Product Requirements Document
Submission 1 Specification

2Pizzas

SWEN90007 SM2 2021 Project

In charge:

Ben Nguyen – benn1@student.unimelb.edu.au

Mahardini Rizky Putri – mahardinip@student.unimelb.edu.au

Max Plumley – mplumley@student.unimelb.edu.au

Sothea-Roth Bak – sbak@student.unimelb.edu.au

Revision History

Date	Version	Description	Author
05/08/2021	01.00-D01	Initial draft	Ben Nguyen Mahardini Rizky Putri Max Plumley Sothea-Roth Bak
05/08/2021	01.00-D02	Completed Use Cases in Section 3.2, 3.3, 3.5, 3.6, 3.8, 3.9, 3.13	Ben Nguyen Mahardini Rizky Putri Max Plumley Sothea-Roth Bak
10/08/2021	01.00-D03	Inserted diagram into Section 4.2 Domain Model Diagram	Max Plumley
10/08/2021	01.00-D04	Completed description in Section 4.1 Domain Model Description	Ben Nguyen
10/08/2021	01.00-D05	Completed Use Cases in Section 3.1, 3.4, 3.7, 3.10, 3.11, 3.12, 3.14, 3.15	Ben Nguyen Mahardini Rizky Putri Max Plumley Sothea-Roth Bak
10/08/2021	01.00-D06	Inserted diagram into Section 3.1 Use Case Diagram	Max Plumley
10/08/2021	01.00-D07	Completed Section 1.3 Conventions, terms, and abbreviations	Mahardini Rizky Putri
10/08/2021	01.00-D08	Completed Section 2 Actors	Ben Nguyen Sothea-Roth Bak
10/08/2021	01.00	Review final version of report	Ben Nguyen Mahardini Rizky Putri Max Plumley Sothea-Roth Bak
23/09/2021	02.00-D01	Update use cases	Max Plumley
25/09/2021	02.00-D02	Update domain model diagrams, pattern diagrams	Max Plumley
26/09/2021	02.00-D03	Create design rationale	Max Plumley
26/09/2021	02.00-D04	Update use cases and pattern diagrams	Mahardini Rizky Putri

27/09/2021	02.00	Review final version of report	Ben Nguyen Mahardini Rizky Putri Max Plumley Sothea-Roth Bak

1. Introduction

1.1 Proposal

This document specifies the SWEN90007 project use cases, actors to be implemented, and the system's domain model.

1.2 Target Users

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

1.3 Assumptions, 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.

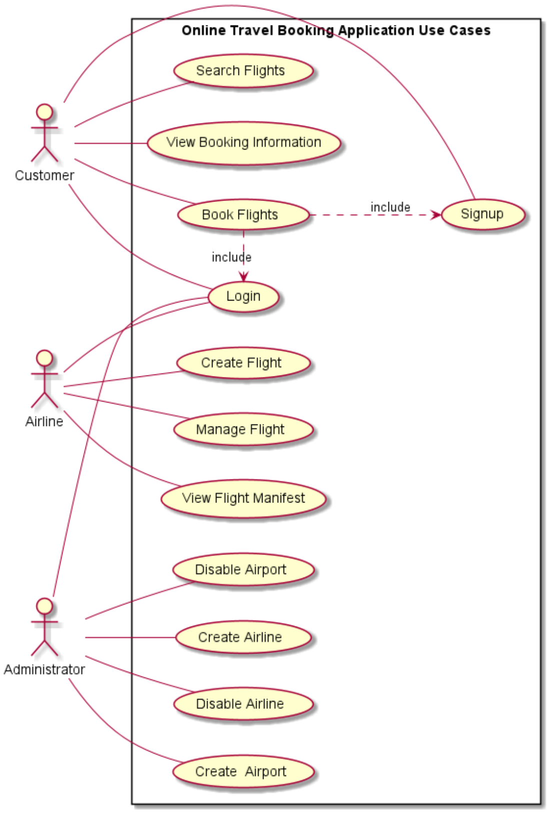
Term	Description
Flight	A flight is a single trip involving a single airline and airplane type that can contain multiple stopover airports between the origin and destination airports.
Stopover	A duration of time spent in a location between the flight departure and arrival.
URL	Uniform Resource Locator
Username	Equivalent to and interchangeable with Email

2. Actors

Actor	Description
Administrator	*A user who is responsible for managing the application, which includes adding airlines and airports to the system.*
Airline	*A user who is responsible for adding, removing, and modifying flights on behalf of an airline.*
Customer	*A user who accesses the application to search and book flights.*

3. Use Cases

3.1 Use Case Diagram



3.2 User Authentication

[UC001] Login

Actors
Administrator, Airline, Customer

Pre-conditions
The user account exists, the user is on the Login page

Main Events Flow
1. User enters username and password
2. The system validates the fields – [AF04]
3. User clicks login button

Commented [EAO1]: Just an example. Students should update and create their own Use Cases.

4. System retrieves user details and verifies with provided login details
5. If the System detects that the provide credentials are invalid – [EF01]
6. System redirects to appropriate dashboard – [AF01] [AF02] [AF03]

Alternative Flows

[AF01] Customer Login

1. If the user is a Customer and the user is completing a booking prior to login then the user is redirected to the Booking Summary page
2. If the user is a Customer and the user is not completing a booking prior to login then the user is redirect to the Home page

[AF02] Airline Login

1. If the user is an Airline, the user is redirected to the Airline Dashboard page

[AF02] Administrator Login

1. If the user is an Administrator, the user is redirected to the Administrator Dashboard page

[AF04] Mandatory Field Not Filled

1. System identifies a record containing at least one mandatory field unfilled and displays a message informing that the field is required

Exception Flow

[EF01] Invalid Credentials

1. System detects that the provided username or password is incorrect
2. System displays message to user explaining that the user could not be authorized as the combination of username and credentials is incorrect

Post-conditions

The user is logged in and authorization token is stored for further interactions with the system

3.3 Airport Management

[UC002] Create Airport

Actors

Administrator

Pre-conditions

The administrator user is authenticated, the user is on the Administrator dashboard.

Main Events Flow

1. User selects Airport page from Administrator Dashboard
2. System presents user with Airport page
3. User selects to create new airport
4. System presents user with form to create new airport
5. User enters new airport details, such as airport name, code, location and time zone ID

6. The system validates the fields – [AF01]
7. User clicks submit button
8. System persists new airport
9. System redirects user back to Administrator Dashboard page

Alternative Flows

[AF01] Mandatory Field Not Filled

1. System identifies a record containing at least one mandatory field unfilled and displays a message informing that the field is required

Exception Flows

Not applicable

Post-conditions

New Airport is added to airport catalogue and available in system for later use

[UC003] Disable Airport

Actors

Administrator

Pre-conditions

The administrator user is authenticated, the user is on the Administrator dashboard. An Airport exists in the system

Main Events Flow

1. User selects Airport page from Administrator Dashboard page
2. System presents user with catalogue of airports
3. User finds desired airport in list of airports
4. User selects disable button for the target airport
5. System sets and persists state of airport in system to INACTIVE
6. System displays airport as INACTIVE in list of airports on Administrator Dashboard page

Alternative Flows

Not applicable

Exception Flows

Not applicable

Post-conditions

The Airport status is set to INACTIVE

3.4 Airline Management

[UC004] Create Airline

Actors

Administrator

Pre-conditions

The administrator user is authenticated, the user is on the Administrator dashboard.

Main Events Flow

1. User selects Airline page from Administrator Dashboard page
2. System presents user with Airline page
3. User selects to add new airline
4. System presents user with form to create new airline
5. User fills out required information to create new airline, such as username, code, name, and password,
6. System validates form fields – [AF01]
7. User selects submit
8. System creates and persists requested Airport in database
9. System redirects user to catalogue of system airports and shows newly created airport

Alternative Flows

[AF01] Mandatory Field Not Filled

1. System identifies a record containing at least one mandatory field unfilled and displays a message informing that the field is required

Exception Flows

Not applicable

Post-conditions

The Airline is created in the system and the airline user is able to log into the system with configured credentials

[UC005] Disable Airline

Actors

Administrator

Pre-conditions

The administrator user is authenticated, the user is on the Administrator dashboard. The Airline exists in the system

Main Events Flow

1. User selects Airline page from Administrator dashboard
2. System presents user with catalogue of system airlines

3. User finds target airline in catalogue
4. User selects button to disable airline
5. System sets and persists state of airline as INACTIVE
6. Airline appears as inactive in airline catalogue

Alternative Flows
Not applicable

Exception Flows
Not applicable

Post-conditions
The Airline account is disabled and attempt to authenticate to the systems as an Airline user will fail

3.5 Flight Management

[UC006] Create Flight

Actors
Airline

Pre-conditions
The airline user is authenticated, the user is on the Airline dashboard

Main Events Flow
1. User selects Flights page from Airline dashboard
2. System presents user with catalogue of system flights for that airline
3. User selects to create a new flight
4. System loads airports and airplane profiles to present user
5. System presents user with form to create a new flight
6. User completes form, selecting origin and destinations airports from loaded airports, selecting an airplane profile from loaded profile, and providing other required fields such as departure and arrival times, code, and costs for each seat class
7. System validates input
8. User submits form
9. System creates and persists flight in database and persists all required flight seats according to selected airplane profile
10. System redirects user to flight catalogue and presents user with newly created flight

Alternative Flows
[AF01] Mandatory Field Not Filled
2. System identifies a record containing at least one mandatory field unfilled and displays a message informing that the field is required

[AF03] Invalid Input Entered

1. System identifies any of the following errors, flight departure occurs after the flight arrival time, any of the stopovers occur outside the time between flight departure and arrival
2. System presents an error message to the user
3. Go to [UC006.5]

Exception Flows

Not applicable

Post-conditions

The new flight is created and persisted in the system with no current seat allocations and with a state of ON_SCHEDULE, flight appears in appropriate search results.

