Product Requirements Document

Submission 1 Specification

**2Pizzas**

SWEN90007 SM2 2021 Project

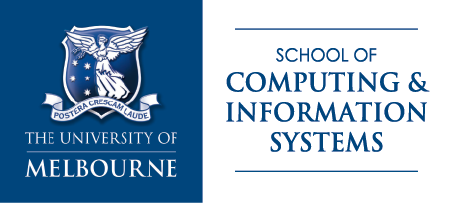
**In charge:**

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 | Move use cases to separate file | Max Plumley |

[1. Introduction 6](#_Toc83326856)

[1.1 Proposal 6](#_Toc83326857)

[1.2 Target Users 6](#_Toc83326858)

[1.3 Assumptions, conventions, terms, and abbreviations 6](#_Toc83326859)

[2. Actors 6](#_Toc83326860)

[3. Use Case 7](#_Toc83326861)

[3.1 Use Case Diagram 7](#_Toc83326862)

[3.2 Use Case 1: Administrator logs in 9](#_Toc83326863)

[3.3 Use Case 2: Administrator creates Airport 9](#_Toc83326864)

[3.4 Use Case 3: Administrator disables Airport 9](#_Toc83326865)

[3.5 Use Case 4: Administrator creates Airline 9](#_Toc83326866)

[3.6 Use Case 5: Administrator searches for Airline 10](#_Toc83326867)

[3.7 Use Case 6: Administrator removes Airline 10](#_Toc83326868)

[3.8 Use Case 7: Airline logs in 10](#_Toc83326869)

[3.9 Use Case 8: Airline creates Flight 10](#_Toc83326870)

[3.10 Use Case 9: Airline cancels Flight 11](#_Toc83326871)

[3.11 Use Case 10: Airline edits Flight 11](#_Toc83326872)

[3.12 Use Case 11: Customer logs in 11](#_Toc83326873)

[3.13 Use Case 12: Customer books Flight 12](#_Toc83326874)

[3.14 Use Case 13: Customer looks up Booking information 12](#_Toc83326875)

[3.15 Use Case 14: Customer searches for a Flight 12](#_Toc83326876)

[4. Domain Model 13](#_Toc83326877)

[4.1 Domain Model Description 13](#_Toc83326878)

[4.2 Domain Model Diagram 14](#_Toc83326879)

[5. Solution Model 16](#_Toc83326880)

[5.1 Class Diagram Description 16](#_Toc83326881)

[5.2 Class Diagram 16](#_Toc83326882)

[6. Patterns Used 16](#_Toc83326883)

[6.1 Domain Model 16](#_Toc83326884)

[6.2 Identity Mapper 16](#_Toc83326885)

[6.3 Data Mapper 16](#_Toc83326886)

[6.4 Unit of Work 16](#_Toc83326887)

[6.5 Layer Supertype 16](#_Toc83326888)

[6.6 Registry 16](#_Toc83326889)

[6.7 Dependent Mapping 16](#_Toc83326890)

[6.8 Inheritance Mapper 16](#_Toc83326891)

[6.9 Lazy Load 16](#_Toc83326892)

[6.10 Identity Field 16](#_Toc83326893)

[6.11 Foreign Key mapping 16](#_Toc83326894)

[6.12 Association Table Mapping 16](#_Toc83326895)

[6.13 Embedded Value 16](#_Toc83326896)

[6.14 Class Table Inheritance 16](#_Toc83326897)

[6.15 Authentication with JSON Web Token Authentication Provider 16](#_Toc83326898)

[6.16 Front Controller 16](#_Toc83326899)

[6.17 Inversion of Control with Dependency Injection 16](#_Toc83326900)

[6.18 Repository 16](#_Toc83326901)

[6.19 Externalized Configuration 16](#_Toc83326902)

[6.20 Data Transfer Object 17](#_Toc83326903)

[6.21 Separated Interface 17](#_Toc83326904)

[7. Design Rationale 17](#_Toc83326905)

[7.1 Unit of Work 17](#_Toc83326906)

[7.2 Lazy Load 17](#_Toc83326907)

