Architecture Document

SWEN90007

Marketplace System

Team: Agora

Student Name	Student ID	UniMelb Username	GitHub Username	Email
Daniel Blain	831953	dblain	djblain	dblain@student.unimelb.edu.au
Christopher Byrnes	747295	byrnesc	chrisbyrnes	byrnesc@student.unimelb.edu.au
Gonzalo Molina	1085253	gmolina	GonzMol	gmolina@student.unimelb.edu.au
Mengjiao Wei	1242147	mengjiaow1	mengjiaowei	mengjiaow1@student.unimelb.edu.au

SCHOOL OF
COMPUTING &
INFORMATION
SYSTEMS

Revision History

Date	Version	Description	Author
19/09/2022	01.00-D01	Initial version the domain model, MVC, identify field, foreign key mapping, association table mapping, embedded value	Mengjiao Wei, Christopher Byrnes
19/09/2022	01.00-D02	Initial version the unit of work, lazy load, overview for architectural patterns	Christopher Byrnes
19/09/2022	01.00-D03	Initial version the authentication and authorisation	Daniel Blain
19/09/2022	01.00-D04	Initial version the data mapper, inheritance patterns	Gonzalo Molina
20/09/2022	01.00-D05	Update domain model, MVC, identify field, foreign key mapping, association table mapping, embedded value with diagrams	Mengjiao Wei
20/09/2022	01.00-D06	Added diagrams for unit of work, lazy load, overview for architectural patterns	Christopher Byrnes
20/09/2022	01.00-D07	Added diagram for authentication and authorisation, initial version introduction	Daniel Blain
20/09/2022	01.00-D08	Added diagrams for data mapper, inheritance	Gonzalo Molina
21/09/2022	01.00-D09	Source code directories structure, development environment	Gonzalo Molina
21/09/2022	01.00-D10	Proof-reading/grammatical for the report; document formatting, reference	Daniel Blain, Christopher Byrnes, Mengjiao Wei, Gonzalo Molina
21/09/2022	01.00-D11	Updated terms, use cases, diagrams	Daniel Blain, Christopher Byrnes, Mengjiao Wei, Gonzalo Molina
21/09/2022	01.00-D12	Finalised document for submission	Daniel Blain, Christopher Byrnes, Mengjiao Wei, Gonzalo Molina

Table of Content

1. Introduction

This document specifies the architecture of the Agora Marketplace system, describing its main standards, modules, components, *frameworks* and integrations.

1.1 Proposal

The purpose of this document is to give a high-level overview of the implemented system, including a discussion of the patterns chosen and the justifications for these decisions.

1.2 Target Users

This document is aimed at the teaching staff responsible for assessing the system and work done.

1.3 Conventions, terms and abbreviations

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

Term	Description
Admin	Special User responsible for overseeing the system's use and managing Listings and Users; short-hand for Administrator
Auction Good	A Good which multiple site users may place a bid on to try and acquire the item; the Good is sold to the user who places the highest bid
Bidding	The act of placing a bid on an Auction good
Buyer	A Marketplace User who purchases Goods
Co-seller	A nominated User which co-manages a Listing created by a Seller
Fixed Price Good	A Good, or Goods, to be sold at a single, non-negotiable price
Good	A product detailed for sale in a Listing
Listing	An entry for a product to be sold as it appears on the Marketplace site
Marketplace	The digital storefront proposed in this document, for buying and selling physical Goods
Order	A purchase, complete or incomplete, of a Good, detailing payee and shipping details
Seller	A Marketplace User who creates Listings for Goods to be sold
User	An individual, or business, registered with the Marketplace system, who can use its services to purchase or sell Goods

1.4 Actors

Actor	Description
Administrator	Manages User accounts and Listings; only one Administrator exists for the Marketplace system
All users	Refers to Standard users and Administrators collectively
Standard User	Standard users can create and purchase listings on the system.

2. Use cases

Each use case contains the name, actor, and basic flow describing the sequence of actions performed to achieve the use case.

Use Case 1: Search for a good

Actor: All users

Basic Flow:

When all users open the marketplace, the active listings are displayed on the homepage. At the top of the homepage is a search bar where users can select to either search by item name or seller username to find relevant listings. After selecting a search method and entering search keyword(s), a subset of matching listings is displayed on the current page.

Use Case 2: View listing

Actor: All users

Basic Flow:

When all users open the marketplace, the active listings are displayed on the homepage without login required. The listings are sorted by most recently created by default. To view more information about the good, the user clicks on a single listing, which redirects them to the listing detail page.

Use Case 3: Manage account details

Actor: All users

Basic Flow:

After logging into the system, users access their account details via the navigation bar. Within the account details page both administrators and standard users can view their current account details, and additionally, standard users can view their current shipping details. From within the page, users can update all of their account details except for their username.

Use Case 4: Create account

Actor: Standard user

Basic Flow:

New users are required to create an account in order to purchase and sell any goods on the system. To create a new account, the user provides a username, email address, password, and shipping details. Their username and email address must be unique. After creating an account, their details are saved in the database and they are able to log into the system with their email address and password.

Use Case 5: Manage listings

Actor: All users

Basic Flow:

All standard users can manage listings if they are the owner of the listing or have been added as a co-seller. To manage a listing they navigate to my listings, which displays all of their active listings, and click on a single listing. From within the listing detail page they click edit listing to modify the listing details, add or remove co-sellers or deactivate the listings. Administrators have the ability to deactivate any inappropriate listings in the system.

Use Case 6: Create listing

Actor: Standard user

Basic Flow:

All standard users can create listings, which can either be fixed price listings or auction listings. Both listing types require users to provide a title, description of good(s), condition, category, and (optionally) co-sellers. Fixed price listings also require a fixed price and stock, while auction listings require a starting price, start time and end time. After the user finishes entering all the details for the listing, they click the create button and the listing is entered into the system.

Use Case 7: Manage orders

Actor: All users

Basic Flow:

Standard users can view, modify and cancel orders for goods they have purchased themselves (purchases) and for goods they have sold (sales). When modifying a purchase order for a fixed price listing, they can increase or decrease the quantity, however, when modifying a sale order for a fixed price listing they can only decrease the quantity. Administrators have the ability to view all orders placed in the system.

Use Case 8: Bid on auction

Actor: Standard user

Basic Flow:

After standard users log in, they browse through the active auction listings and select a good they wish to bid on. Within the listing details page they enter a bid amount which is greater than either the item's starting price, or the current highest bid if there are existing bids. Their bid is then entered into the system and if they are the highest bidder when the auction ends, they win the auction, resulting in the system creating an order for them.

