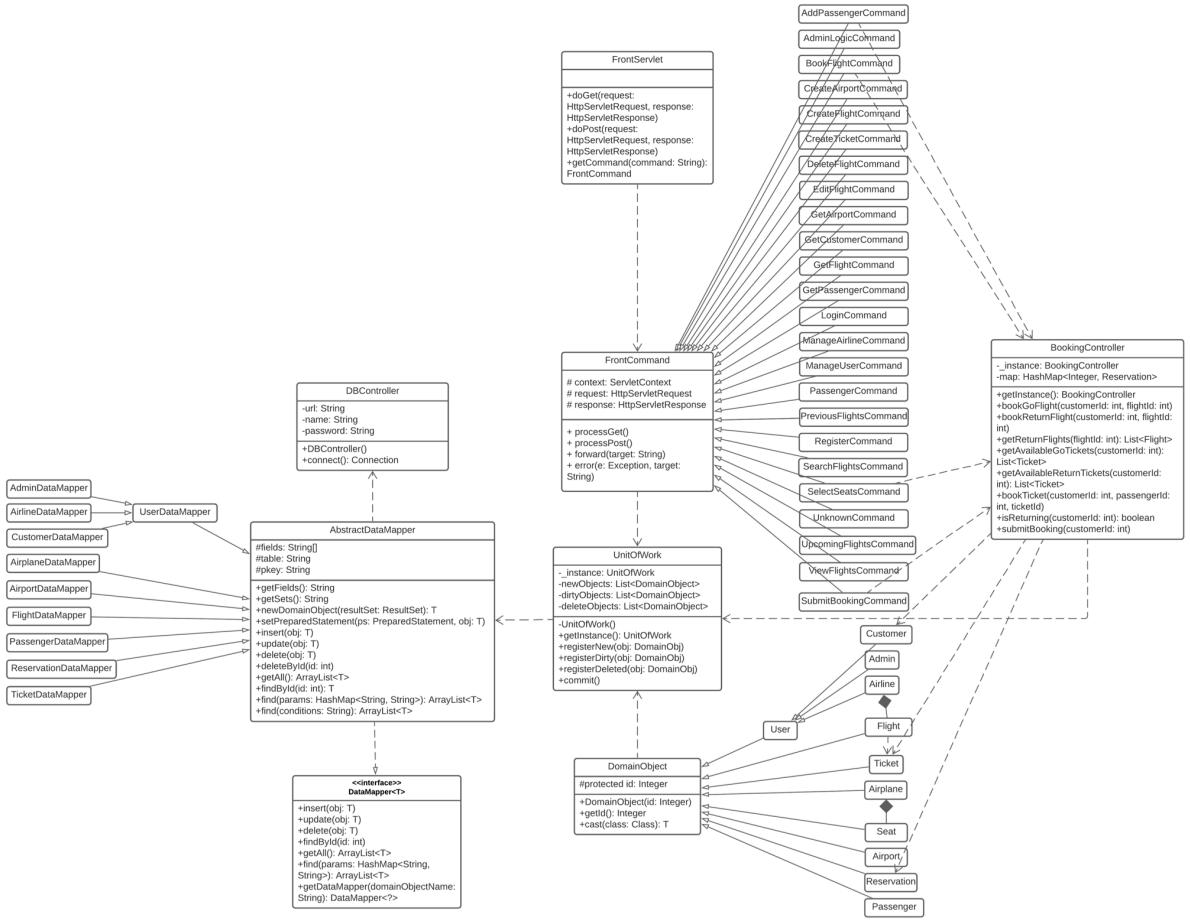


**Figure 2.** Class diagram

## 1.3 Process View

The section shows the mapping of the logical architecture elements to the processes and threads of the system execution.