# Introduction

## Proposal

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

## Target Users

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

## 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.

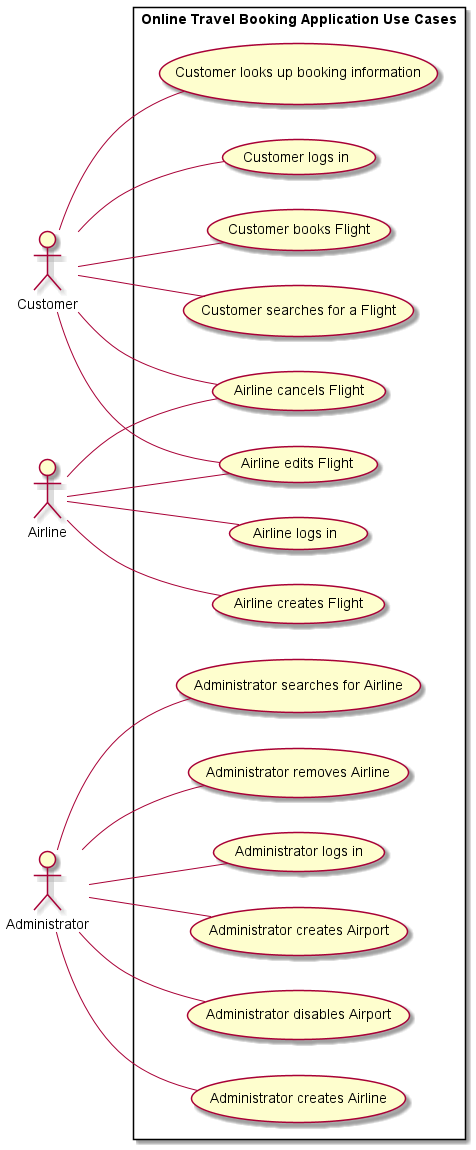
|  |  |
| --- | --- |
| Term | Description |
| Flight | A flight is a trip involving a single airline and airplane type that can contain multiple stopover airports between the origin and destination airports.  A return flight is another trip that will be booked using the same airline and airports (including stopovers), but with a flipped itinerary.  Example: Return flight with Airline A from Melbourne to Jakarta, through 1 stopover in Singapore  Original flight = MEL -> SIN -> CGK using Airline A  Return flight = CGK -> SIN -> MEL using Airline A |
| URL | Uniform Resource Locator |
| Username | Equivalent to and interchangeable with Email |

# Actors

|  |  |
| --- | --- |
| Actor | Description |
| Administrator | *An individual who is responsible for managing the application, which includes adding airlines and airports to the system and aggregating flights.* |
| Airline | *An entity who is responsible for adding, removing, and modifying flights from their airline on the system.* |
| Customer | *Individuals who use the application to search and book flights.* |

# Use Case

## Use Case Diagram



## Use Case 1: Administrator logs in

**Actors**

1. Administrator

**Preconditions**

* Account has already been created for the administrator

**Basic Flow**

Administrator enters the website URL for the travel booking application, which directs them to the application home page. Then they click on the Login option in the navigation bar and are presented with the email and password text fields, where they then enter their login credentials. Finally, upon clicking the Login button, they are presented with the Administrator dashboard page, which contains Administrator-only actions such as managing airports and airlines.

## Use Case 2: Administrator creates Airport

**Actors**

1. Administrator

**Basic Flow**

Administrator logs into system and selects option to create a new airport, inputs airport name, short name and location and selects create. If the airport does not already exist, then the input is persisted in the system and a success message is shown. Alternatively, if the airport already exists an error is shown to the Administrator.

## Use Case 3: Administrator disables Airport

**Actors**

1. Administrator

**Preconditions**

* Airport currently exists and not disabled in the system

**Basic Flow**

Administrator logs into system and selects an existing airport and selects disable and a success message is shown to the Administrator. Airlines can no longer create flights such that the disabled airport acts as the origin, stopover, or destination airport.

## Use Case 4: Administrator creates Airline

**Actors**

1. Administrator

**Basic Flow**

Administrator logs into system and selects to create a new Airline. The administrator inputs the Airline name, Airline code and the email address and name of the Airline user account and selects create. If the Airline does not already exist, the Airline details and Airline user account is persisted by the system and a success message is shown to the Administrator. If the Airline already exists, then an error message is shown to the Administrator.

## Use Case 5: Administrator searches for Airline

**Actors**

1. Administrator

**Basic Flow**

Administrator logs into system, selects the search bar, and inputs the name of the Airline they are searching for. If there are no airlines in the system or if no airlines match the search query, then the list is empty.

## Use Case 6: Administrator removes Airline

**Actors**

1. Administrator

**Preconditions**

* Administrator is logged in and has searched for airlines, such that the results are available

**Basic Flow**

Administrator selects an Airline from the search results. The administrator is presented with a page containing a remove option and the details of the airline, which includes the airline name and code, the email address and name of the airline’s user account, and a list of the airlines ongoing or future flights. The remove option is only enabled if the Airline does not have any ongoing or future flights. Then the administrator clicks the remove option, and the airline user account is disabled

## Use Case 7: Airline logs in

**Actors**

1. Airline

**Preconditions**

* Account has already been created for the airline

**Basic Flow**

Airline enters the website URL for the travel booking application, which directs them to the application home page. Then they click on the Login option in the navigation bar and are presented with the email and password text fields, where they then enter their login credentials. Finally, upon clicking the Login button, they are presented with the Airline dashboard, which contains Airline-only actions such as managing flights.