Use Case 9: Purchase a fixed price good

Actor: Standard user

Basic Flow:

After standard users log in, they browse through the fixed price listings in search of what they want to purchase. Once they find a good they would like to purchase, they select the listing, enter a quantity and click checkout. After they confirm their shipping details, a new order is placed for the good(s).

Use Case 10: Manage all accounts

Actor: Administrator

Basic Flow:

After logging in, the administrator can view all active and inactive accounts registered in the system by navigating to manage all accounts page. From within this page, they also have the ability to deactivate any accounts belonging to problematic users.

3. Architectural Patterns

3.1. Overview

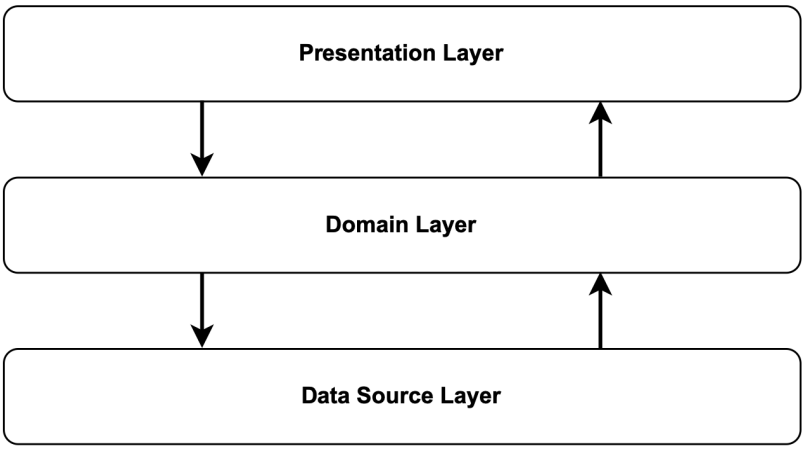

Figure 3.1.1. Overview of the layers in Agora Marketplace

The Agora Marketplace system makes use of three layers: the presentation layer, domain layer and data source layer. The presentation layer contains the model-view-controller (MVC) pattern, the domain layer contains the domain model pattern, and the data source layer makes use of the data mapper pattern, each of which are explained in further detail below.

3.2. Data Mapper

The data mapper pattern serves as a way to move in-memory data between domain objects and a database whilst also keeping them isolated from each other. The role of the data mapper is to load in-memory domain objects to and from the database, keeping the data consistent across both once a transaction is complete. This is achieved by calling data mapper methods which *insert*, *update* and *fetch* rows in the database.

The benefits of using data mapper for this project was that in memory domain objects are isolated from any database connection or SQL logic. The objects are unaware of the existence of a database and their implementation in the domain is not dependent on it. This results in domain objects that are loosely coupled with their corresponding rows in the database. That is to say that the objects are free to be manipulated and differ from what is in the database while they are in use. An added benefit of this was the development team's ability to work on both the domain model and data mappers concurrently as the functionality of domain objects did not require any knowledge of the database implementation.

The data mapper pattern is also highly compatible with the domain model pattern, implemented in this project, where each domain object is created by the data mapper objects. Similarly to the justification of using domain model pattern for this project, this pattern encourages reusability and extensibility of the data source layer.

Implementation

Most data mappers implemented in this project follow the same structure with a few exceptions. Most have an *insert, update* and some form of *a fetch* method. A design decision by the team was not to delete *listing*, *user* or *order* records from the database but instead mark them as *inactive* or *cancelled*.

This was because the *admin* would still need the ability to view *listings* or *users* that have been deactivated, and *users* still need the ability to view cancelled *orders*.

The mappers in this project move data bi-directionaly to and from the database. *Insert* and *update* methods move data towards the database whilst the *fetch* methods move data from the database to the domain.

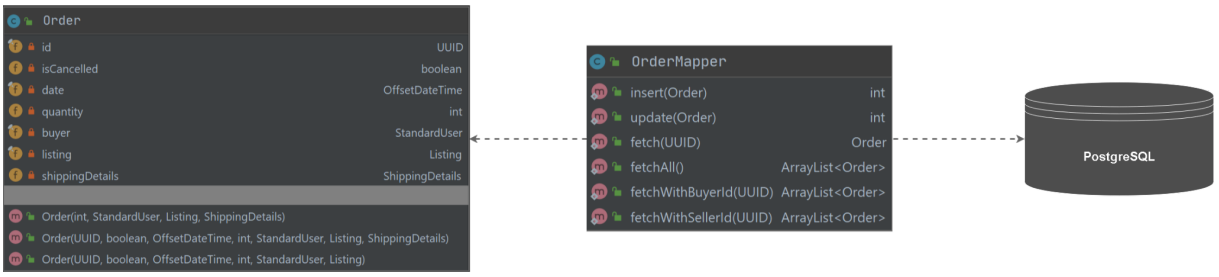

Figure 3.2.1. Data flow of the OrderMapper implementation

The following sequence diagrams illustrate the general functionality of each method that is commonly used amongst each DataMapper Class.