## 4.1 Sequence Diagram

### 4.1.1 Front Command Sequence Diagram

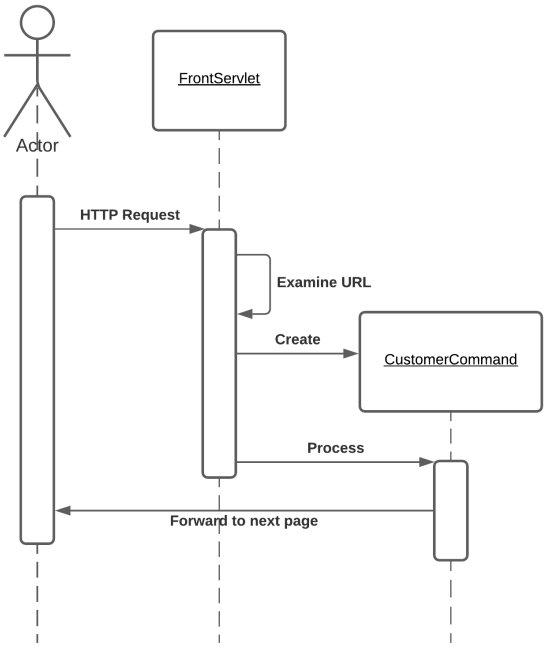


**Figure 3.** Front Command Sequence Diagram

### 4.1.2 Lazy Load & Data Mapper Sequence Diagram

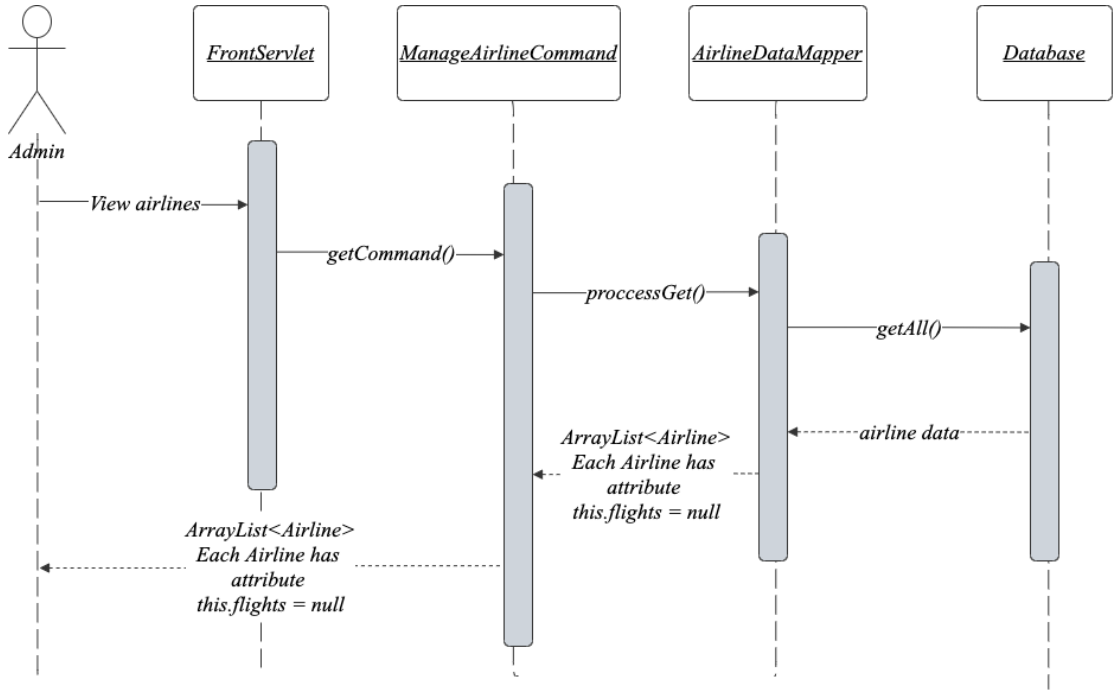


**Figure 4.** Lazy Load & Data Mapper Sequence Diagram

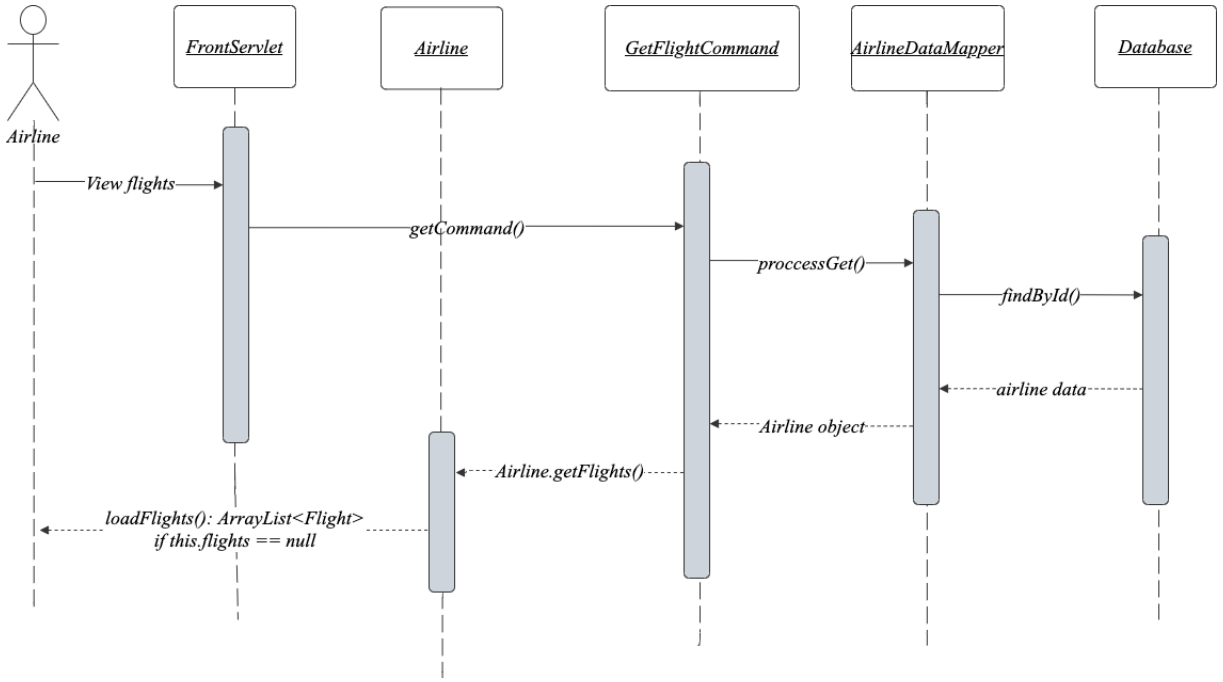