## Use Case 8: Airline creates Flight

**Actors**

1. Airline

**Preconditions**

* Airline has an existing account in the system
* Origin and destination airport have been added into the system

**Basic Flow**

Airline logs into system and selects the option to create a flight. Then they select the airplane type for the flight, the origin airport, destination airport, departure time, and stopover airport(s) if applicable. They then click the Confirm button to finalise the flight, a success message is then shown to the Airline.

## Use Case 9: Airline cancels Flight

**Actors**

1. Airline
2. Customer

**Preconditions**

* Airline has an existing account in the system
* Airline has existing future flight(s) already in the system

**Basic Flow**

Airline logs into system and selects to view all flights, they then select the Cancel option for a chosen flight that is due to commence in the future. The Cancel option for flights that are ongoing or have run are disabled. They then are presented with the details of the flight including the origin, stopover (if any), destination airports and the departure and arrival times from and to each airport, the seats that have been booked for the flight, an Edit button, and a Cancel button. Finally, the Airline clicks the Cancel button to finalise the flight’s cancellation, a success message is shown to the Airline and a notification email is sent to the customer.

## Use Case 10: Airline edits Flight

**Actors**

1. Airline
2. Customer

**Preconditions**

* Airline has an existing account in the system
* Airline has existing future flight(s) already in the system

**Basic Flow**

Airline logs into system and selects to view all flights, they then select the Edit option for a chosen flight that is due to commence in the future. The Edit option for flights that are ongoing or have run are disabled. They then are presented with the details of the flight including the origin, stopover (if any), destination airports and the departure and arrival times from and to each airport, the seats that have been booked for the flight, an Edit button, and a Cancel button. The Airline then clicks the Edit button to initiate editing the flight. They are then able to edit the time and/or date of the flights. The Airline then clicks the Confirm button and a success message is shown if the change is possible and a notification email is sent to the customer, else an error message is shown if there is a clash with the use of the airplane.

## Use Case 11: Customer logs in

**Actors**

1. Customer

**Preconditions**

* Account has already been created for the customer

**Basic Flow**

Customer enters the website URL for the travel booking application, which directs them to the application home page. Then they click on the Login option in the navigation bar and are presented with the email and password text fields, where they then enter their login credentials. Finally, upon clicking the Login button, they are presented with the home page, which shows them search bar for flights.

## Use Case 12: Customer books Flight

**Actors**

1. Customer

**Precondition**

* Customer has searched for flights and the results are available

**Basic Flow**

Customer selects a flight from the search results, if the customer has elected to search for return flights, then the customer also selects a return flight from the return flight search results. The customer is presented with a page to select their seats for each selected flight, the customer selects seats on each flight for each passenger and selects done. The customer is presented with a page to input the details for each passenger, such as passenger name, age, and passport ID, the customer inputs their email address and selects book. The booking is persisted in the system and the seats are reserved to the booking; the details of the booking are emailed to the customer at the email address provided.

## Use Case 13: Customer looks up Booking information

**Actors**

1. Customer

**Precondition**

* Customer is an existing user of the application
* Customer has an existing flight booked on the system

**Basic Flow**

Customer logs into the system, they then click on the View Bookings button. They then are presented with a list of all future and ongoing flights that they have booked; they can also select to view previous bookings. When they click on a booking, their booking details including passenger details, flights date(s) and time(s) and seat numbers will be displayed.

## Use Case 14: Customer searches for a Flight

**Actors**

1. Customer

**Basic Flow**

Customer selects to search for flights, customer inputs the number of passengers that will be travelling as well as origin and destination airports, a range of departure dates, and whether the flight is to be a one-way or return flight, if the flight is a return flight the customer also selects a range of dates for the return flight, finally the customer selects search. The system looks up available flights for each Airline and each seat class available for each flight and presents the results to the customer, if the search is for a return flight, then the system also searches for all appropriate return flights and presents them as a second list to the customer.