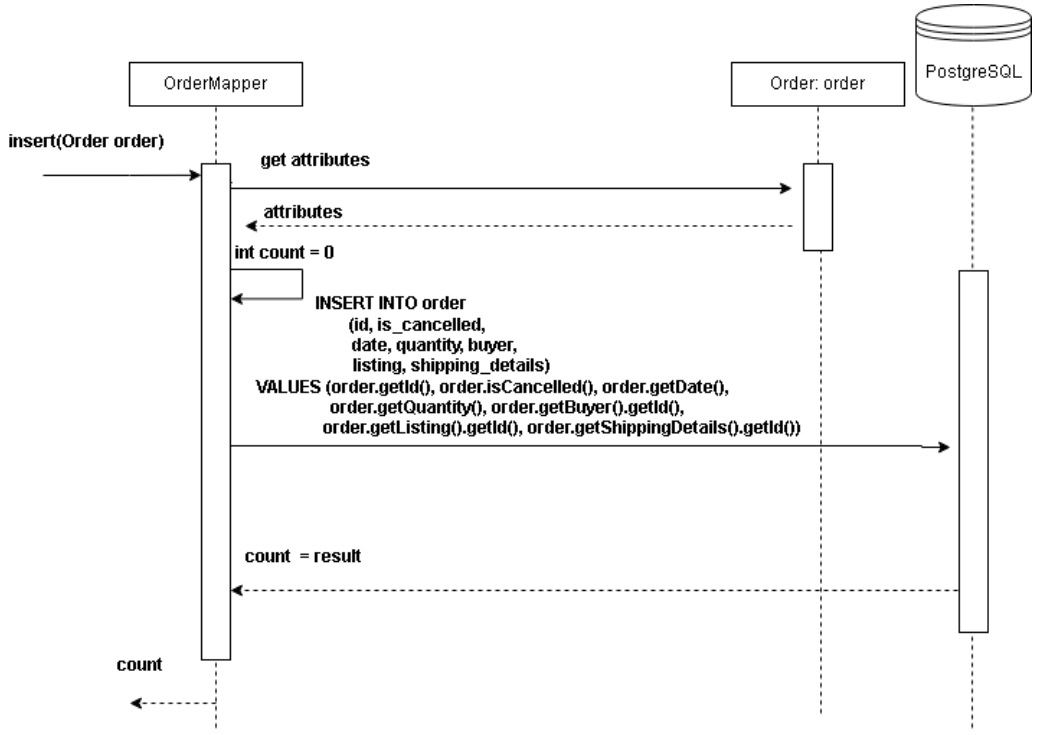

Figure 3.2.2. OrderMapper *insert* sequence diagram

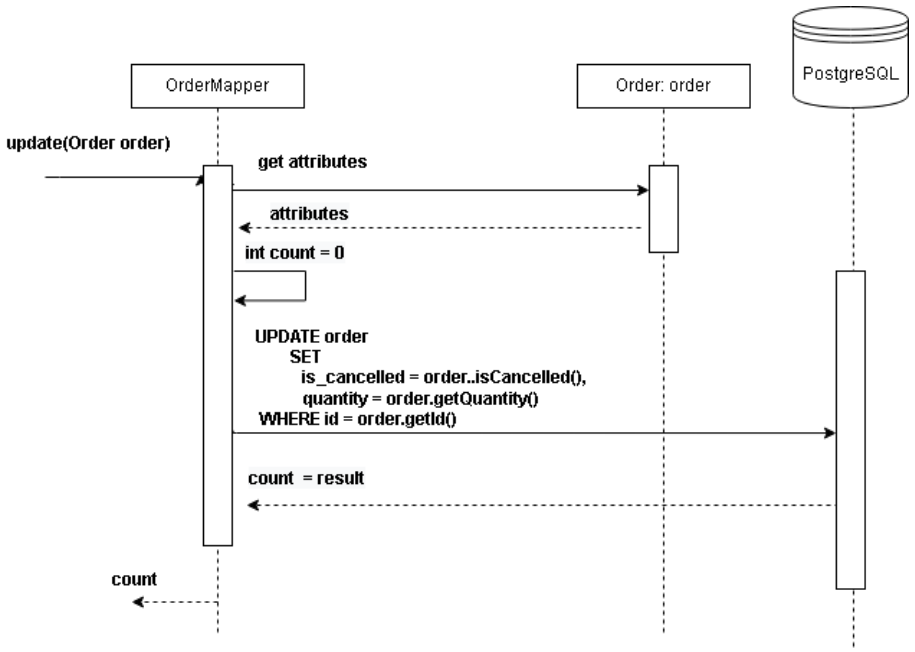

Figure 3.2.3. OrderMapper *update* sequence diagram

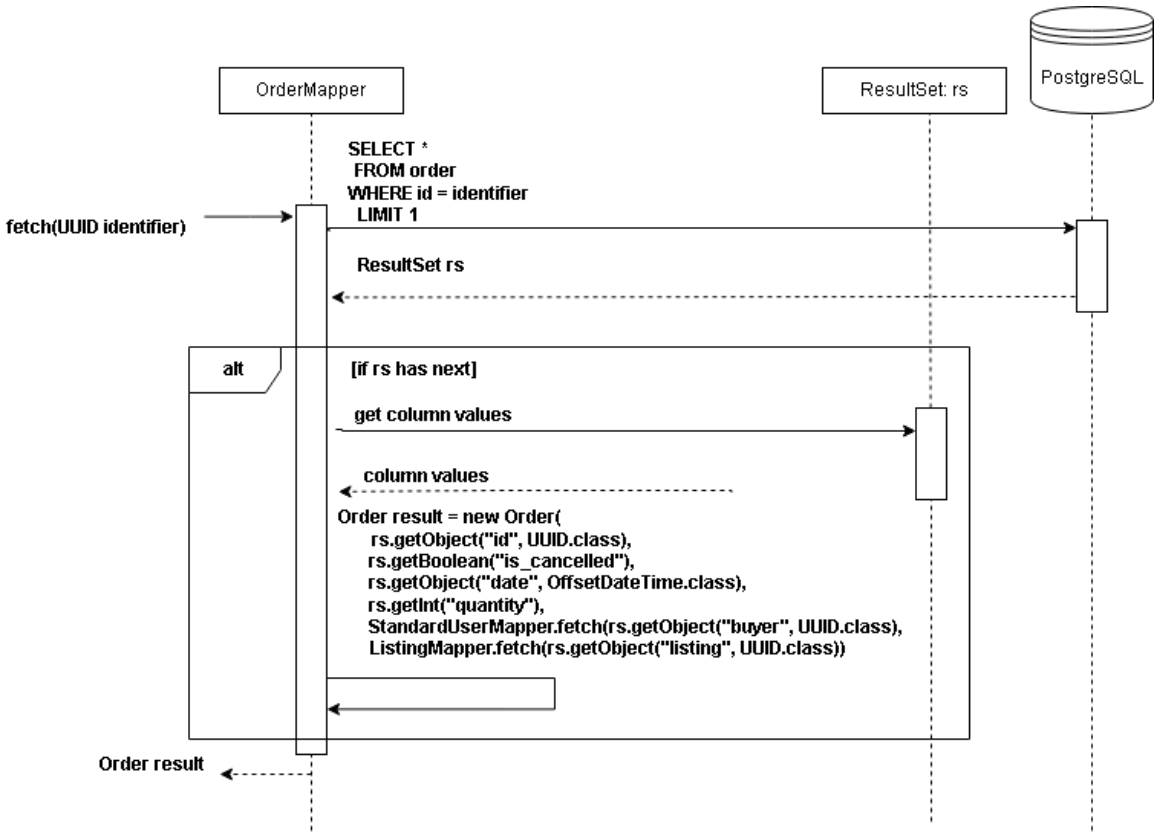

Figure 3.2.4. OrderMapper *fetch* sequence diagram

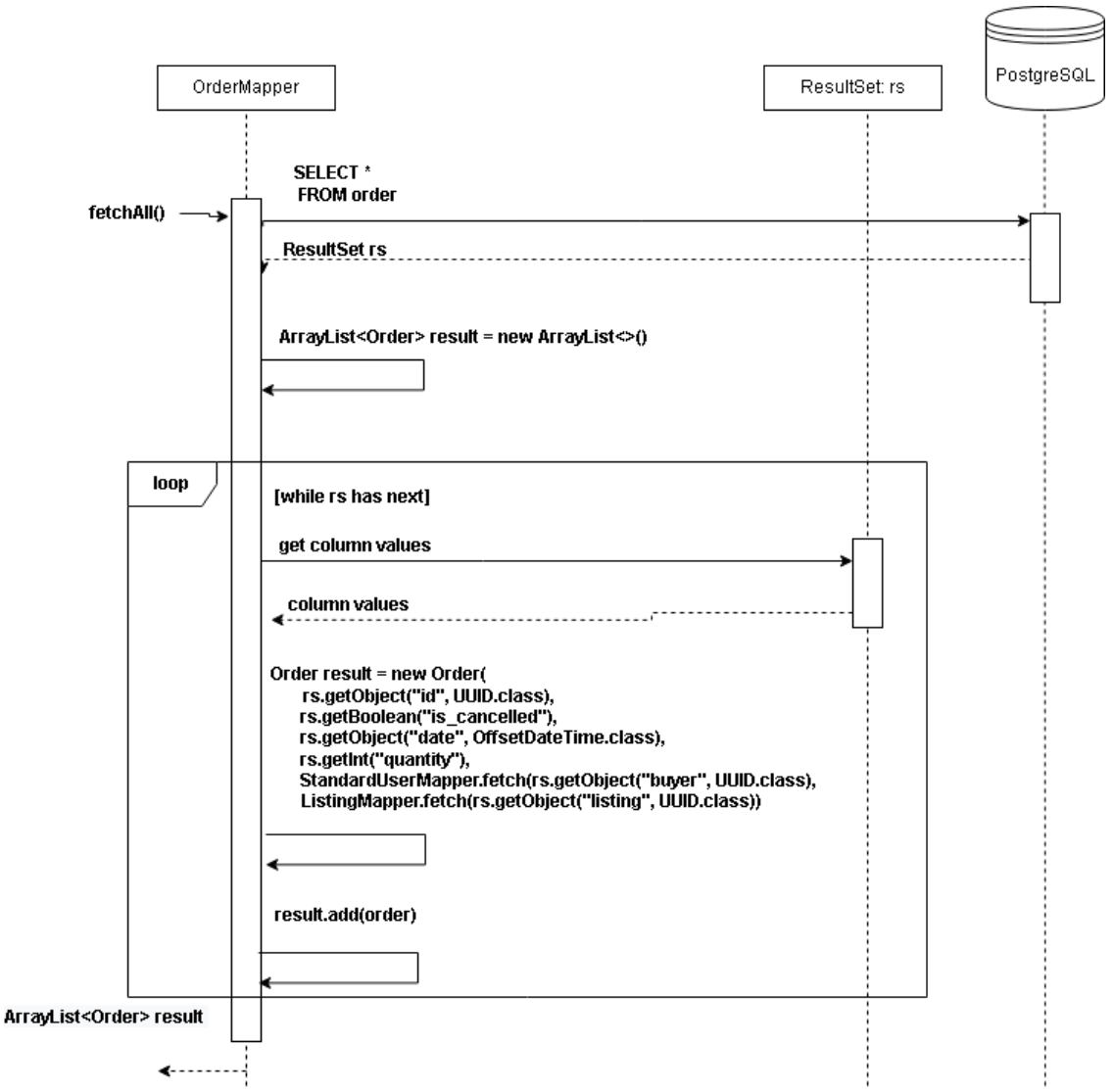

Figure 3.2.5. OrderMapper *fetchAll* sequence diagram

Justification

The decision to use data mappers for this project was mostly made due to the implementation of the domain model pattern and the use of inheritance throughout the system.

Table data gateway and row data gateway are not compatible with the domain model pattern, and hence, were not considered.

The development team did not want to introduce any performance issues which may arise from the use of the active record pattern, which couples domain objects with the database and requires multiple database connection calls to keep the domain objects representative of what information is in the database.

3.3. Inheritance

Single Table Inheritance

The single table inheritance pattern serves to represent the hierarchy of an object oriented class into a single relational database table. That is to say that each entry in the table maps to all attributes of all classes that are members of that hierarchy. For this project, the pattern was used to map the hierarchy of the *Listing* class which consisted of both *AuctionListing* and *FixedPriceLising* child classes.

Implementation

The *Listing* table was designed to be generalised to accept all *Listing* objects by creating an additional column value *type* to record the inserted listing's type of either *Auction* or *Fixed*. Secondly, the *Listing* table accepts null values in some columns, namely the *start_time* and *end_time* if the record is of *Fixed* type and *stock* if it is of *Auction* type.