**Figure 5** Lazy Load & Data Mapper Sequence Diagram

### 4.1.3 Unit of Work Sequence Diagram

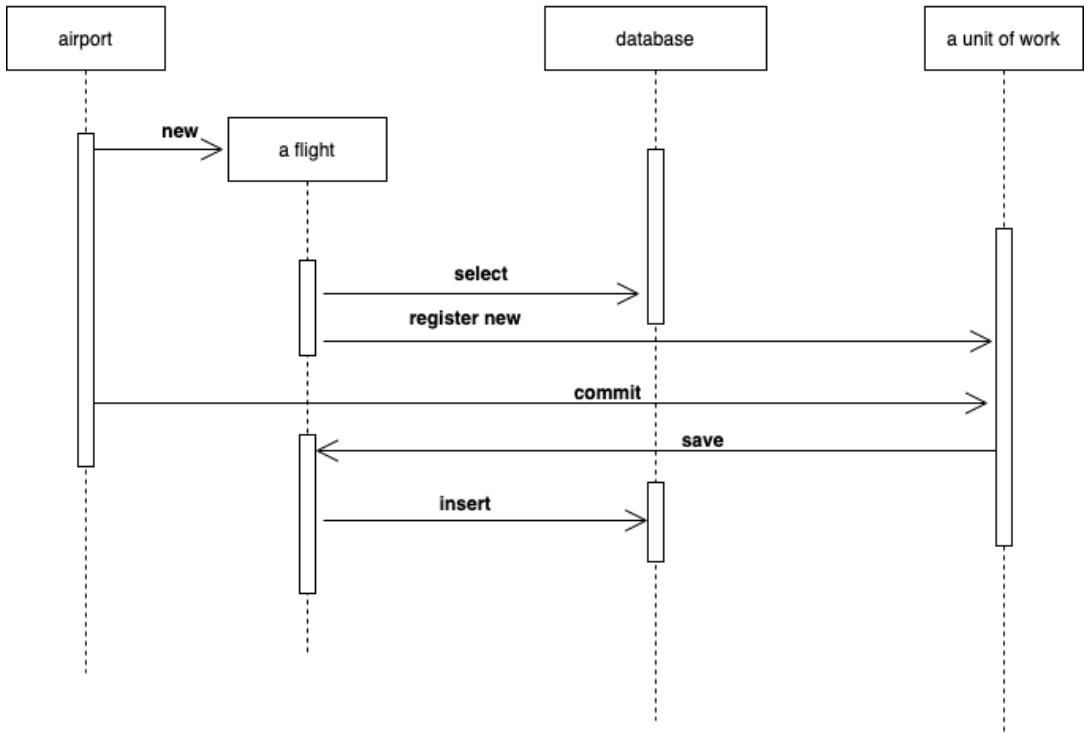


**Figure 6.** Unit of Work Sequence Diagram

### 4.1.4 Authentication and Authorization

## Development View

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

### 5.1 Component Diagram

The component diagram describes how components are connected to each other.