# Domain Model

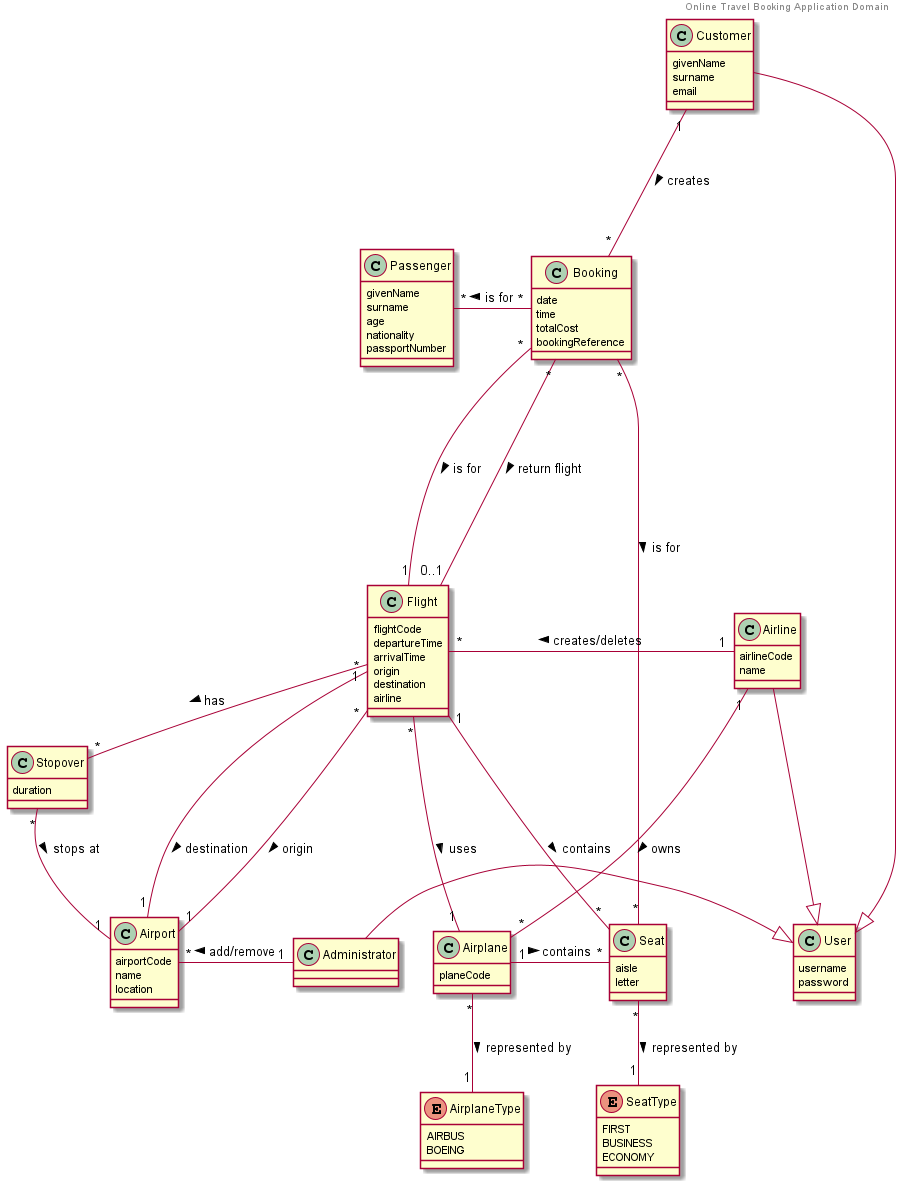
## Domain Model Description

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.

## Domain Model Diagram



# Solution Model

## Class Diagram Description

## Class Diagram

# Patterns Used

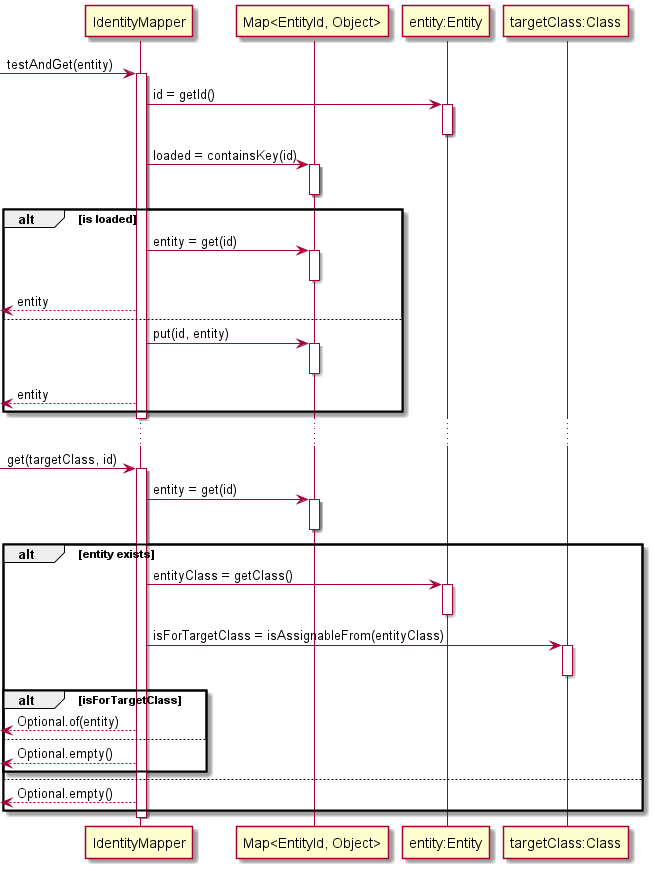
## Domain Model

## Identity Map

Becuase 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 a number of 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 retireved or a prior retrieved instnace if one is already registerd 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 etiher that entity if it has not been loaded or a prior registered instance.