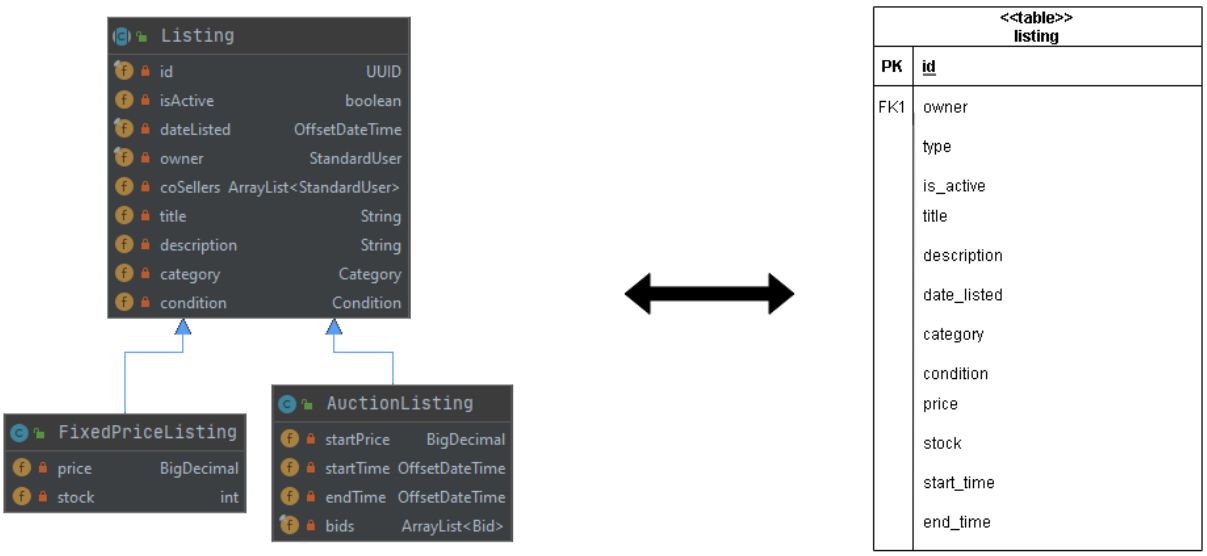

Figure 3.2.1. Mapping all fields from the *Listing* hierarchy to the *Listing* table

The key benefit of using single table inheritance for this project was its simplicity in implementation. There is only one table that represents that whole *Listing* inheritance hierarchy which contains all of their inherited attributes and only one additional field to record its type. It also simplified the implementation of the *ListingMapper* where the use of complicated SQL joins are not required to fetch *Listing* objects from the database.

Justification

Single table inheritance was used in this project to enforce the use of a single *Listing* foreign key in the *Order* database table. As demonstrated in the figure below (**figure 3.3.2**), class table inheritance offers similar functionality however the team wanted to avoid the use of table joins to fetch data from the database into memory. Which may cause memory inefficiencies if many records are kept in the database.

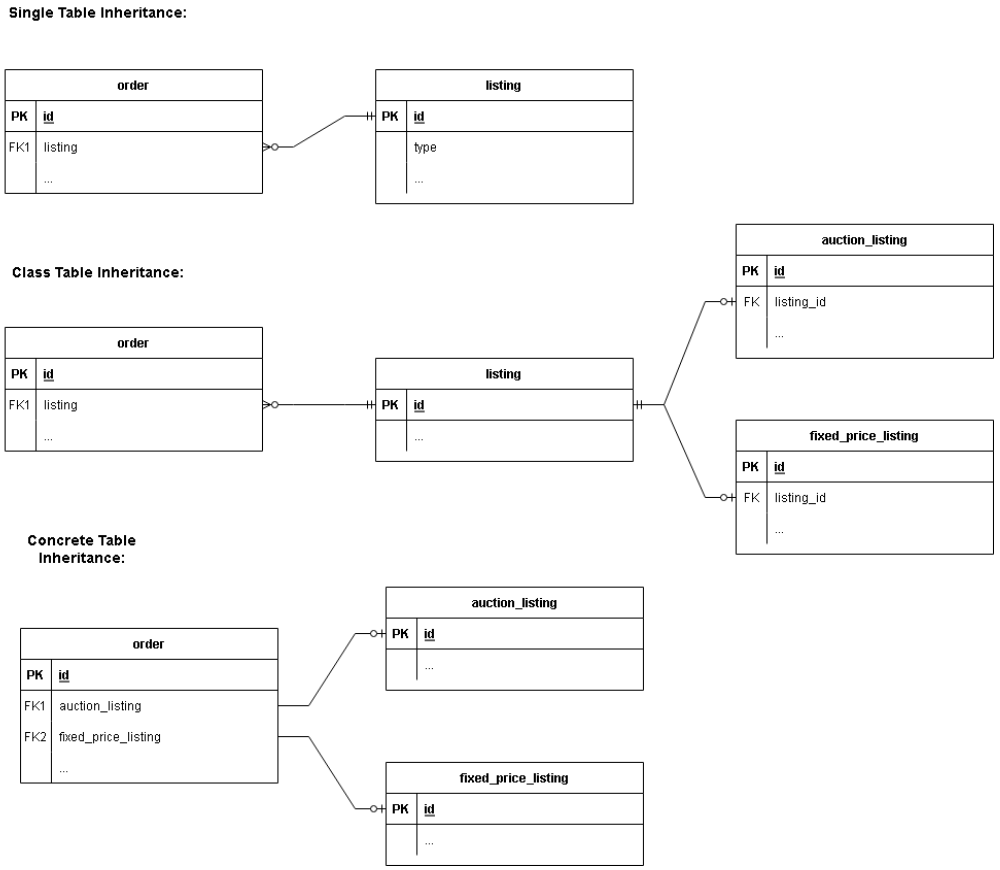

Figure 3.3.2. Team decision making on type of inheritance pattern

3.4. Domain model

Domain model is an object-oriented model of the domain which contains the model of the business domain and its corresponding behaviour with the use of object methods. The benefits of using this pattern are extensibility, managing domain complexity and re-usability. The corresponding data and behaviour of the object can be extended in this model based on expanding business requirements, and some classes can be used in other applications with similar business requirements.

The team decided to use the domain model for the domain layer architecture design based on the benefit of designing a system using object-oriented principles as well as using other design patterns that complement this model such as data mappers. This model allows more direct mapping from the initial domain model to code, extensibility during development and re-usability for future development.

During the design phase, the team identified the main objects, as well as similarities between objects, and based on the nature of the behaviour and data contained within each object, designed the basic domain model. The initial domain model was updated (**figure 3.4.1.)** before being converted to classes for the Agora Marketplace (**figure 3.4.2.**).