[UC007] Manage Flight

Actors

Airline

Pre-conditions

The airline user is authenticated, the user is on the Airline dashboard. The flight exists in the system

Main Events Flow

1. User selects Flights page from Airline dashboard
2. System presents user with catalogue of system flights for that airline
3. User finds target flight in flight catalogue
4. User selects view flight button for target flight
5. System presents the user with the flight view page and status change buttons [AF01] [AF02] [AF03]
6. User selects edit button
7. System presents the use with the flight edit page with form to edit flight details
8. [AF04]
9. System redirects user to flight catalogue and presents user with updated flight

Alternative Flows

[AF01] Cancel Flight

1. User selects the cancel flight button
2. System sets state of flight to CANCELLED and persists the new flight state

[AF02] Delay Flight

1. User selects the mark as delayed button
2. System sets state of flight to DELAYED and persists the new flight state

[AF03] Schedule Flight

1. User selects the mark as to schedule button
2. System sets state of flight to TO_SCHEDULE and persists the new flight state

[AF04] Edit Flight Details

1. User alters fields on flight edit form, optionally setting a new departure time, arrival time and stopovers.
2. System validates form input – [AF05]
3. User selects submit
4. System sets new fight details for flight and persists updated flight

[AF05] Invalid Input Entered

1. System identifies any of the following errors, new departure occurs after the flight arrival time, any of the existing or new stopovers occur outside the time between flight departure and arrival
2. System presents an error message to the user
3. Go to [UC007.8]

Exception Flows

Not applicable

Post-conditions

New flight state and details are persisted in the system

[UC008] View Flight Manifest

Actors

Airline

Pre-conditions

The airline user is authenticated, the user is on the Airline dashboard. The flight exists in the system

Main Events Flow

1. User selects Flights page from Airline dashboard
2. System presents user with catalogue of system flights for that airline
3. User finds target flight in flight catalogue
4. User selects view flight for target flight
5. System presents flight view page with list of passenger details (given name, surname, passport number, date of birth, name of booked seat, class of booked seat, nationality) for that flight

Alternative Flows

Not applicable

Exception Flows

Not applicable

Post-conditions

Not applicable

3.6 Booking Management

[UC009] Search Flights

Actors
Customer

Pre-conditions
The user is on the home page.

Main Events Flow
1. User selects origin and destination airports, departure date and number of passengers
2. User optionally selects to search for return flights – [AF01]
3. User submits search criteria
4. System searches flight catalogue for flights that satisfy the flight criteria
5. System presents user with list flights in search results, system filters out all flights with CANCELLED status, all flights that have already departed, and all flights with no seat availabilities

Alternative Flows
[AF01] Search with Return Flight Results
1. User selects a return date to search for return flights
2. User submits search criteria
3. System searches flight catalogue for flights that satisfy the outbound flight criteria and a second search that satisfies the return flight criteria
4. System directs user to flight search results page
5. System presents user with list of outbound flights and seconds lists of return flights in the search results, system filters out all flights with CANCELLED status, all flights that have already departed, and all flights with no seat availabilities

Exception Flows
Not applicable

Post-conditions
Flight search results is stored in session to be used later for booking flow

[UC010] Book Flights

Actors
Customer

Pre-conditions
The user has completed a flight search according to [UC009], user is on the Flight Search Results page

Main Events Flow

1. System presents user with list of flight results from original search
2. User selects desired flight from list of flights in flight search results
3. System adds flight to booking request
4. If the original search includes return flights – [AF02]
5. User selects book button
6. If user is not logged – [AF03]
7. System presents user with Booking Details page with a passenger form for each passenger
8. User enters passenger details for each passenger (given name, surname, passport number, date of birth, name of booked seat, nationality) – [AF01]
9. User assigns an available seat to each passenger
10. System adds seat allocations to the booking request
11. If the original search includes return flights - [AF04]
12. User selects complete booking button
13. System finalizes booking request and submits request
14. System checks the availability of requested seats and creates a new booking in system for flight, seats and user – [EF01]
15. System redirects user to Customer Booking page and presents new booking in list of bookings for customer

<u>**Alternative Flows**</u>

[AF01] Mandatory field not filled

1. System identifies a record containing at least one mandatory field unfilled and displays a message informing that the field is required

[AF02] Choose Booking Return Flight

1. User selects desired flight from the list of return flights
2. System adds flight to booking request

[AF03] Customer Authentication

1. System presents login form to user
    a. If user has a Customer account include [UC001]
    b. If user does not have a Customer account include [UC011]
2. System redirects user to booking details page to complete booking flow. Go to [UC010.6]

[AF04] Choose Return Flight Seats

1. User allocates seat on return flight to each passenger
2. System adds seat allocations to booking request

<u>**Exception Flows**</u>

[EF01] Flight Seats are Already Booked

1. System presents a message to the user informing them that the requested booking could not be completed because one or more seats requested in the booking are no longer available
2. System performs original search showing updated availabilities for flights
3. System redirects user to Flight Search Results page and presents new search results

Post-conditions
Booking is created in the system and appears in the logged in users Customer Bookings page

[UC011] Signup

Actors
Customer

Pre-conditions
User is on the login page

Main Events Flow
1. System presents user with login and signup buttons
2. User selects signup button
3. System presents user with Customer Signup page with signup form
4. User fills out required information in form (given name, surname, username, email and password) – [AF01]
5. User selects signup button – [AF02]

Alternative Flows
[AF01] Mandatory field not filled
4. System identifies a record containing at least one mandatory field unfilled and displays a message informing that the field is required

[AF02] Username not Available
1. System detects that the requested username is already assigned to a user
2. System displays and error to the user requesting that the user provide an alternate username, Go to [UC011.4]

Exception Flows
Not applicable

Post-conditions
The Customer user account is created and the user is able to authenticate as a Customer to the system with the provided credentials

[UC012] View Booking Information

Actors
Customer

Pre-conditions
The Customer user is authenticated and on the Customer Dashboard page

Main Events Flow

6. User selects my bookings
7. System presents user with the Customer Bookings page with a list of bookings
8. User selects view button on a booking
9. System presents user with detailed information about booking such as flights booked and passengers and seats associated with the booking

Alternative Flows

Not applicable

Exception Flows

Not applicable

Post-conditions

Not applicable

4. Domain Model

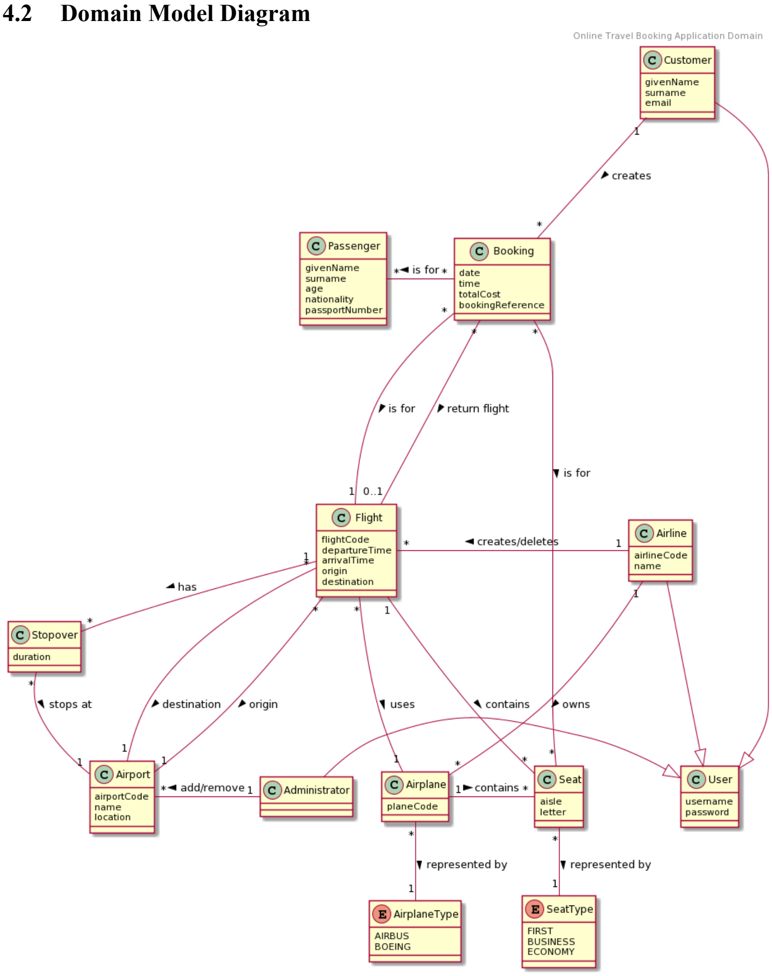
4.1 Domain Model Description

Based on the specifications provided for the Online Travel Reservation System, the system entities, attributes, and business rules can be summarised as:

- **Users** can be either an **Administrator, Airline, or Customer**;
- Only **Administrators** can create **Airports**;
- Only **Airlines** can create, modify and remove **Flights**;
- Only **Customers** can search for and book **Flights**;
- **Seats** *are either* in **First**, **Business**, or **Economy** class;
- **Planes** *are either* **Airbus** or **Boeing** type;
- **Customers** must have an email;
- **Passengers** must have a passport number;
- **Passengers** can have *one or more* **Bookings;**
- **Bookings** must have a bookingReference;
- **Bookings** are for *one or more* **Passengers;**
- **Booking** are for *one or more* **Seats;**
- **Bookings** have *one* **Flight;**
- **Bookings** may have *one* return **Flight;**
- **Flights** may have *one or more* **Stopovers;**

Entities have been bolded; attributes have been underlined; and important *associations* have been italicized.