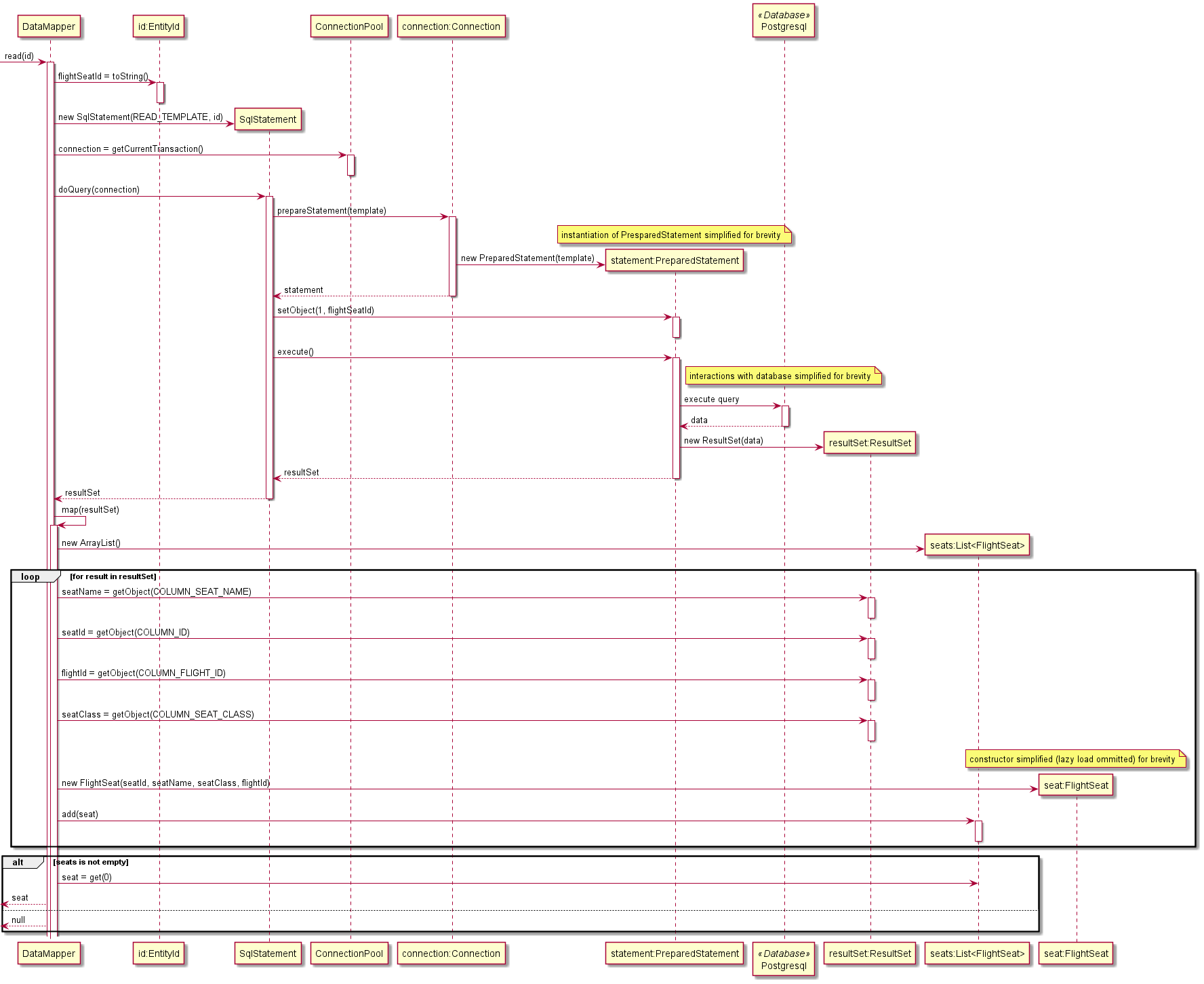


## Data Mapper

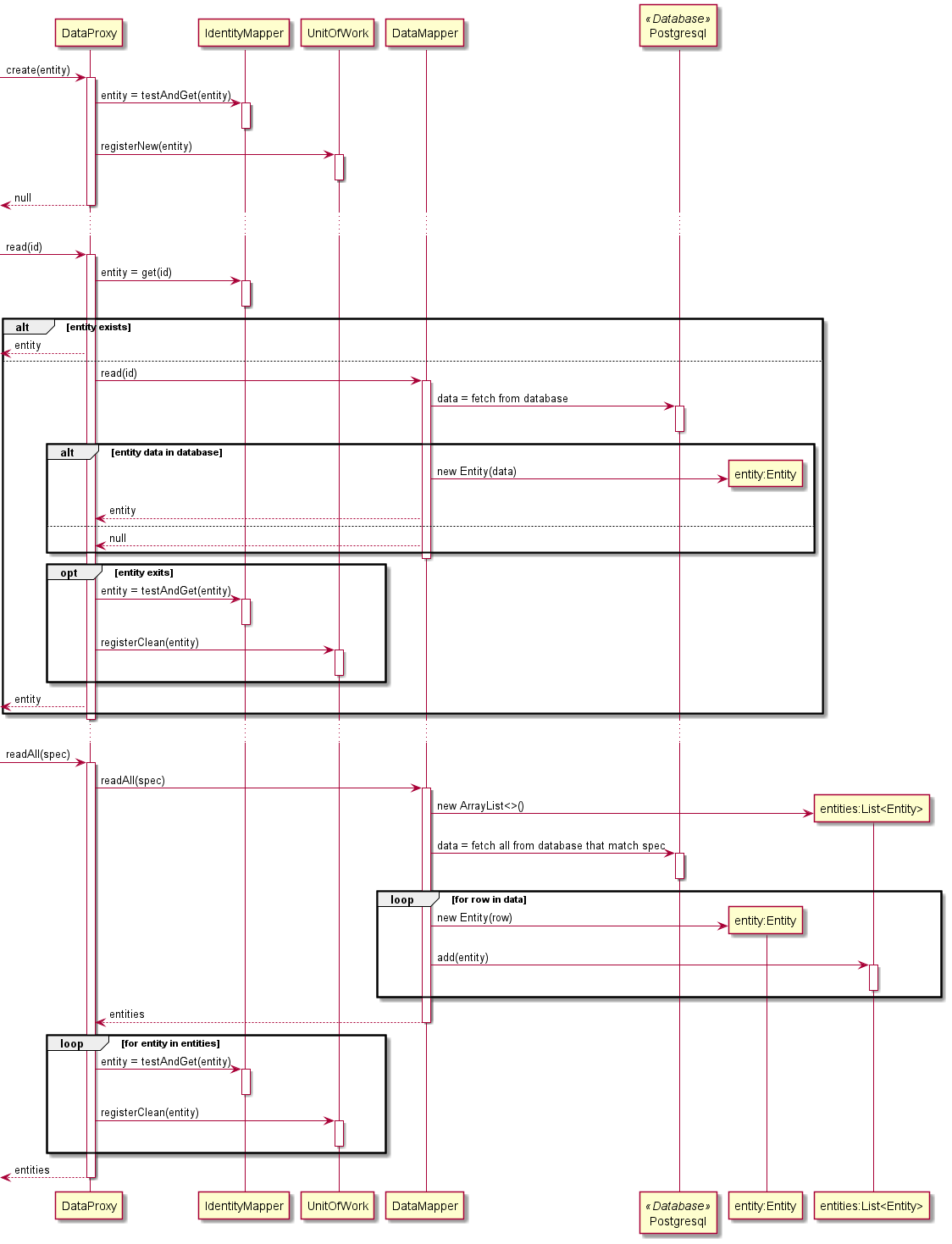
The Data Mapper pattern was implemented for each entity in the domain - such as Flight, FlightSeats, Passenger etc. A Separated Interface was used to define a generic interface to be used by client code in data layer and implemented elsewhere for each entity that requires persistence. This Separated Interface exposes the basic CRUD operations as well as a findAll() method that accepts a generic Specifcation 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 tranasaction and additionaly to guard against instantiating multiple instances of the same entity while retrieving entites from the database. Rather than relying on correct and consistant implmentation 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 recieved 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 exists for the other CRUD operations and are ommited for brevity