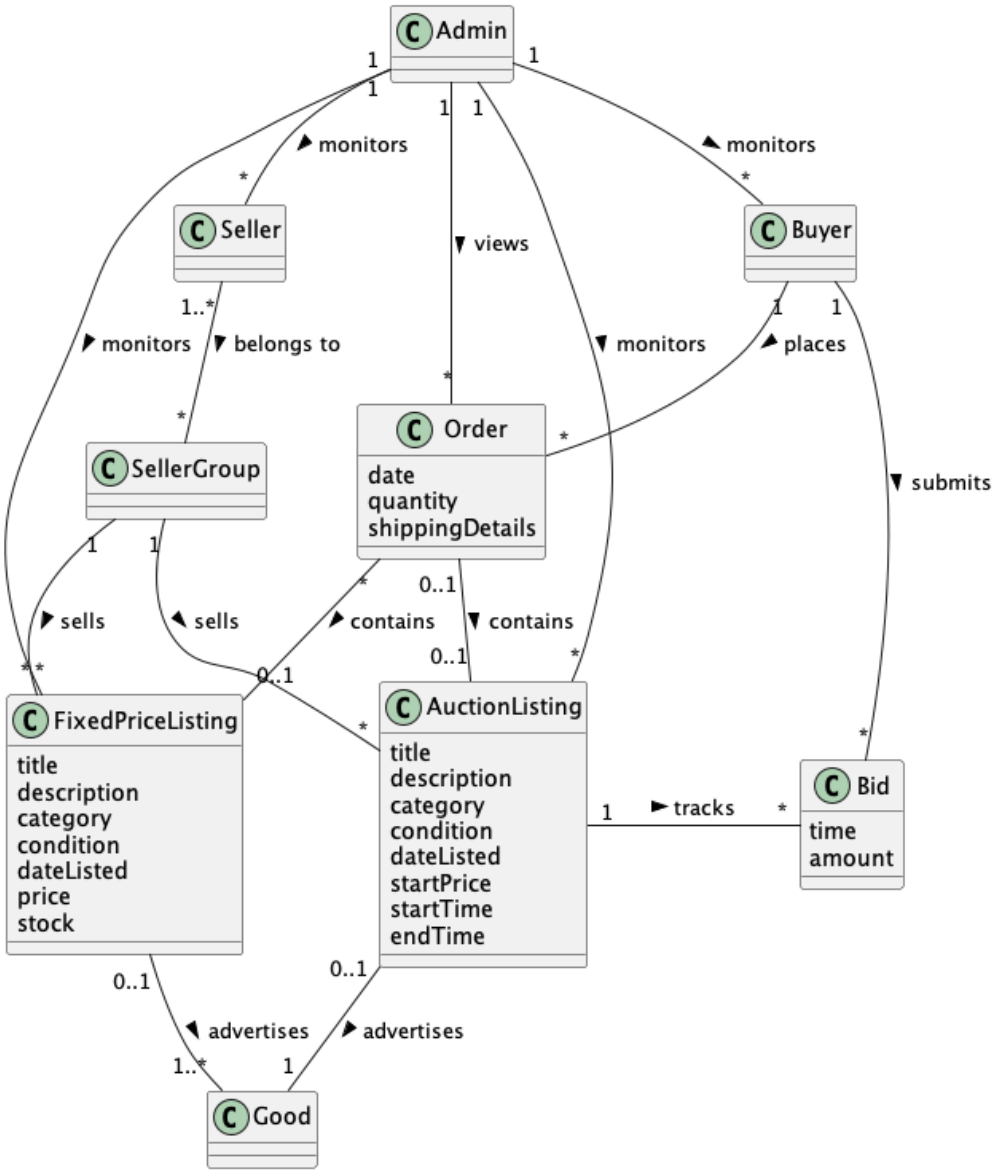

Figure 3.4.1. Domain model for Agora Marketplace

Implementation

The project team analysed the business requirements and observed multiple opportunities to implement object-oriented principles across multiple classes. For example, *StandardUser* and *Admin* classes would share many attributes with each other, therefore it was logical to create a *User* parent class with extended behaviour. *StandardUser* inherits from *User* and has access to *userName*, *passHash*, *email* and *id* as well as attributes only relevant to that class such as *ShippingDetails*. The *StandardUser* also has extended system behaviour such as making purchases, placing bids and creating listings. This hierarchical implementation of Standard*User* and *Admin* is fairly general and applicable to other domains; hence, it could easily be reused for another application with similar authentication and authorisation requirements.