4.2 Domain Model Diagram



5. Solution Model

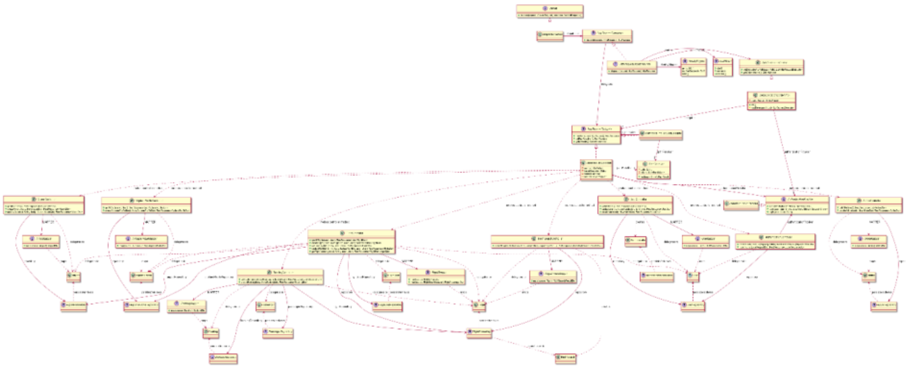
Because the 2Pizza system is quite large and complex the three most important layers Application, Domain, and Data have been documented separately.

5.1 Application Layer

5.1.1 Class Diagram Description

The Application Layer aggregates a number of controllers that delegate to the Domain Layer. This layer is primarily responsible for accepting, validating and transforming HTTP requests into a format that can be consumed by the Domain Layer, additionally this layer transforms responses from the Domain Layer into an appropriate HTTP response to a waiting client.

5.1.2 Class Diagram

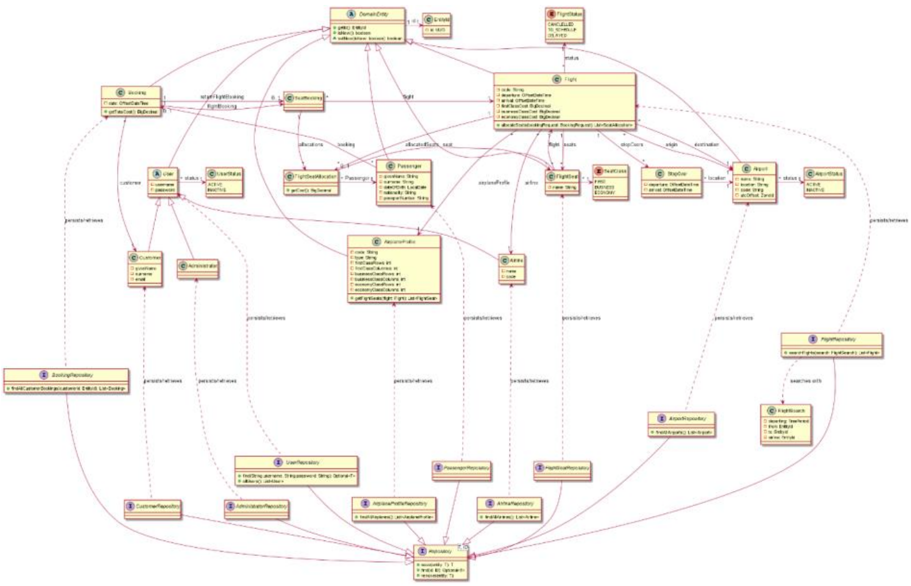


5.2 Domain Layer

5.2.1 Class Diagram Description

The Domain Layer aggregates all the domain classes, which store and operate on business data. All business logic is implemented in this layer. The Domain Layer also declares a number of Repository interfaces to be implemented in the Data Layer.

5.2.2 Class Diagram



5.3 Data Layer

5.3.1 Class Diagram Description

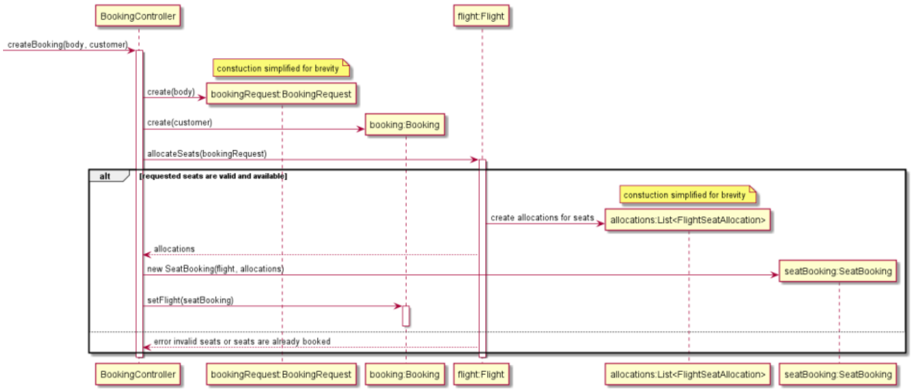
The Data Layer aggregates all the Data Mapper classes as well as implementations for Domain Layer Repository classes.

6. Patterns Used

6.1 Domain Model

To implement the business logic concerning such actions as creating, managing and booking flight the team made use of the Domain Model to build an object structure that co-locates business data with code that operates on such data. Various classes such as Flight, FlightSeat, User and Booking were identified as entities within the system and attributed appropriate responsibilities for performing business actions within the domain, for example Flights aggregate FlightSeat entities and were identified as the best class to manage allocating FlightSeat entities to a booking, thus a method allocateSeats(BookingRequest request) is implemented within the Flight class.

The following diagram shows the interactions within the Domain Model implementation to allocate seats from a flight to a booking, note that the Booking controller delegates creating the SeatAllocation classes to the Flight class itself – which encapsulates the allocation business logic – before setting these allocations on the booking.

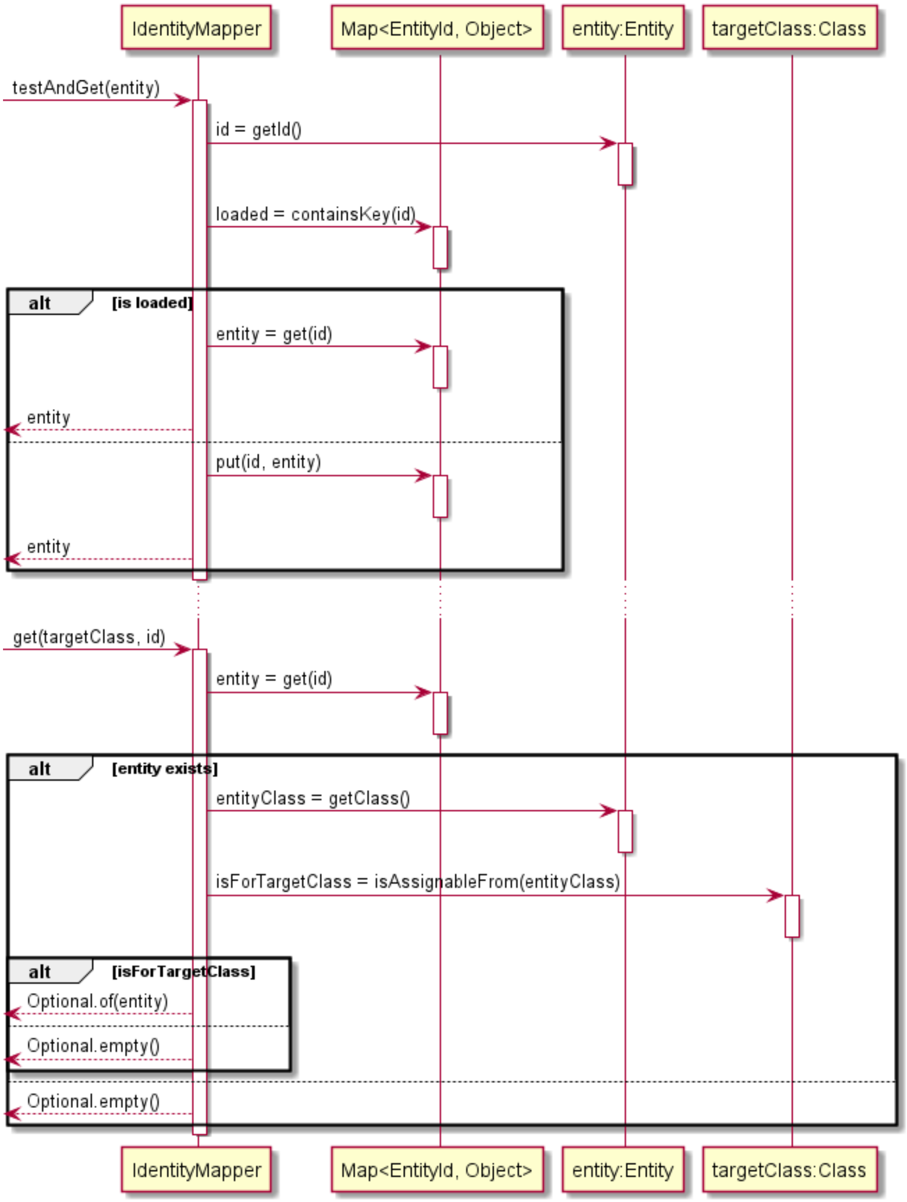


6.2 Identity Map

Because the team opted to use UUIDs for implementing the Identity Field pattern, entity identifiers are guaranteed to be unique across the system, this means that one generic Identity Mapper could be implemented to manage all entities in the system.

The IdentityMapper interface exposes several methods to store and retrieve entities retrieved from the database. The get method returns an entity of the required type and identifier if it exists, while the testAndGet method registers a newly retrieved entity with mapper and returns either that entity if it has not yet been retrieved or a prior retrieved instance if one is already registered with the mapper. A reset method is exposed so that the mapper (which is instantiated for each thread) can be cleared between requests.

The diagram below shows the flow taken for retrieving an entity by id from the map and also a test and get method for both registering a new entity and returning either that entity if it has not been loaded or a prior registered instance.