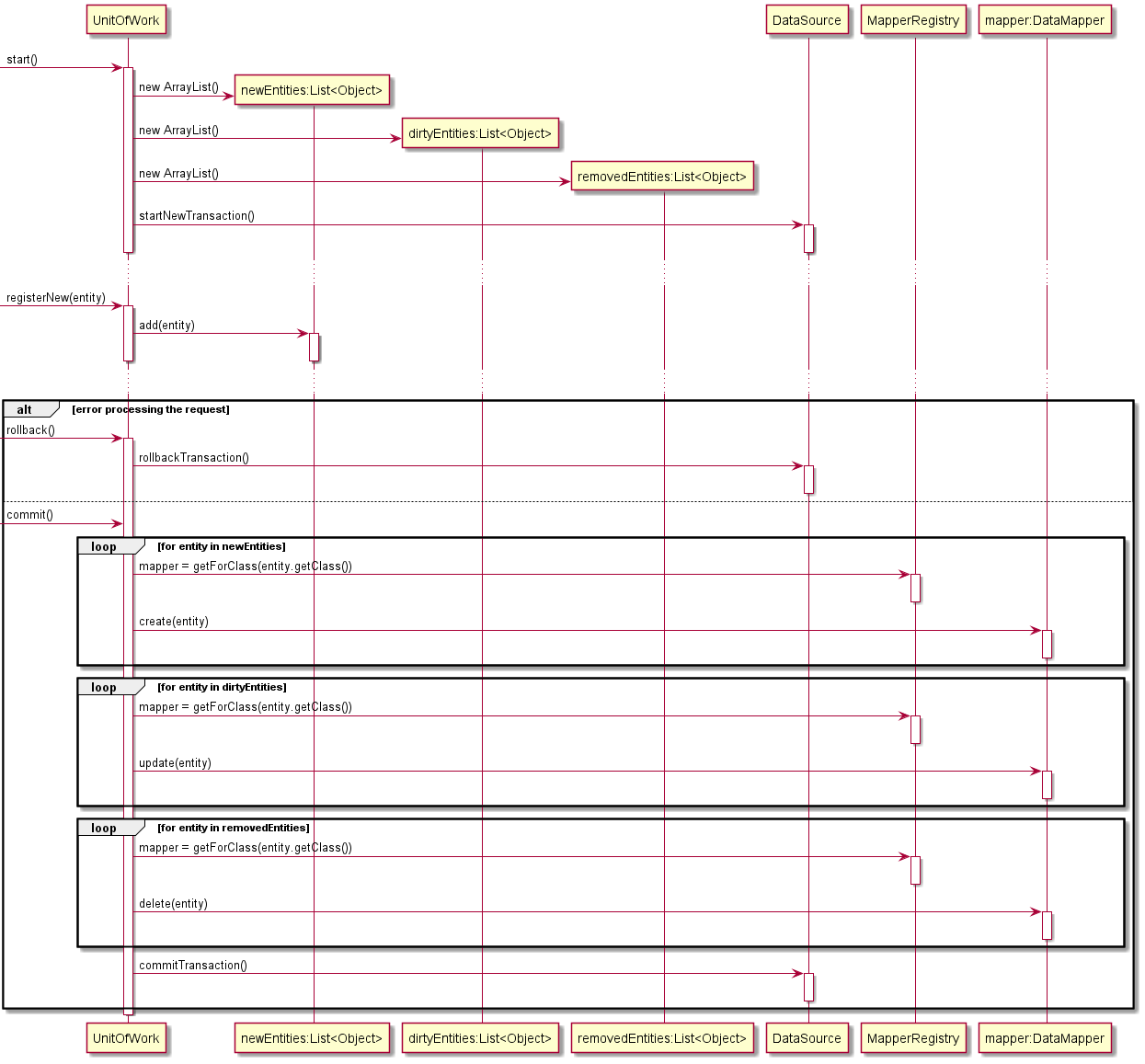
The diagram below shows the flow taken by the DataProxy class when invoked with a subset of the DataMapper CRUD methods. The primary responsibility of the DataProxy class is to intercept requests from the domain layer that manipulate database resources. The nature of this interception behaviour is relatively consistant 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 entites 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.



## Unit of Work

Implementation of the Unit of Work pattern was fairly straight forward, the interface exposes hooks used by a Front Contoller 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 implementaiton makes use of a DataMapperRegistry that has access to un-proxied versions of each mapper in the domain (the rest of the application recieves 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.