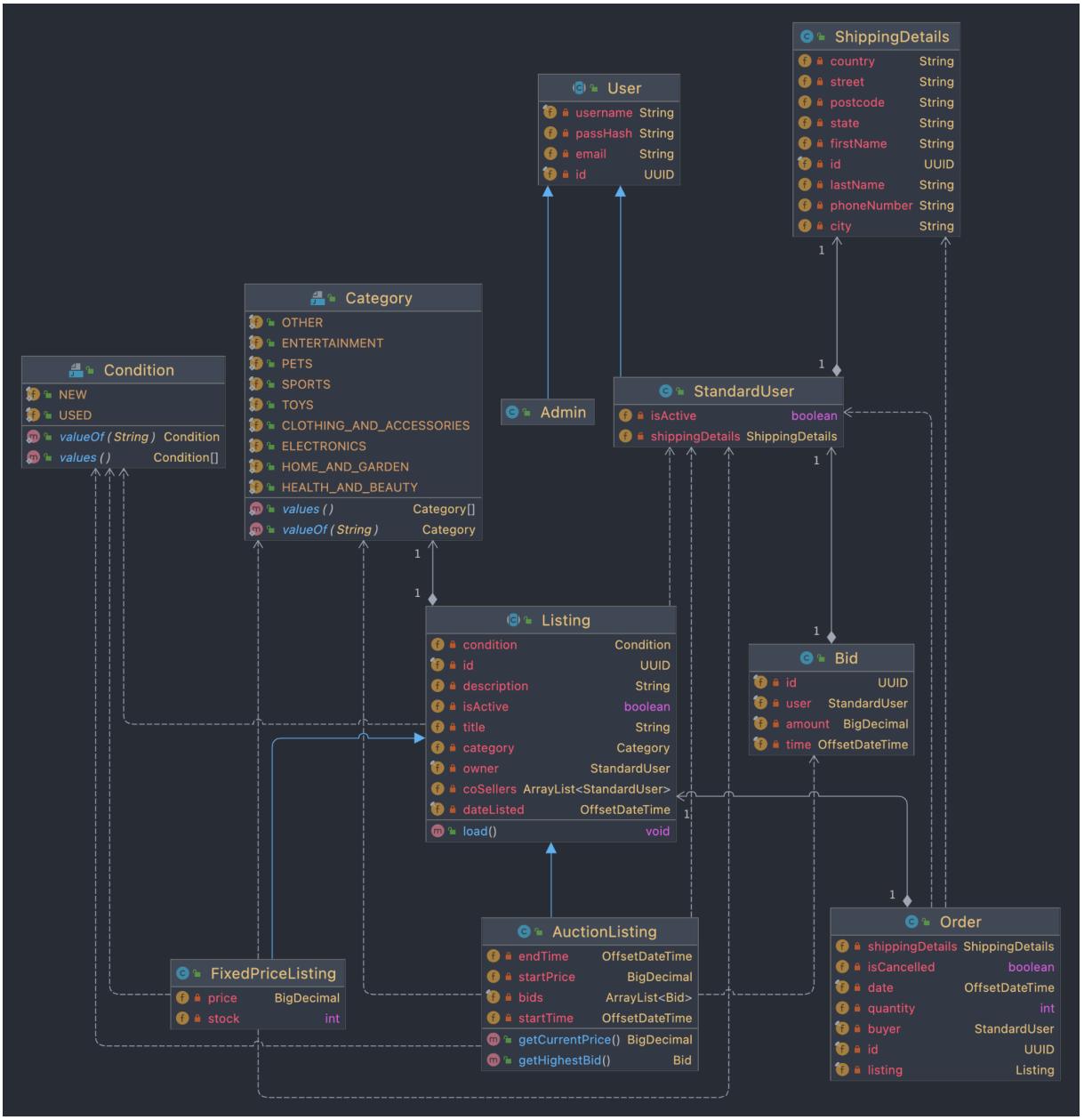

Figure 3.4.2. Class diagram for Agora Marketplace

3.5. Model View Controller (MVC)

Agora Marketplace followed the MVC architecture pattern which contains a model, view and controller

Model

The model in the Agora Marketplace system represents the data and methods of the domain. The model was divided into two major parts, domain model and data mappers. The design rationale and implementation for the data mappers and domain model are discussed in **sections 3.2** and **3.4**.

View

For the Agora Marketplace system, the team opted to implement the template view design pattern. The template view design pattern involves embedding markers, in this case for Java code, into an otherwise static HTML page. The benefit of using this pattern is it's simple to design and easy to use. As the team decided to prioritise other aspects of the system over the UI design, its relative simplicity made it an appealing choice.

To implement the view template pattern, the team decided to use JSP server page technology for developing the view, and the styling was done with Bootstrap5.

Controller

To implement the controller aspect of MVC, the team opted to use the page controller pattern. The page controller is a design pattern where a single object, called a controller, handles requests either from a specific page or for one specific action on a website. The page controller is called by the client via a RESTful API call (such as GET or POST), upon which it extracts user data provided through a form or URL parameters. The controller then follows the defined behaviour for the given method that has been called, communicating with the mappers and domain model to request data and handle the user's query. The controllers then serve the user with a resulting content page in the view. The benefits of using page controllers are the ease of understanding and simplicity. One controller has its own responsibility for the model and view, making it easy to debug during development while reducing coupling in the system.

The team decided to use the page controller based on the ease of understanding and direct correlation with front-end page development. The system was designed with each controller responsible for calling the relevant methods from the models, processing the user's query, and displaying the relevant resulting view.

Implementation

Given below (**figure 3.5.1.**) is an example where a standard user creates a fixed-price listing on the Agora Marketplace system. The *createFixedListing* controller responds to an initial GET request to display the content defined in *createListing.jsp*. When the user has entered their details and submits to the controller as a POST method, the *createFixedListing* controller receives the data from the user, upon which, if the data is valid, the controller calls the relevant mappers to create new data in the database, redirecting the user when the actions have completed.

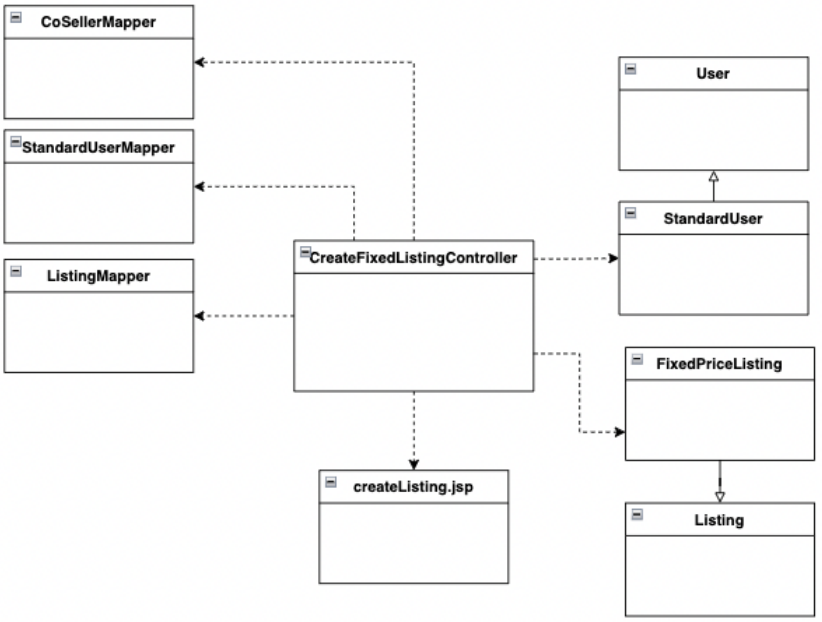

Figure 3.5.1. Model of the Behaviour of the Create Fixed Listing Controller

3.6. Identity field

Identity field pattern is the pattern that distinguishes each row in the database from others by its unique identifier key. The key can be generated by humans manually, or generated automatically by either the domain or the database. The methods for generating the key can be decided based on the requirement for the system or if the key will be needed before storing it in the database.

For this system, the team initially considered leaving the responsibility of generating and managing identifiers for each record to the database, however, upon initial implementation it was found that the solution was inefficient as the domain would need to retrieve the generated keys for each object from the database when inserting any new records. To avoid unnecessary calls to the database, the team decided to generate a random UUID (Universally Unique Identifier) in the constructor of each object before storing the data in the database. Using this approach, each object has its unique ID across the database and can be identified easily by the mapper and controllers. Although not implemented, this would also allow for a single identity map to be used.