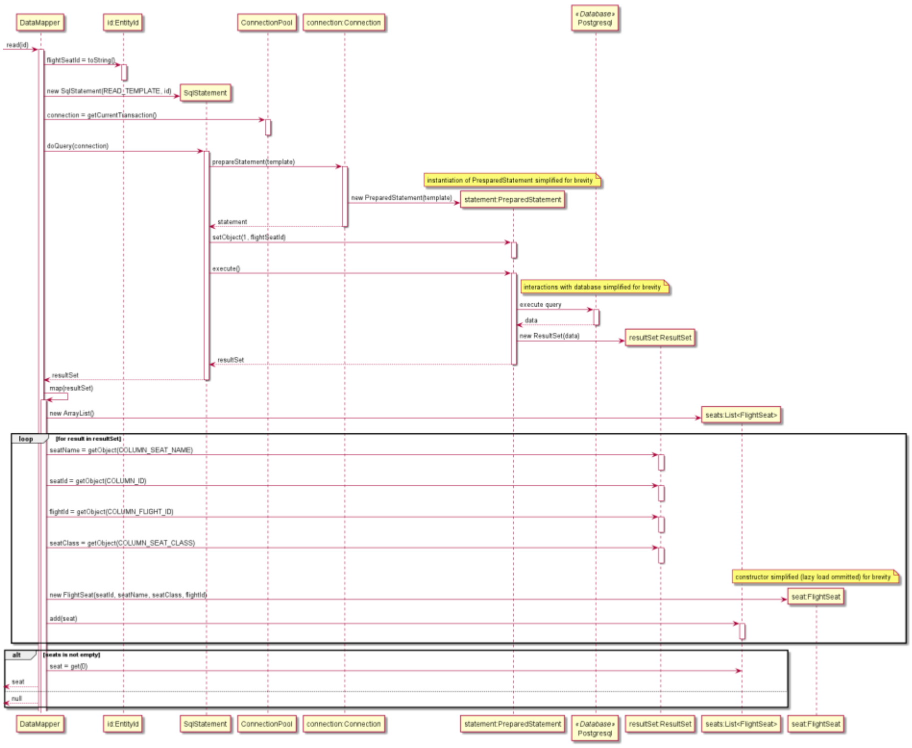


6.3 Data Mapper

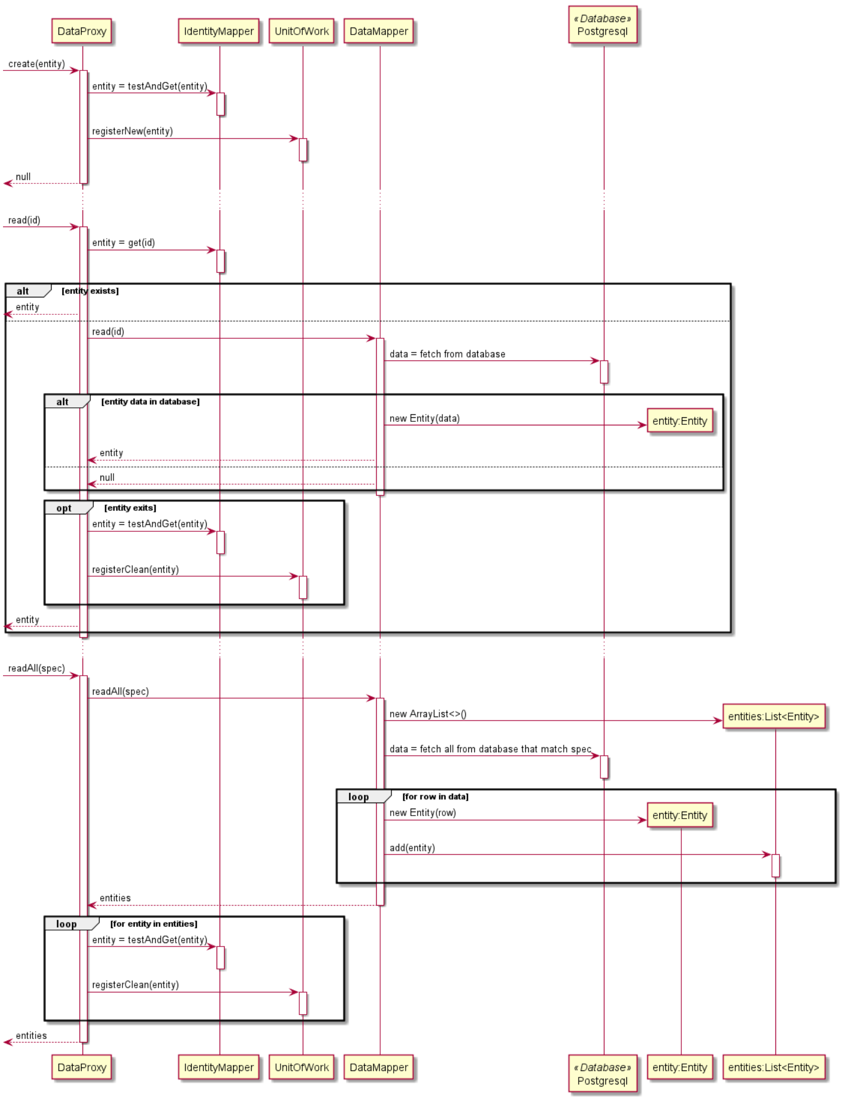
The Data Mapper pattern was implemented for each entity in the domain - such as Flight, FlightSeats, Passenger etc. A Separated Interface was used to define a generic interface to be used by client code in data layer and implemented elsewhere for each entity that requires persistence. This Separated Interface exposes the basic CRUD operations as well as a findAll() method that accepts a generic Specification object for finding entities that satisfy a criteria.

The DataMapper classes are required to interact with both the UnitOfWork and IdentityMapper classes to ensure that actions against the database are executed in the context of a single transaction and additionally to guard against instantiating multiple instances of the same entity while retrieving entities from the database. Rather than relying on correct and consistent implementation of these interactions in each entity specific mapper the team opted to wrap mappers injected as dependencies to the domain layer with a DataProxy class that intercepts calls to create, read, update and delete entities and instead delegates as appropriate to the UnitOfWork or IdentityMapper implementations. Note that the UnitOfWork received un-proxied references to each mapper (via the DataMapperRegistry) so that when it is required commit changes to the database calls to CRUD methods on mappers are not intercepted.

The diagram below shows the flow taken by a typical Data Mapper implementation (in this case the FlightSeatMapper) for the read method, similar interactions exist for the other CRUD operations and are omitted for brevity



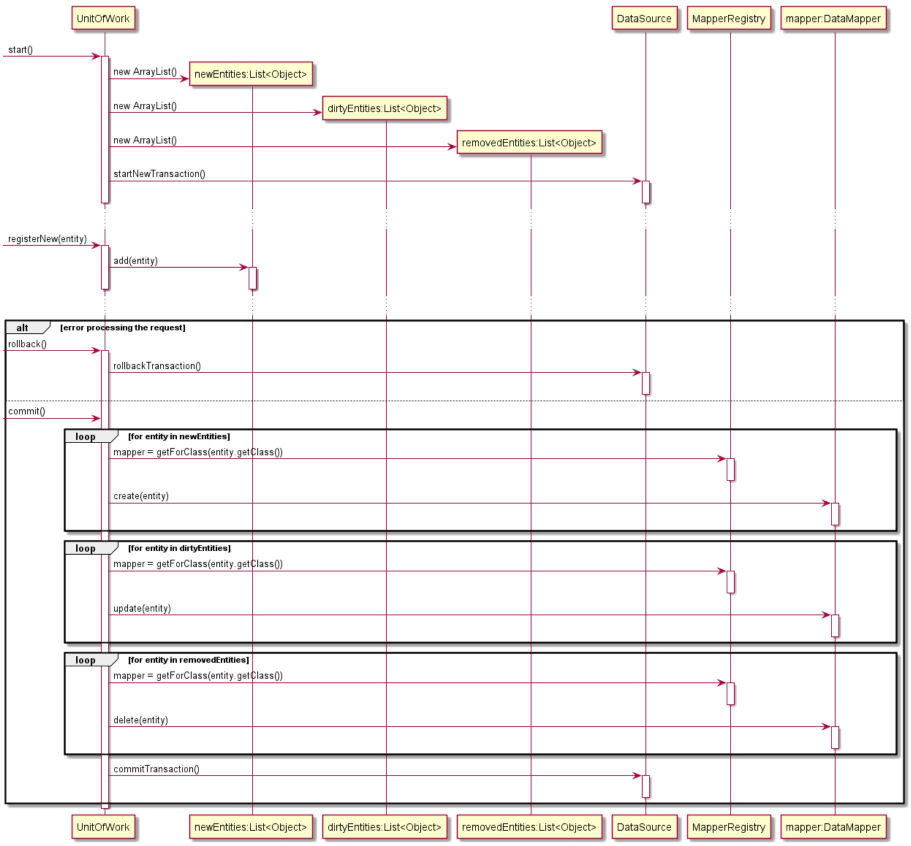
The diagram below shows the flow taken by the DataProxy class when invoked with a subset of the DataMapper CRUD methods. The primary responsibility of the DataProxy class is to intercept requests from the domain layer that manipulate database resources. The nature of this interception behaviour is relatively consistent for create, update and delete invocations, for brevity only create is show in the diagram. Read requests are intercepted if the entity has not been loaded prior. Request to read all entities according to a Specification are executed without interception however the results are registered with the IdentityMapper and newly loaded entities are replaced with prior loaded instances where required.



6.4 Unit of Work

Implementation of the Unit of Work pattern was straight forward, the interface exposes hooks used by a Front Controller to start and commit a transaction – or optionally rollback the transaction should there be an error while processing the request. Additionally, the UnitOfWork provides methods to register entities for creation, update or deletion in the database, to be invoked primarily by DataMapper proxies. The implementation makes use of a DataMapperRegistry that has access to un-proxied versions of each mapper in the domain (the rest of the application receives proxied mappers that are intercepted by the IdentityMapper and UnitOfWork implementations).