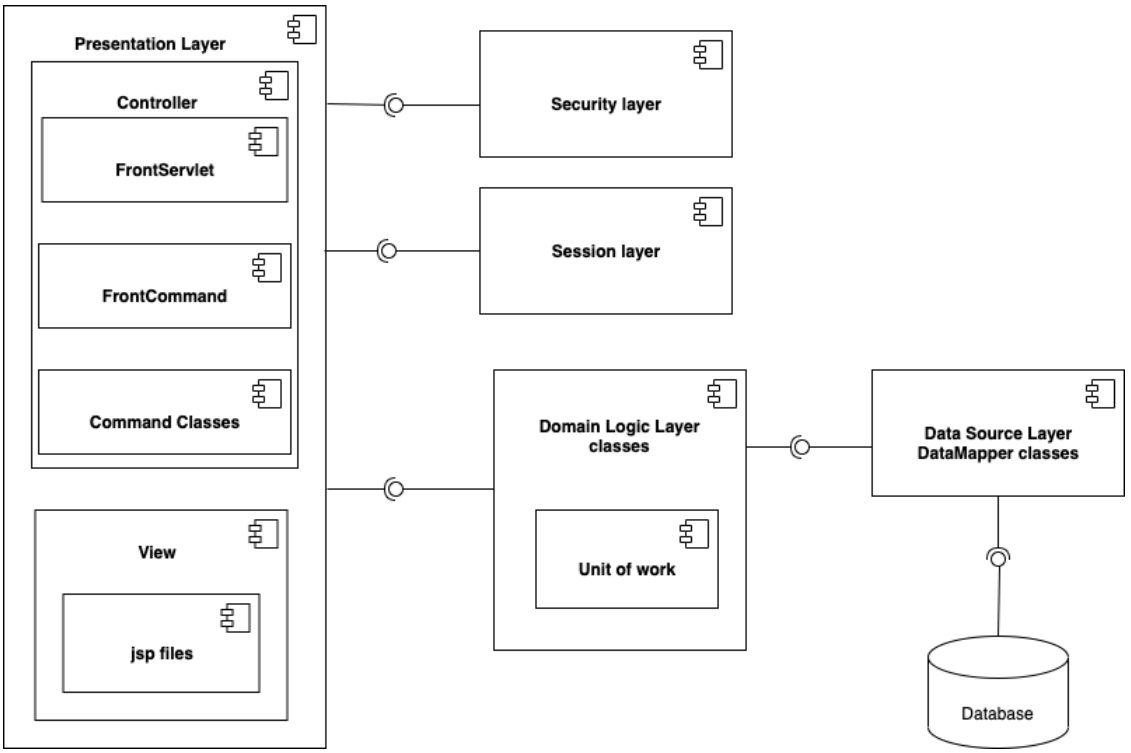


**Figure 7.** Component Diagram

| Command Classes | Domain Logic Layer Classes | Data Source Layer Classes |
|---|---|---|
| ViewFlightCommand.java | Admin.java | AbstractDataMapper.java |
| GetFlightCommand.java | Aline.java | AdminDataMapper.java |

| | | |
|---|---|---|
| BookFlightCommand.java | Airplane.java | AirlineDataMapper.java |
| DeleteFlightCommand.java | Airport.java | AirportDataMapper.java |
| EditFlightCommand.java | Booking.java | AirplaneDataMapper.java |
| SearchFlightsCommand.java | Customer.java | CustomerDataMapper.java |
| GetPassengerCommand.java | DomainObject.java | DataMapper.java |
| PassengerCommand.java | Flight.java | FlightDataMapper.java |
| AddPassengerCommand.java | Passenger.java | PassengerDataMapper.java |
| GetCustomerCommand.java | Reservation.java | ReservationDataMapper.java |
| CreateAirportCommand.java | Seat.java | TicketDataMapper.java |
| GetAirportCommand.java | Ticket.java | UserDataMapper.java |
| CreateTicketCommand.java | UnitOfWork.java | |
| ManageAirlineCommand.java | User.java | |
| ManageUserCommand.java | | |
| AdminLoginCommand.java | | |
| RegisterCommand.java | | |
| LoginCommand.java | | |
| SubmitBookingCommand.java | | |
| UnknownCommand.java | | |

| | | |
|---|---|---|
| FrontCommand.java | | |

**Table 1.** Component Diagram Class Specification

## 5.2 Architectural Patterns

This section describes all architectural patterns which are used in the system, lists the pattern adaptations and provides examples of use in the system.

### 5.2.1 Domain Model

When faced with a decision on which pattern to use for the domain logic, we had 3 options: transaction script, table module and domain model. Transaction script is a basic way of developing the system and can be used in simple transactions. However, the TRS has very complex business logics that would be very difficult to fully cover in transaction scripts. Table module is more object-oriented by having each database table represented by one class. However, the class has to handle every instance of the object. The complexity scales dramatically as the TRS has to handle numerous instances per object, for example, there can be thousands of flights being used. Therefore, domain model is the most sensible choice.

Our domain model is designed to resemble the business logic between objects in the system. Each object behaves like a real object as part of the system. For example, each Flight has its own attributes like destination, airplane,... and can be made available for Customers to book. It allows for many actions to be performed at the same time. When the system is changed or extended, classes can be modified or added in to reflect the new logic without the need to revise the whole system.