## Layer Supertype

This basically just DomainEntity and AbstractRepository

## Registry

MapperRegistry used by UnitOfWork

## Dependent Mapping

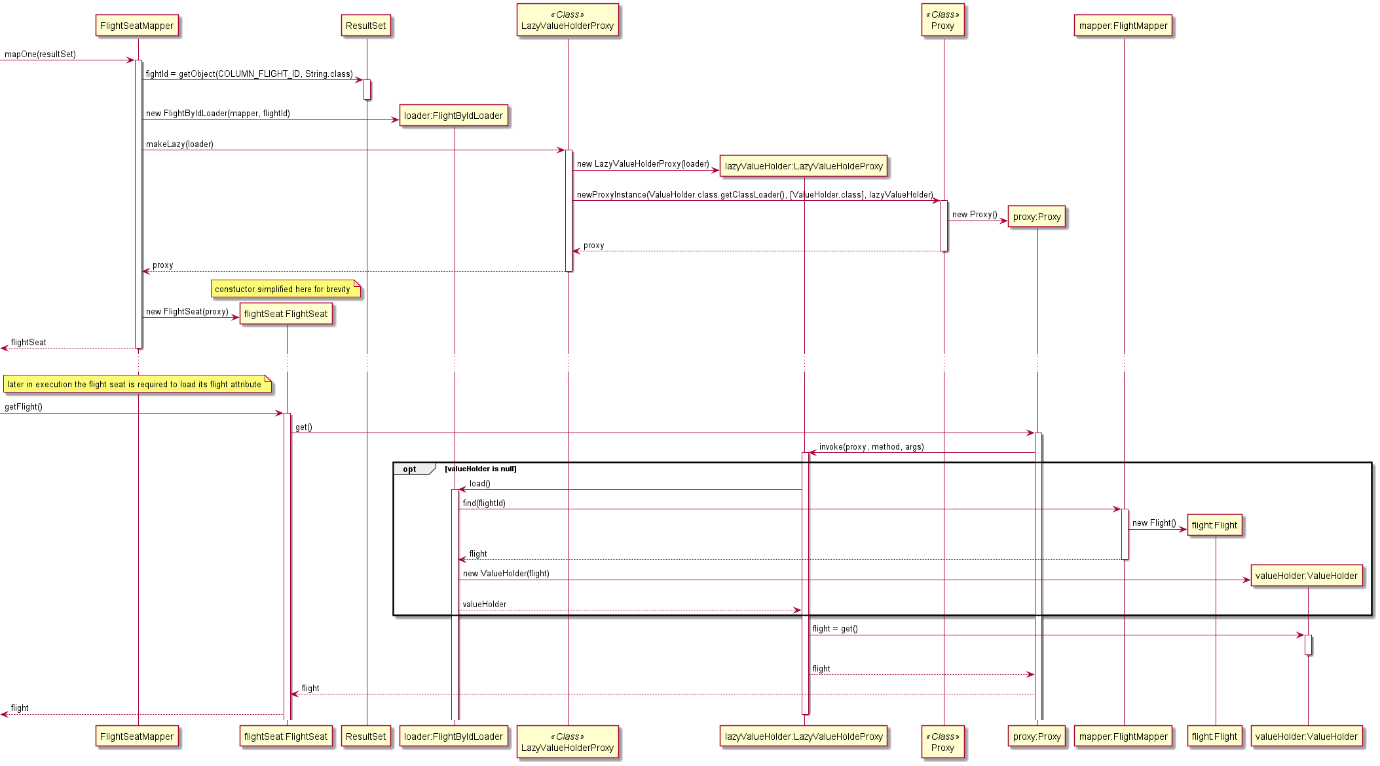
## Inheritance Mapper

## 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.

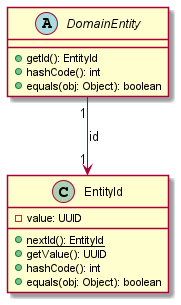
The diagram shows the flow taken to instantiate a LazyValueHolderProxy for the Flight field of a FlightSeat class and the subsequent flow taken if and when the wrapped value holder value is required in the course of application execution.



## 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 consistancy 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 overriden hashCode() and equals(Object) methods, the overriden implmentations for EntityId use the encapsulated UUID while the implementations for DomainEntity delegate equality to the encapsulated EntityId.



## Foreign Key mapping

## Association Table Mapping

## Embedded Value

SeatAllocation and also Stopover classes

## Class Table Inheritance

## Authentication with JSON Web Token Authentication Provider

## 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, interpretting 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 controler. The system makes use of Dependency Injection pattern to discover and instantiate controller classes at runtime.

## Inversion of Control with Dependency Injection

Inversion of controll allows generic control code to be implemented by a framework that calls into specially implemented objects that encapsulate the specific behaviour of the domain. This process allows the wrapping framework to be decoubpled from the specifics of a software implementsation and thus be reused across multiple domains. Our project uses inversion of controll 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 Controll makes use of Dependency Injection to transparently load domain specific implementations where required by the framework. For example implementions 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. One of the key benefits of Dependancy Injection is to reattribute the responsibily of finding or instantiating service objects from the client objects to a managing framework, this reatribution allows for a single point at which to manage dependencies across an application and the oportunity 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 staticaly decided at compile time. Dependency Injection is arguablly superior to Static Singleton pattern as it has no such restrictions, which is a huge benefit with respect to testability of an application as it allows for tests to exercise the client code against Service Stubs rather than the real implementations that would normally be shipped to production. The team has written various unit tests for critical parts of the application including implementations of Data Mappers, these tests utilize Service Stubs generated with the Mockito framework to easily exercise branching target code.

The diagram below shows the flow the application takes to intantiate and initialize an ApplicationContext for the purposes of Dependency Injection. Some implementation details to note are that: the ComponentLoader class uses reflection to discover classes annotated with @Component annotation. The presence of the @Component annoation on a class declaration marks the class as a class that should be managed by the application context 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, recursivly instantiating service components on which client depends before injecting these dependencies at the point of instantiating client components.

## 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 persistance of domain objects, CRUD opperations 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 impemented in DomainEntity Layer Supertype and is futher 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 typlically only extend the abstract functionality by constructing appropriate Specification objects imput to a findAll() invocation on a wrapped Data Mapper class.

## Externalized Configuration

Often it is required that an application be deployed to multiple envronments, the application may be deployed to a testing environment during the execution of CICD pipeline, and then 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 nessary 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 behavious allows a single build of a software to be tailored to operate in mutliple environments. The team decided to implement externalized configuration to enable developing and testing production code in a local environment. By setting a profile JVM argurment at startup 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 or via 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 instead provide secret values as environment variables that can be managed securely with gitActions.

The diagram below shows the flow the application takes after startup to select and read an external configuration from a file and then inject this configuration into the correct components.

## Data Transfer Object

The Data Transfer Object pattern provides a means to explicitely control the nature of

## Separated Interface

The Separated Interface pattern was used in multiple areas of the project. Repository interfaces we 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 in web package with in 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.

# Design Rationale

## Unit of Work

## Lazy Load