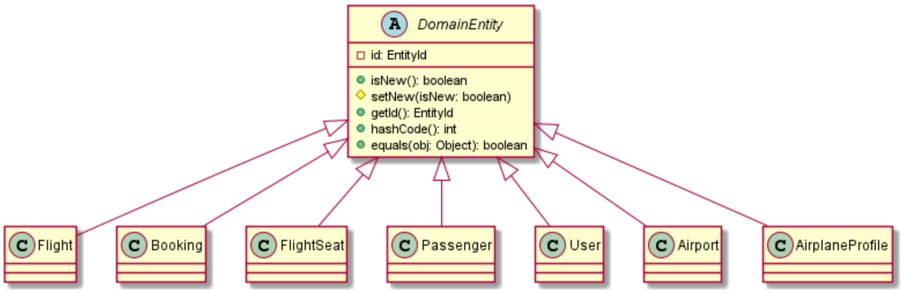
The diagram below shows the sequence of calls made to the UnitOfWork while servicing a request, for brevity the diagram shows a single call to register a new entity, the registration of dirty and removed entities is very similar.



6.5 Layer Supertype

The Layer Supertype pattern collects common features of objects within layer, for example the domain layer, into a single superclass from which all other objects in the layer inherit, and in the process helps to avoid duplicating common code across the objects of the layer.

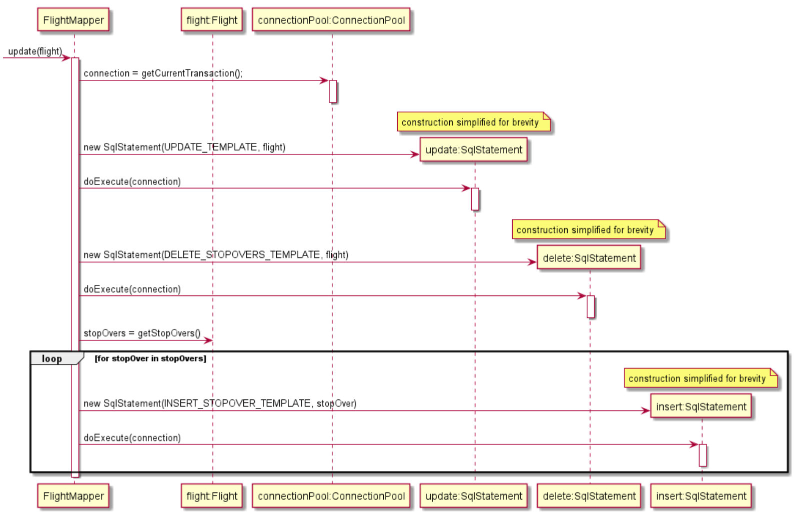
The DomainEntity and AbstractRepository classes were created in accordance with the Layer Supertype pattern. The DomainEntity supertype implements common code concerning entity identity in the application and the isNew feature that is common to all entities in the 2Pizza system.



6.6 Dependent Mapping

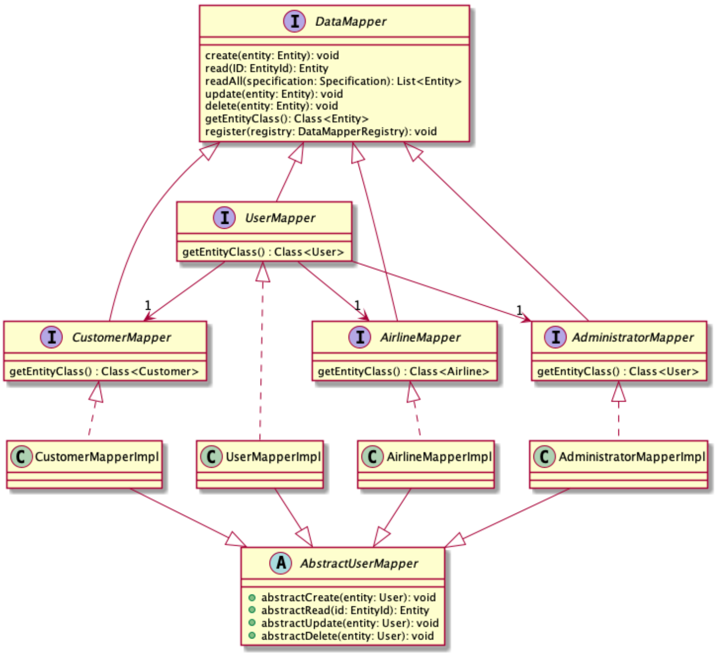
The Dependent Mapping pattern was used to implement persistence for Stopover value objects associated with a particular Flight entity. Stopovers were a suitable candidate for Dependent Mapping pattern as they are a true composite of their singular owning Flight entity, Stopovers are only ever accessed via their owning Flight entity and so the FlightMapper was delegated the responsibility of managing rows in the 'stopover' table with respect to a particular Flight entity. When persisting a Flight, the FlightMapper class also adds the appropriate rows to the 'stopover' table; conversely when deleting a Flight, the FlightMapper class selects and deletes all associated stopovers. The team settled for the simple approach of removing and reinserting all stopover rows on an update of a Flight.

The following diagram shows the flow to persist an update of a Flight and associated StopOver classes.



6.7 Inheritance Mapper

The inheritance mapper was implemented to map Customer, Admin and Airline objects to tables in the database. Because these 3 classes inherit from the User superclass, we must use the inheritance mapper pattern in order to map these objects to the corresponding tables in the database. Our implementation is represented as following:



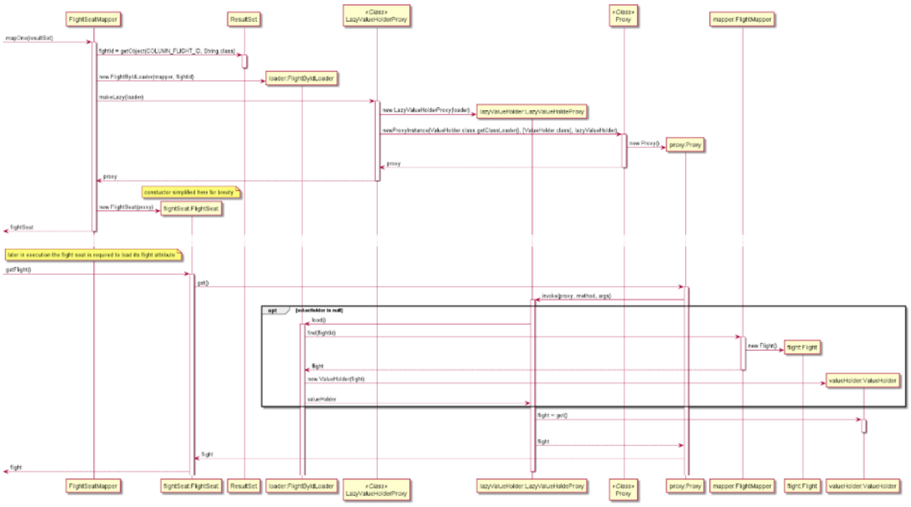
When an Admin, Airline, or Customer object is instantiated, the UserMapper create method is invoked, which delegates the creation of the object to the relevant mapper based on what type of user is being created. For example, if a Customer is created, the UserMapper will call the CustomerMapper's create method. The CustomerMapper inherits from the AbstractUserMapper, and uses the create method of the AbstractUserMapper to create the User in the User table. It will then create the corresponding Customer object in the Customer table.

6.8 Lazy Load

Lazy Load was implemented via the Value Holder variant of the pattern. Cyclical relationships between entities, such as between Flight and FlightSeat classes were managed by lazily loading entities to break loading cycles. Where lazy loading was required an instance of LazyValueHolderProxy class created instead of a ValueHolder class. Because the team made use of the Domain Model pattern much of the awkwardness of passing ValueHolder classes around could be abstracted from client code in each entity.

The ValueHolder class is a simple generic container with one primary method get() which returns the value held by the ValueHolder. When implementing eager loading the developer has the option of immediately loading the required value and encapsulating it within a BaseValueHolder. If lazy loading of a value is required, the developer instead instantiates a LazyValueHolderProxy and with a ValueLoader class that defers loading and is able to load the required value only if an when requested.

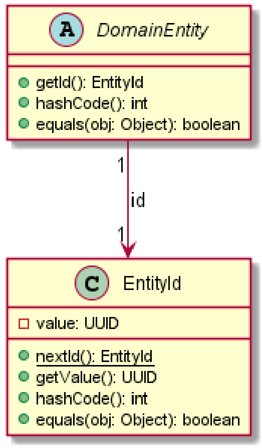
The diagram shows the flow taken to instantiate a LazyValueHolderProxy for the Flight field of a FlightSeat class and the subsequent flow taken if and when the wrapped value holder value is required in the course of application execution.



6.9 Identity Field

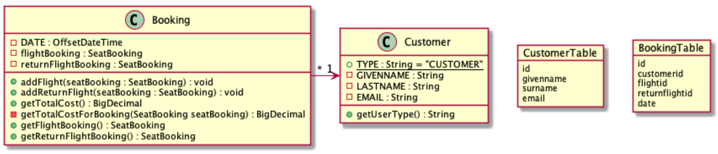
The team decided to use V4 UUIDs to identify, persist and retrieve entities in the system, the rationale for this was largely due to the ability to simplify Identity Mapper implementations by guaranteeing that entity identifiers are unique across the whole system and thus across all tables in the database. To ensure consistency across the application the UUID java class was encapsulated in a custom EntityId class to be used as an Identity Field for entities.

The class diagram below shows both DomainEntity and EntityId classes. Instances of DomainEntity classes keep a reference to and are identified by an instance of an EntityId class. The static nextId() method returns a new unique EntityId. Both classes have overridden hashCode() and equals(Object) methods, the overridden implementations for EntityId use the encapsulated UUID while the implementations for DomainEntity delegate equality to the encapsulated EntityId.