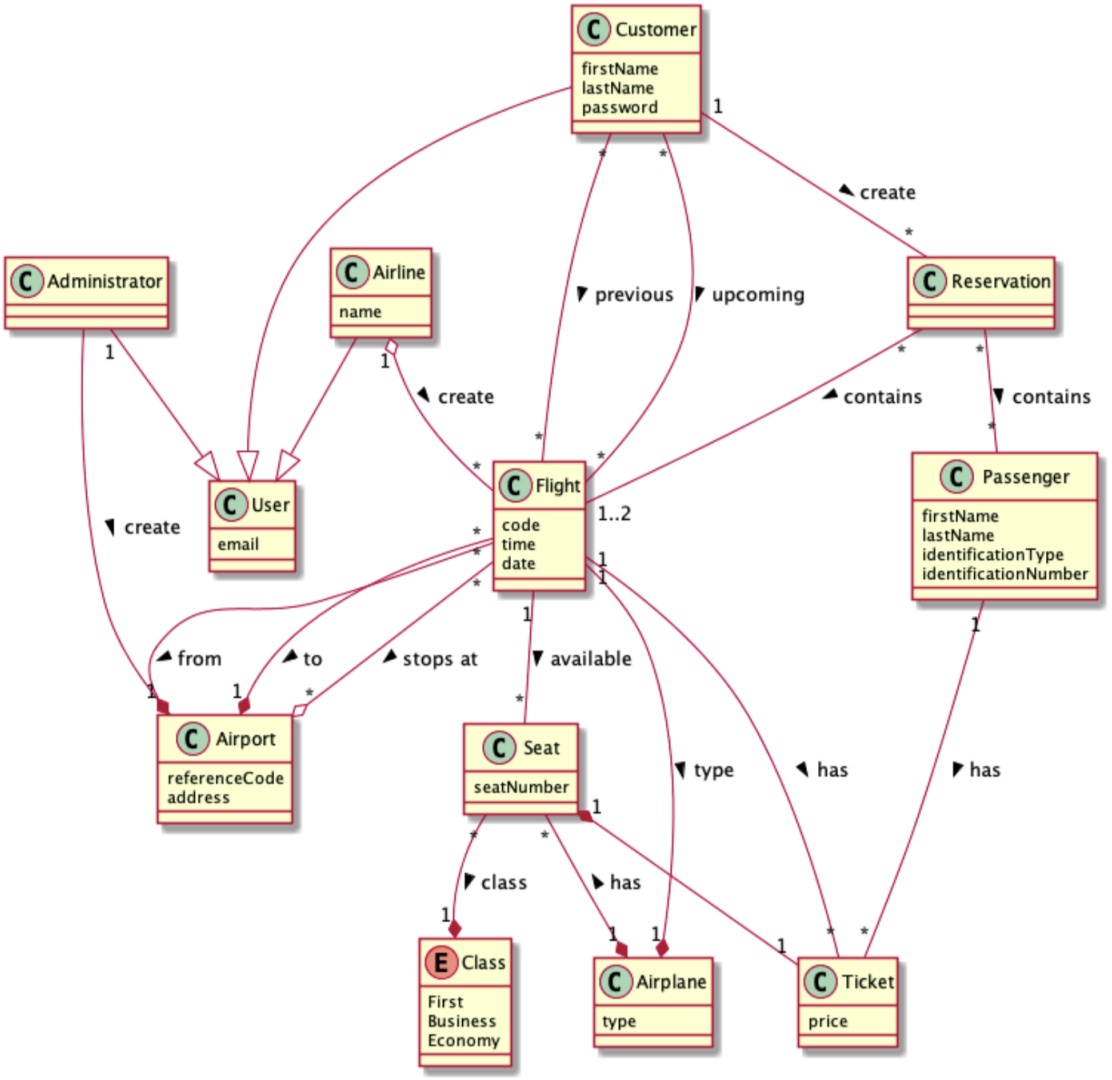


**Figure 8.** Domain Diagram

### 5.2.2 Data Mapper

Data Mapper is the pattern which can separate the domain objects from the database persistent data. We apply this pattern to our TRS system that can transfer the data from the domain layer to the data source layer allowing our domain layer proceeding design independently of the database layer.

Example of use:

In our project, we created a package called “datasource” that contains a generic DataMapper interface, an abstract AbstractDataMapper class and concrete data mappers. The DataMapper interface defines abstract insert, update, delele, find, getAll and getDataMapper methods. The AbstractDataMapper class implements the interface, provides implementation methods declared in the interface and defines two abstract methods: newDomainObject and setPreparedStatement. Method newDomainObject takes the response provided by the database as its parameter and returns a domain object. setPreparedStatement is used for setting parameters in SQL query statements. All the concrete data mappers classes are singleton classes. They inherit from the AbstractDataMapper and override abstract methods in the parent class.