6.10 Foreign Key Mapping

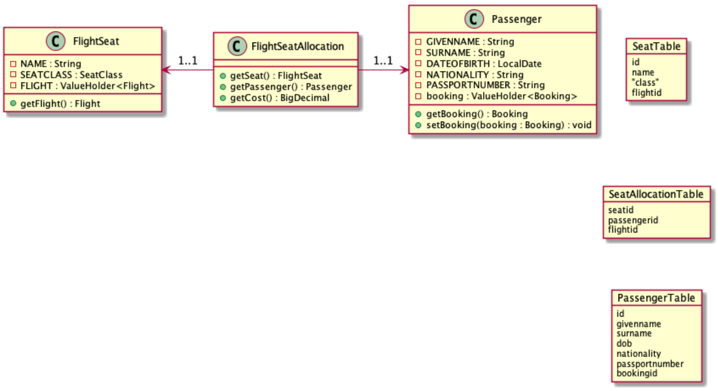
Foreign key mapping addresses the issue of representing object references in the database. Specifically, when an object has a relationship with another, it stores a reference to it. The foreign key mapping pattern addresses this issue by mapping the object reference to a foreign key in the database. We used this pattern to map relationships such as Customer – Booking to the database. The representation of the relationship as objects and in the database is as follows:



The customer object stores the booking object as a reference, but in the database, the booking table stores the foreign key to the customer instance that it is associated with.

6.11 Association Table Mapping

A many-to-many relationship is simple to represent in objects – an object will store a list of references to the objects it has a relationship with, and vice versa. However, in relational databases we must create an association table to store the relationship between the two tables. The association table stores the foreign keys from both tables, to represent the relationship. The team implemented a SeatAllocation association table to store the many-to-many relationship of seats and passengers.



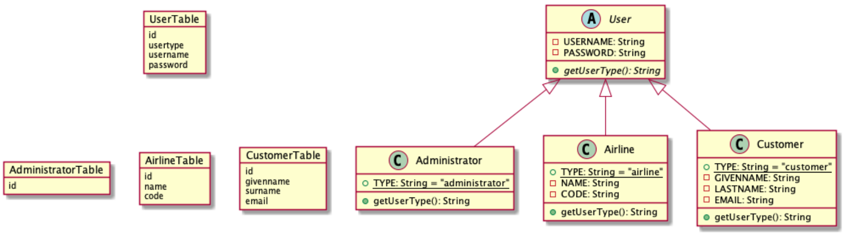
6.12 Embedded Value

Embedded value is a special case of Dependent Mapping class where the value is a single dependent object. No value objects in the 2Pizzas system (Stopover and SeatAllocation classes) have an appropriate multiplicity that would suit the implementation of the Embedded Value pattern. See section of Dependent Mapping for discussion around implementing persistence of value objects with 0..* multiplicity.

6.13 Class Table Inheritance

A downside of using a relational database is that it does not support inheritance, which has been implemented with our User superclass and Admin, Airline, and Customer subclasses. To model this relationship in our database, we must use the Class Table Inheritance pattern.

The entities in our domain and their inheritance structure, and the related Class Table Inheritance pattern used can be represented as following:

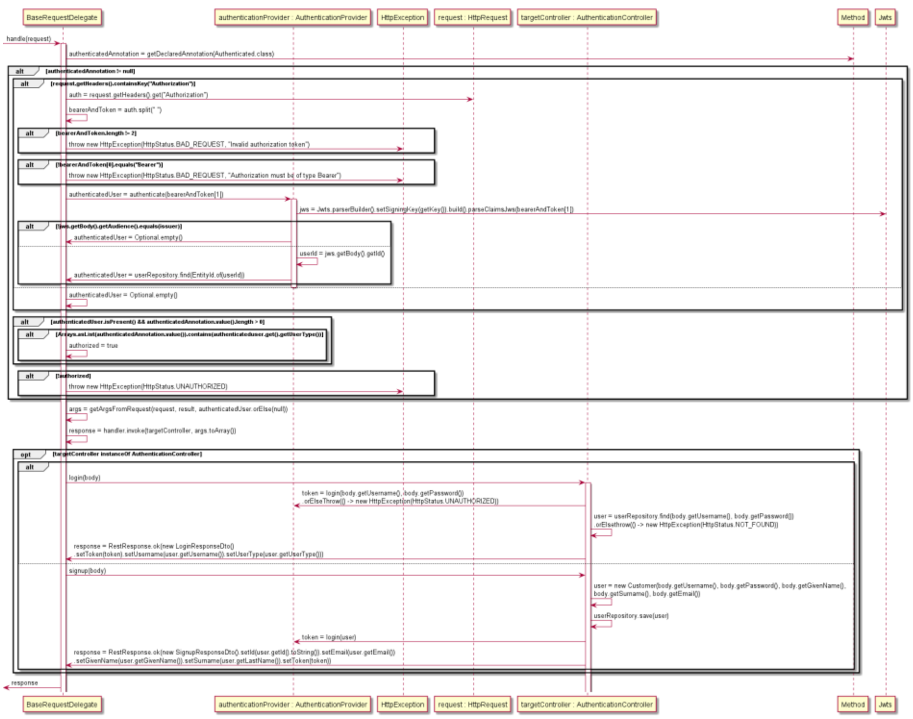


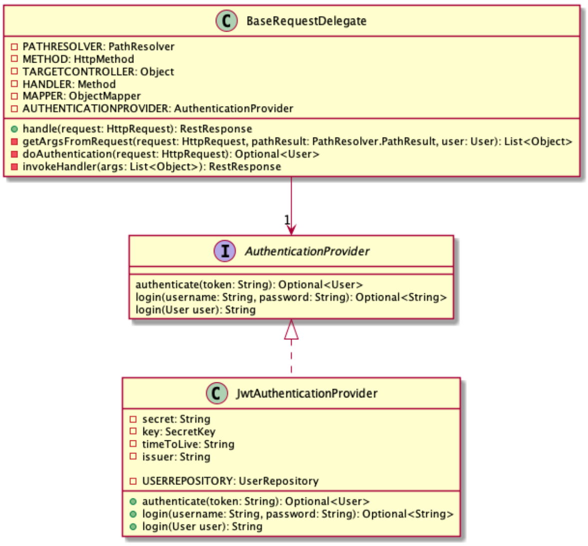
In the Class Table Inheritance pattern, each class has a corresponding table in the database which stores the associated properties with that subclass.

Our implementation consisted of an abstract User superclass, which stored the username and password properties, which were inherited by the Admin, Airline and Customer classes. The User table in the database also had these fields, alongside a primary key (id), which is used as a foreign key in the Admin, Airline and Customer tables, to join with the User table and fetch the username and password of the instances of the concrete classes.

6.14 Authentication with JSON Web Token Authentication Provider

The team used JSON Web Token technologies to handle authorization of users within the 2Pizzas system. JWTs are convenient as they don’t require management of an authenticated session state for each user. Our authentication solutions required generating and securely signing a token on login of a user and then providing this token as a header in each subsequent request. To properly encapsulate operations against a token the team implemented against an AuthenticationProvider interface with authenticate() and login() methods. The login method is used by the AuthenticationController when servicing user requests to login, this method searches the database for a user with matching username and password (stored salted and hashed at rest) and if such a user exists generates the requested token. When receiving requests that require authentication the Front Controller handles extracting the authentication token from the request headers and passes it to the AuthenticationProvider to verify; if the token is verified and a matching user is discovered in the database then the Front Controller verifies that the user’s roles align with the permitted roles associated with the requested resource. If the token is invalid or the user roles are not sufficient to access the resource, then the request is rejected, and the Front Controller replies with an unauthorized message. The Diagram bellow show the comleplete authentication, signup and login flow.