| Data Mappers | Methods |
|---|---|
| DataMapper <Interface> | insert, update, delele, find, getAll and getDataMapper |
| AbstractDataMapper | insert, update, delete, find, newDomainObject, setPreparedStatement |
| AdminDataMapper | getInstance, newDomainObject, setPreparedStatement |
| AirlineDataMapper | getInstance, newDomainObject, setPreparedStatement |
| AirportDataMapper | getInstance, newDomainObject, setPreparedStatement |
| AirplaneDataMapper | getInstance, newDomainObject, setPreparedStatement |
| CustomerDataMapper | getInstance, newDomainObject, setPreparedStatement |
| FlightDataMapper | getInstance, newDomainObject, setPreparedStatement |
| PassengerDataMapper | getInstance, newDomainObject, setPreparedStatement |
| ReservationDataMapper | getInstance, newDomainObject, setPreparedStatement |
| TicketDataMapper | getInstance, newDomainObject, setPreparedStatement |
| UserDataMapper | getInstance, newDomainObject, setPreparedStatement |

**Table 2.** DataMapper Classes Table

### 5.2.3 Unit of Work

The unit of work acts as a dump of all transaction processing codes. The responsibility of the unit of work managing transactions, such as inserting, deleting, and updating data in database. It is also used to prevent repeated updates. The value of the unit of work pattern is to free the rest of your code from worrying about making changes to the database so that you can otherwise focus on business logic.

The unit of work keeps track of three lists of objects

- newObjects: newly created objects that will be inserted in database
- dirtyObjects: modified existing objects that will be updated in database
- deleteObjects: existing objects that will be removed in database

The unit of work uses registerNew, registerDirty and registerDeleted to add objects to the above three lists. It uses the commit method to commit all changes to the database.

- registerNew(): add newly created objects to newObjects list
- registerDirty(): add modified existing objects to dirtyObjects list
- registerDelete(): add deleted objects to deletedObjects list
- commit(): loop through the above three lists and commit all changes to database

Example of Use: In our system, we use the unit of work to insert, update and delete records in the database. For example, in Flight class, we applied the unit of work to process updating flight code, date, time, departure and destination to save the cost of many times interactions with the database.

| Classes | Operation | Implement Principle |
|---|---|---|
| Admin | insert | Add the new Admin to the newObjects list. |
| Airline | insert | Add the new Airline to the newObjects list. |
| Airport | insert | Add the new Airport to the newObjects list. |
| Customer | insert | Add the new Customer to the newObjects list. |
| Flight | insert, update, delete | Add the new Flight to the newObjects list.<br>Add the changed Flight to the dirtyObjects list.<br>Add the deleted Flight to the deleteObjects list. |
| Passenger | insert | Add the new Passenger to the newObjects list. |
| Ticket | insert, delete | Add the new Ticket to the newObjects list.<br>Add the deleted Ticket to the dirtyObjects list |
| Reservation | insert, delete | Add the new Reservation to the newObjects list.<br>Add the deleted Reservation to the dirtyObjects list. |

**Table 3.** Unit of Work Table

### 5.2.4 Identity Field

In programming languages, objects and data structures are uniquely identified by their memory addresses. In a relational database, rows are uniquely identified using their keys. When objects in memory are associated with rows in a relational database, identity field pattern can be used to ensure that the correct objects are written back to the correct rows. By applying the identity field pattern, for any object which is associated with a row, the object will store the primary key of the associated row as one of its attributes.

Example of Use: In our system, we define an abstract DomainObject class. All the objects with an unique ID inherit from the DomainObject class. The id is an auto increment ID generated by the database.

### 5.2.5 Foreign Key Mapping

Since some objects contain references to other objects in the domain layer, it is necessary for those references to persist in the database. This pattern maps an association between objects to a foreign key reference between tables.

Example of Use: The Ticket object has the flight code as a foreign key referencing the Flight object, and the Ticket object has the seat ID as foreign keys to reference the ticket object.

### 5.2.6 Association Table Mapping

The association table mapping saves an association as a table with foreign keys to the tables that are linked by the association. For many-to-many relationships, an associative table is created to represent the mapping between two tables. The associative table will store the primary keys from the two tables it joins as its foreign keys and use the combination of two foreign keys as its primary key.

Example of Use: There are two many-to-many relationships in our system. The Customer table and the Flight table are linked by the CustomerFlight table which contains the customer ID and flight ID as foreign keys. The Reservation table and the Passenger table are linked by the PassengerReservation table which contains the Reservation ID and Passenger ID as foreign keys.

The table below is the database diagram of our system.