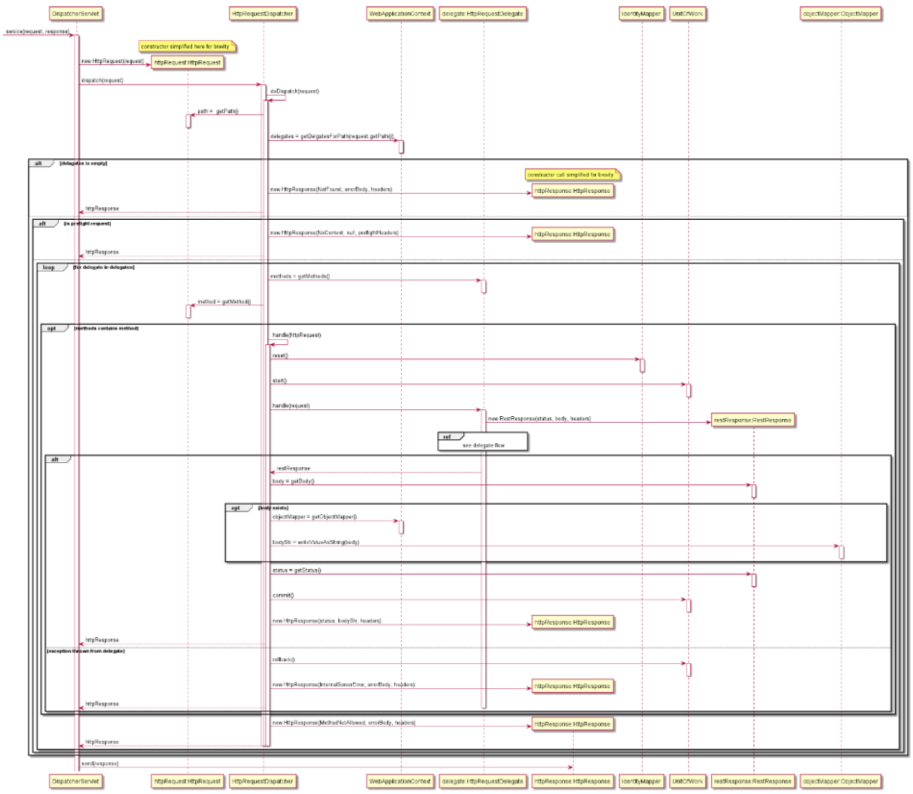




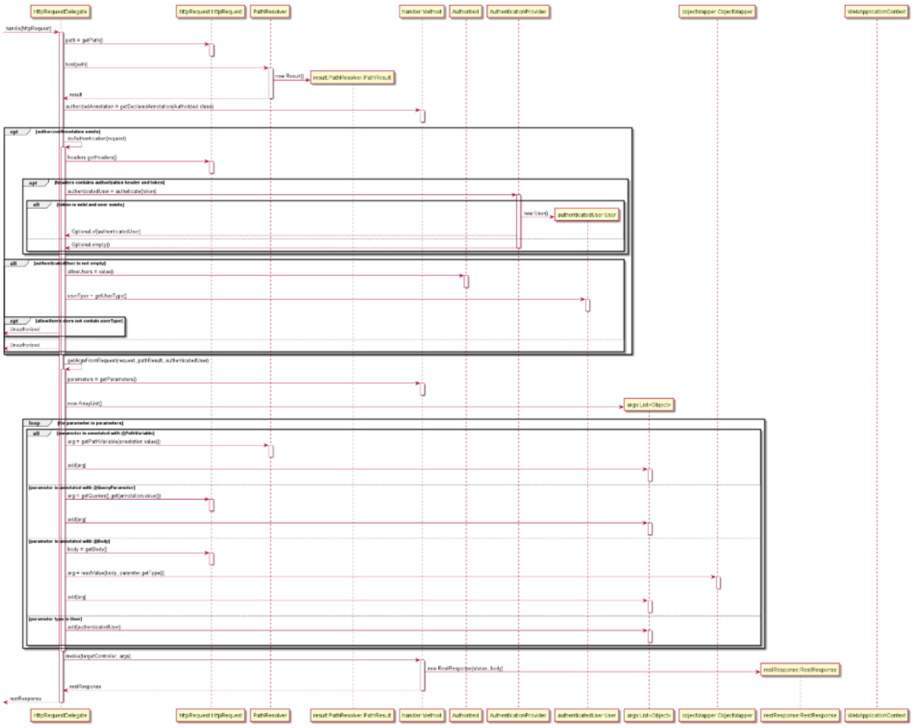
6.15 Front Controller

The team implemented a front controller to handle all requests to the application. The benefit of a front controller is that it provides a single point at which to perform generic tasks for each request, such as marshalling and unmarshalling JSON payloads, interpreting HTTP query parameters, handling error responses, authentication, CORS and initializing/commit/rollback of Unit of Work. The system makes uses of a HttpRequestDispatcher class that performs the previously mentioned tasks and delegates valid requests to the appropriate domain controller. The system makes use of Dependency Injection pattern to discover and instantiate controller classes at runtime.

The following diagram shows the flow taken by the application when it receives a servlet request. Servlet request are received by a single servlet listening on the root path '/', these requests are marshalled into a custom HttpRequest class and passed to the HttpRequestDispatcher that uses a PathResolver class to direct the request to an appropriate HttpRequestDelegate class.



HttpRequestDelegate flow is referenced in the above diagram and is detailed below

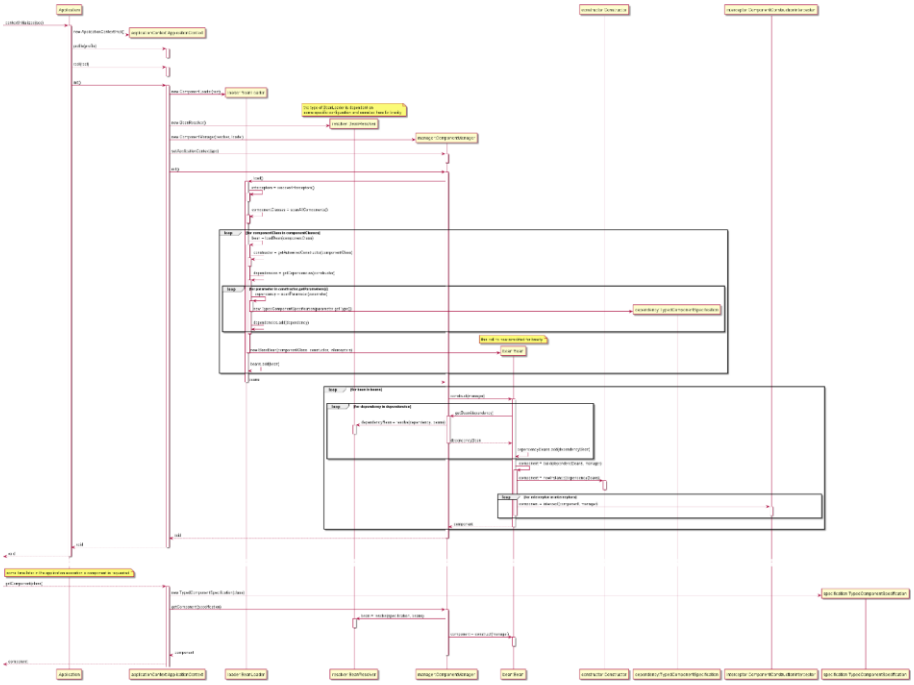


6.16 Inversion of Control with Dependency Injection

Inversion of Control pattern allows generic control code to be implemented by a framework which calls into specially implemented objects that encapsulate the specific behaviour of the domain. This process allows the wrapping framework to be decoupled from the specifics of a software implementation and thus be reused across multiple domains. Our project uses Inversion of Control to implement a generic framework for handling HTTP requests, authentication, and database interactions, providing specific points for domain specific behaviour to be plugged in.

Our solution for Inversion of Control makes use of Dependency Injection to transparently load domain specific implementations where required by the framework. For example, implementations of HTTP controllers are discovered at run time and loaded into a HttpRequestDispatcher object. Managing the dependencies of these controllers (some of which may require access to repositories or other domain services) becomes the responsibility of the dependency injection framework. This reattribution of responsibility is one of the key benefits of Dependency Injection as it decouples client code from the process of instantiating or discovering the service code on which it depends, and thus allows for a single point at which to manage dependencies across an application and enables an application to easily switch out implementations at runtime. Dependency Injection is often described as an alternative to the Static Singleton pattern, which provides global access to shared services but should generally be avoided as it requires that service implementations be statically decided at compile time. Because Dependency Injection defers binding of dependencies until run time it's possible to configure an application to run client code against Service Stubs for testing and then configure that same code to run against real implementations when executing in a production environment. The team has written various unit tests for critical parts of the application including the various implementations of Data Mappers required by the domain. These tests utilize Service Stubs generated with the Mockito framework to efficiently exercise branching client code, and rely on Dependency Injection to inject real server code implementation when executed in production

The diagram below shows the flow the application takes to instantiate and initialize an ApplicationContext for the purposes of Dependency Injection. At start up the ComponentLoader class uses reflection to discover classes annotated with an @Component annotation. The presence of the @Component annotation on a class declaration marks it as a class that should be managed by the ApplicationContext rather than instantiated directly by client code. The framework identifies the dependencies for each component by inspecting the parameters of a constructor annotated with @Autowired. At start up the ComponentManager loads all required components and instantiates each required component, recursively instantiating service components on which client components depend. Components are by default instantiated as Singletons within the ApplicationContext but can optionally be marked with an @ThreadLocalComponent annotation and instead instantiated with a ThreadLocal context, which is particularly important for the implementations of both Unit of Work and Identity Map patterns.

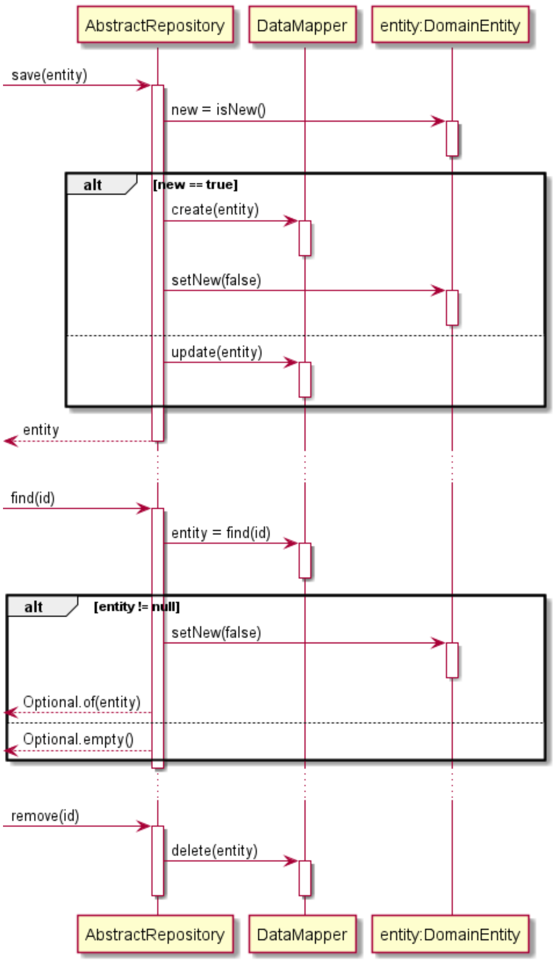


6.17 Repository

The Repository pattern is a mediator between domain logic and Data Mappers in the data layer. The Repository pattern provides a collection-like interface to manage persistence of domain objects, CRUD operations become simply save() find() and remove(). To implement the save operation the application needed a mechanism to decide if an object is ‘new’ and requires a call to create() on an appropriate Data

Mapper class or is not 'new' and should be updated with update(), this functionality was implemented in DomainEntity Layer Supertype and is further discussed in the Layer Supertype section of this report.

The diagram below shows the flow for AbstractRepository from which implementations specific to each domain entity (ie Flight, Booking, User etc) inherit, these specific implementation are rather light weight and typically only extend the abstract functionality by constructing appropriate Specification objects as input to a findAll() invocation on a wrapped Data Mapper class.