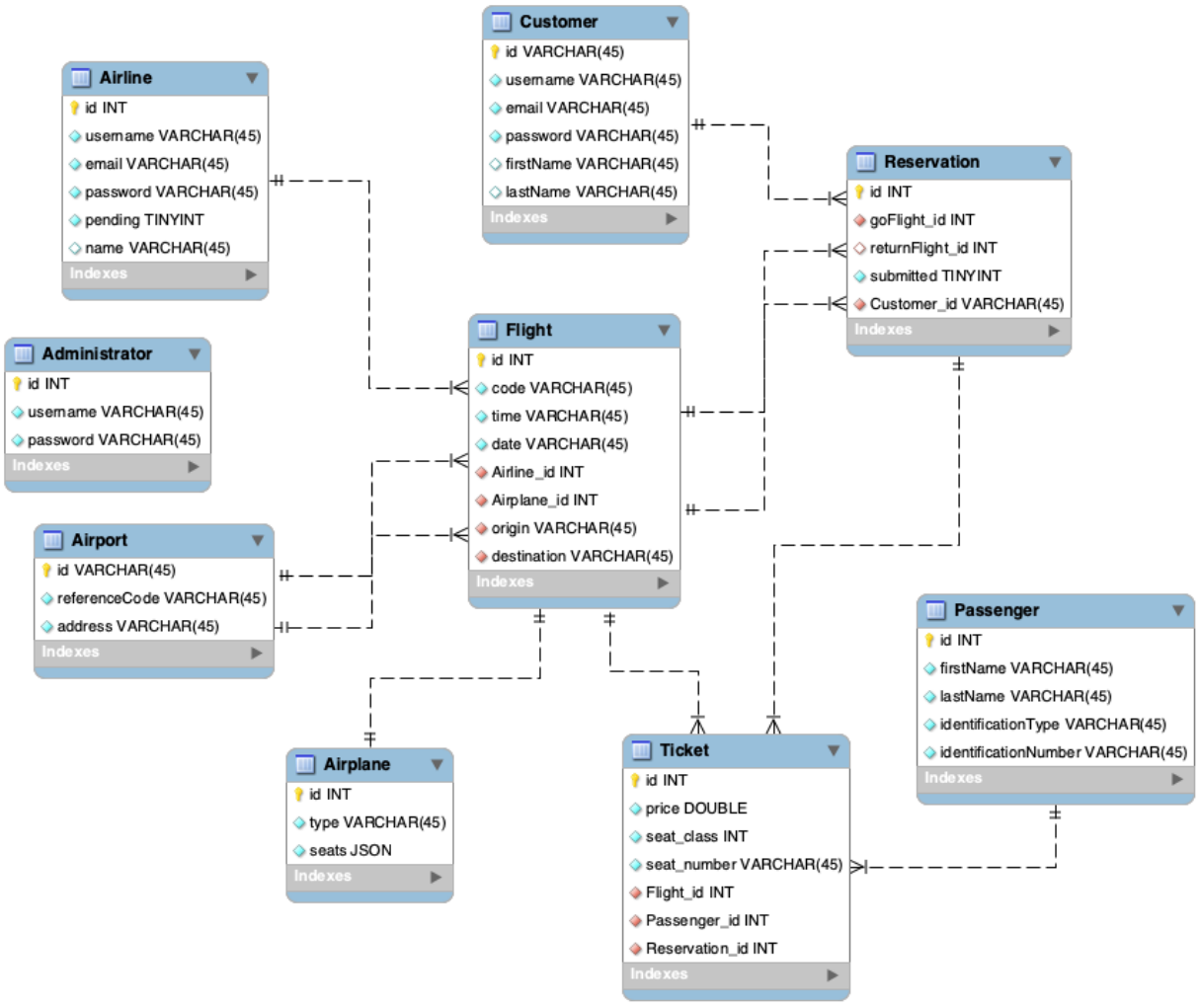


**Figure 9.** Database Table

### 5.2.7 Lazy load pattern

For the TRS we designed, it is unlikely to have all information loaded while the system is running because the amount of data stored in the database is large. So it is necessary for the system to have the

ability to load information only when it is actually needed. With the lazy load pattern, the system is able to do this so that we can have better system performance.

There are four main approaches to implement lazy load and ghost pattern is chosen for this system. The reason for this choice is that it will not increase the complexity of the design model too much compared to the other method. Since TRS does not have heavy memory usage, it is not necessary to use the virtual proxy method to implement the lazy load, not to mention using this method can increase the complexity of the model seriously. The reason why not choosing value holder as our implementation is that it is not ideal to do the generic typecast for the TRS due to the difference among the objects involved. Choosing ghost over lazy initialisation is because that ghost implementation can reduce the number of accessing the database which can result in the reduction of the workload of the server because accessing to the database can be resource consuming sometimes.

Example of use: Airline class has an attribute flights which is an ArrayList and stores all the flights created by the airline. When the administrator views all airlines, the flights list is initialized to null. Only when an airline user views its flights, we load the flights created by that airline.

### 5.2.8 Inheritance Pattern

We implement our TRS system following the Concrete Table Inheritance Pattern. The concrete table inheritance pattern which maps only concrete classes to tables in the database. Each table contains columns for all corresponding fields from the mapped classes, including their superclasses [1].