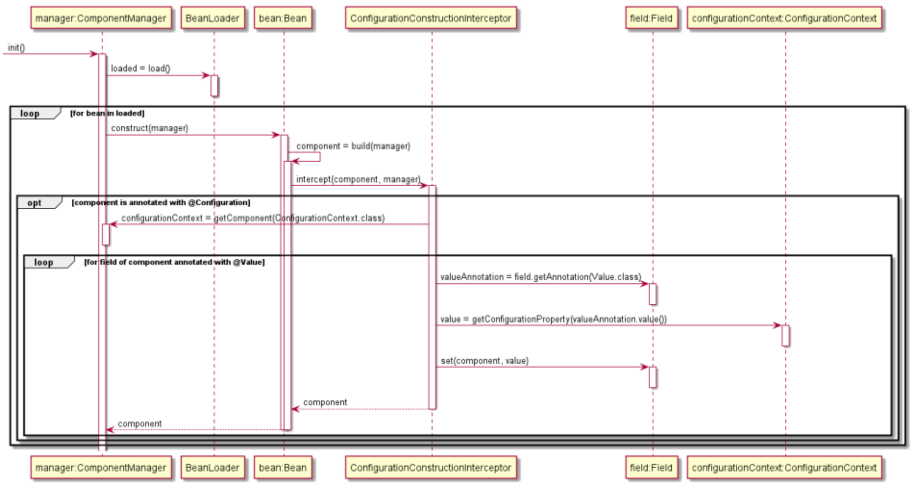


6.18 Externalized Configuration

It is often required that an application be deployed to multiple environments, the application may be deployed to a testing environment during the execution of CICD pipeline, and then later deployed to a set of further environments for validation prior to being promoted to a production environment. When deploying an application to multiple environments it is often necessary to consider the different behaviours that each environment may require of the software, for example URLs to external resources may differ between environments, or it might be desirable to disable functionality such as authentication for some testing deployments. Externalizing the configuration of these environment dependent behaviours allows a single build of a software to be tailored to operate in multiple environments. The team decided to implement externalized configuration to enable developing and testing production code in a local environment. By setting a profile JVM argument at start up the application can be configured to run against a local development environment or – when deployed to Heroku – against a production environment.

A secondary benefit of externalized configuration is that sensitive configuration, which should be kept secret (for example API keys and passwords), can be removed from the source code of the application and instead passed at runtime via configuration files environment variables. The 2Pizza system requires integration with a database; authentication of requests to the database is managed via username and password, both of which should be kept secret. The team used Externalized Configuration pattern to remove secret configuration from the code version controlled on GitHub to ensure such configuration is not exposed.

The diagram below shows the flow the application takes after start up to select and read an external configuration from a file and then inject this configuration into the correct components. Interpolation of environment variables is done while reading the configuration files, patterns of the form ${SOME_ENV_VAR} are substituted for the value of an appropriate environment variable. Classes can be marked as requiring configuration by adding the @Configuration annotation to the class declaration, such classes are intercepted by a ConfigurationConstructionInterceptor when instantiated by the ApplicationContext as described in the Dependency Injection section of this report. The ConfigurationConstructionInterceptor sets object fields annotated with @Value on configured components with values discovered by the ConfiguationContext class.

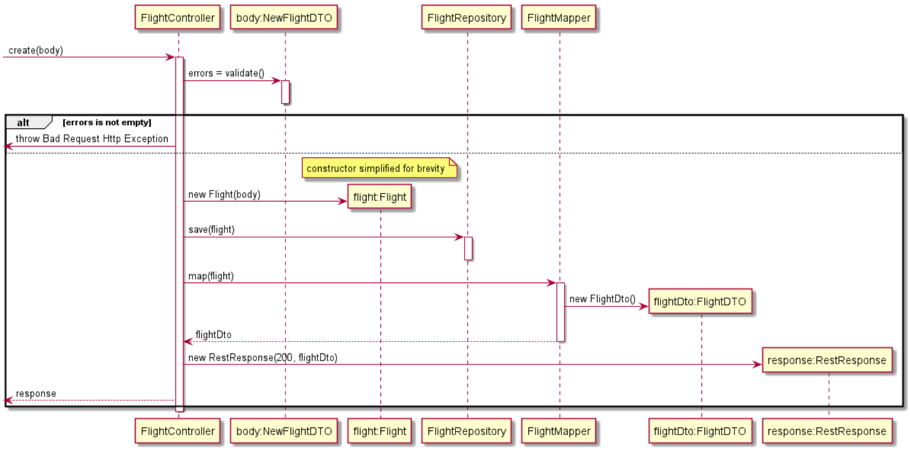


6.19 Data Transfer Object

The Data Transfer Object pattern provides a means to explicitly control the schemas of objects transferred to and from the application. The Data Transfer Object pattern provides a layer of abstraction that allows the specification of the application's API to evolve independently of the business logic (in our case a Domain Model) it exposes.

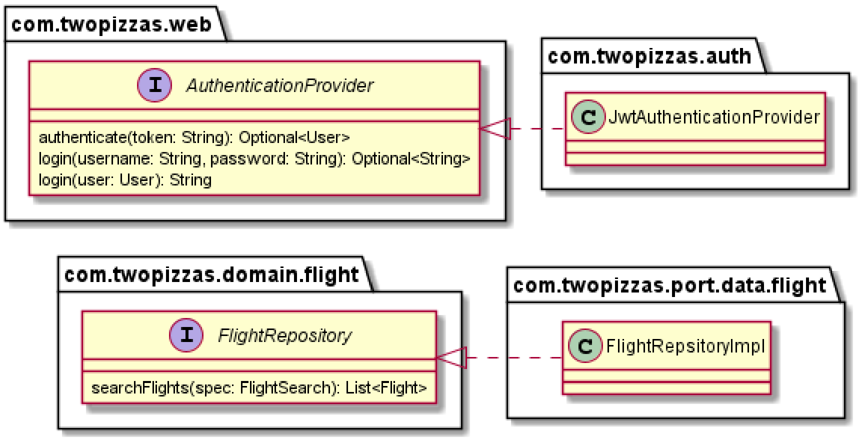
Data Transfer Objects (DTOs) were implemented for the request and response bodies of each endpoint. Validation of each DTO was also delegated to the DTO itself via a validate() method which returns a list of discovered errors. Mapping domain classes to DTOs can be tedious task so the MapStruct framework was introduced to automatically generate much of this mapping code.

The diagram shows the flow for creating a new flight in the system, the validation of the request body DTO and mapping the created Flight domain class to a DTO for serialization as a response body.



6.20 Separated Interface

The Separated Interface pattern was used in multiple areas of the project. Repository interfaces were defined in the domain package and implemented in the port.data packages, further instances include implementing DataSource interface found in the data package within the port.data package and implementing the AuthenticationProvider interface from the web package within the auth package. Separated Interface ensures that a service interface can be collocated with the client code that depends on it while the real implementation can be located somewhere else more appropriate and evolve independently of the client code.



7. Design Rationale

7.1 Unit of Work

The team used Unit of Work pattern to manage transaction boundaries for requests as well as reduce the number of calls to the database. Changes to the Domain Model eventually need to be persisted in the database, Unit of Work provides a mechanism to record changes made within the domain layer and write them out as a single transaction to the database. Our implementation makes heavy use of a Front Controller that initializes a UnitOfWork object (by invoking start()) just prior to delegating to an appropriate handler for the request. During the execution of the request all resulting changes to system entities are registered with the UnitOfWork via a call to either registerNew(), registerDirty() or registerRemoved(). After successfully servicing the request the Front Controller invokes commit() on the UnitOfWork which calls the appropriate Data Mapper to handle the mapping of register entities to the database. If the request is not successful the Front Controller invokes rollback() on the UnitOfWork, and all registered changes are discarded.

The team decided to implement a DataProxy class that proxies requests from the Domain Layer to Data Mappers. This proxy intercepts CRUD requests and instead registers them for execution post request processing where appropriate. A full discussion around the implementation of this Proxy is provided in the Patterns section of this report. The Main benefit of this approach is that neither the Domain Layer nor the Data Mapper implementations need be aware of the UnitOfWork allowing both to focus on execution of business logic and data mapping respectively.

7.2 Lazy Load

Lazy Load was implemented to improve the efficiency of interactions with the database as well as short circuit cyclical dependencies between entities in the domain. The team identified a number of situations where the entire persisted object graph was not required for fulfilling a request to the system, in these instances Lazy Load saves fetching and instantiating objects that are not required and would be immediately discarded either part way through the request or a the end. One such situation is during a retrieval of a Flight object, the graph of objects extending from a Flight include FlightSeatAllocation classes which in turn are associated with a Passenger and Booking, for the purposes of retrieving and displaying a Flight only the Passenger details are required, the Booking of details are discarded. Instead of eagerly loading the Booking details for each Passenger the team implemented Lazy Load to avoid loading Booking details for this use case.

Cyclical dependencies between domain entities can be problematic as they can result in recursive calls to load entities with no clear terminating base case. The 2Pizza Domain Layer features a few such instances of cyclical dependency, such as the cyclical relationship between a Flight entity and its aggregated FlightSeat entities; the Flight object retains a reference to multiple FlightSeat entities and the FlightSeat entities in turn retain a reference to their owning Flight object – primarily for the purposes of implementing Foreign Key Mapping pattern. Lazy Load was implemented to ensure that loading a Flight along with its FlightSeat objects does not reclusively reload the owning Flight. To short circuit this loading pattern the team deferred the load of a Flight into one of its FlightSeats objects until it is required – by which time the Flight will already be present in the IdentityMapper.

Lazy Load was implemented by introducing a ValueHolder generic container class that can be proxied by a LazyValueHolderProxy. When proxied, request to the get() method on instances of a ValueHolder class are intercepted, during this interception if the contained value is had not been loaded then the proxy class initiates a load of the required value.