Example of use: In our system, Customer, Airline and Administrator classes extend the User class. The User class all the objects which need a unique ID, such as Flight, Airport, Ticket inherits from DomainObject class. This pattern makes our mapping from the domain model to tables becomes straightforward and simple. Only the child class has a table in the database. Therefore, we do not need to store plenty of null value, avoiding the confusion or wasted space. In addition, we do not need to perform join operations when querying the database [2].

### 5.2.9 Authentication and Authorization

The authentication pattern specifies a centralized authentication mechanism that encapsulates the detailed information presentation layer of all users who handle all authentication operations. More specifically, the enforcer needs to verify that each request comes from an authenticated entity and ensure that the user is exactly what they say.

We have three types of users which include administrator, airline company and customer in our TRS system. Each of them has their identity verifying choice. For our system's authentication mechanism which consists of two parts. The first part, we assign our users with unique and public statements which are the usernames. For the other part, we use private statement passwords to verify each user's identity to achieve authentication mechanisms. And we use administrator approving the airline login to verify airline's identity, if the airline users are not approved by administrator, they will not be able to login and will get an exception TRSException Waiting for the administrator to approve it. Besides, our deployment on Heroku provides us with HTTPS, In HTTPS, communication is encrypted using Transport Layer Security (TLS) or previously Secure Sockets Layer (SSL). Therefore, it makes sure our channel is secure when we send the password from the front-end to the back-end.

### 5.2.10 Front Controller

We choose to use the Front Controller pattern to simplify the processing of incoming requests. The FrontServlet receives all requests, then based on what the command is, it uses the getCommand methods to get the corresponding command class to process its request. Each class takes care of one type of command, which helps increase cohesion in the design.

The Front Controller pattern is more favorable than the Page Controller pattern in our system because some commands are repeated between pages. By implementing the Front Controller pattern, we can avoid duplicated code when handling requests, and each class would not have to handle different types of requests.

## 6. Physical View

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

### 6.1 Deployment Diagram

Our project is deployed with three parts including the front-end and back-end connected to the Heroku platform. TRS users visit the system via a web browser, the HTTP request is sent to the Servlet. And we use Apache Tomcat to act as both a web server and a servlet container transferring the JSP to Java.

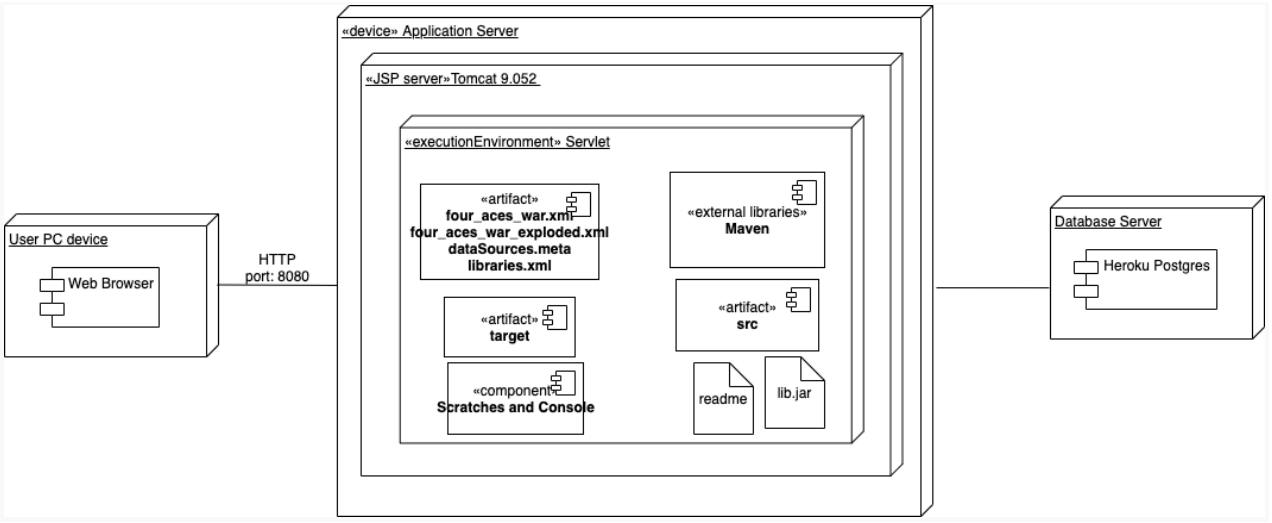


**Figure 10.** Deployment Diagram

## 7.References

*[1] The University of Melbourne, "Software Design and Architecture Subject Notes for SWEN90007", 2021.*
*[2] The University of Melbourne, "Software Design and Architecture Subject Notes for SWEN90007", 2021.*