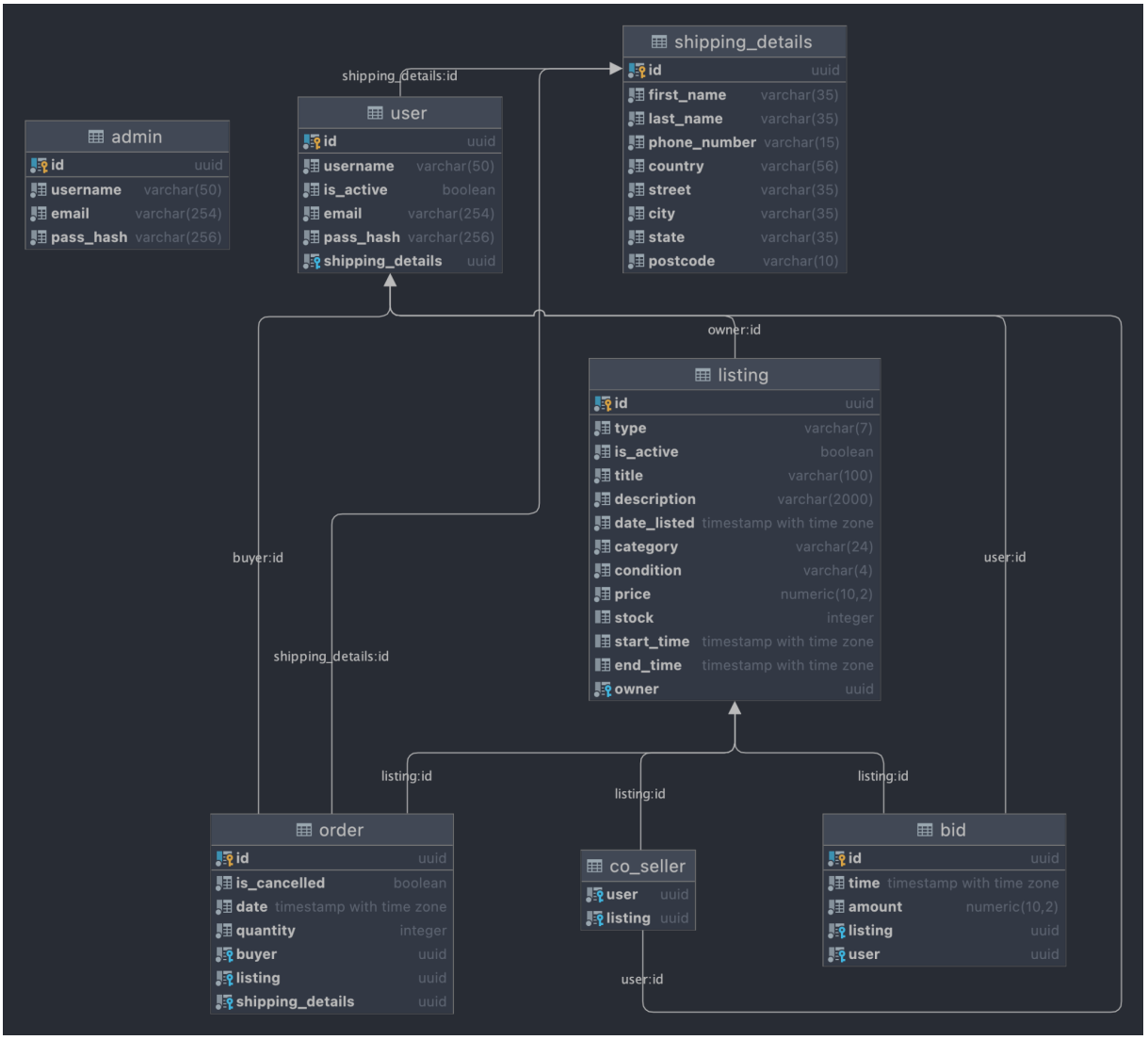

Figure 3.6.1. Entity relationship diagram (ERD)

3.7. Foreign key mapping

Foreign key mapping pattern is used to create associations between a table row record in the database to another. Using this pattern allows the system to create references between data within the database that is representative of the domain model. Foreign key mapping is dependent on the identity field, since it is used to identify the corresponding row in the referenced table.

In order to implement a database that is representative of the domain, the project team required the use of foreign key mapping to create relationships within the database. Whilst the underlying principle of foreign key mapping is simple, using keys to create references to other rows in the table, it did introduce some complexity to the system. The domain and datasource layers are now coupled together and row updates require additional care in implementing as one row could contain multiple references from many other tables. This was however somewhat mitigated with the use of the unit of work pattern that tracks changes to objects (detailed in **section 3.10**).

3.8. Association table mapping

Association table mapping is a pattern that creates associations from multiple tables. As a result an association table is created consisting of only foreign keys that supports many-to-many associations.

Similar to the foreign key, association table mapping depends on the identity field to identify specific rows in referenced tables.

The team implemented the association table mapping pattern for the *co-seller* table to enable users to be co-sellers for multiple listings, for each listing to have multiple co-sellers. In the *co-seller* table, the data only includes the foreign keys to reference the user and listing. The design decision was based on the fact that multiple users can be co-*sellers* for one listing and one user can be *co-seller* for multiple listings.

3.9. Embedded value

Embedded value is the structural design pattern that saves an object from one table into several fields in another table. This design is only applicable to one-to-one mapping relationships, with the benefit of improving the independence of the object-oriented domain model.

In the initial design phase, the team implemented the embedded value pattern for listing details. More specifically, the title, description and category were stored in a separate class in the domain model, but stored with the listing in the database. However, the implementation of the embedded value for listing details was discarded after consultation with the supervisor. It was decided this implementation would increase complexity during development and result in unnecessary extra controller logic.

3.10. Unit of Work

Unit of work is an object to relational design pattern that keeps track of changes to domain objects during a single business transaction, and when the transaction is committed, writes all changes to the database in one go. More specifically, it maintains new, dirty and deleted lists (and sometimes a clean list, although it is not necessary) and when committed, inserts all items in the new list into the database, updates all items in the dirty list, and deletes all items in the deleted list. There are a number of benefits to using unit of work, one of the most obvious is it provides a simple way to keep track of changes and is fairly trivial to implement. For transactions that only result in a small number of changes being made to a large set of objects, it significantly improves efficiency, as only the objects with changes are updated, rather than the entire set. Furthermore, keeping track of all changes made by a thread in a single place improves the cohesiveness of the design. There are two different methods for updating the lists within unit of work - caller registration, where the calling code (e.g., controller) registers the objects with the unit of work, and object registration, where the objects register themselves with the unit of work (through constructor and setter methods). Unfortunately if a developer forgets to implement either of these methods, the resulting fault can be difficult to trace, however, the same can be said for many alternatives. One alternative to unit of work is saving any changes to the database as soon as they are made, which is fine for a single change, however, when there are multiple changes and there is the potential for concurrency problems, unit of work is a better choice.

Implementation

Unit of work was implemented for both creating new orders and modifying existing orders. Creating a new order involves updating the *Listing* (reducing the stock for fixed price listings or setting the listing as deactivated for auction listings), creating new *ShippingDetails* for the order, and finally creating a new *Order*. Similarly, updating an order can result in updates to the corresponding *Listing*, *ShippingDetails* and *Order* objects. Since there are multiple changes occurring during a single business transaction, it made sense to utilise unit of work for these use cases.

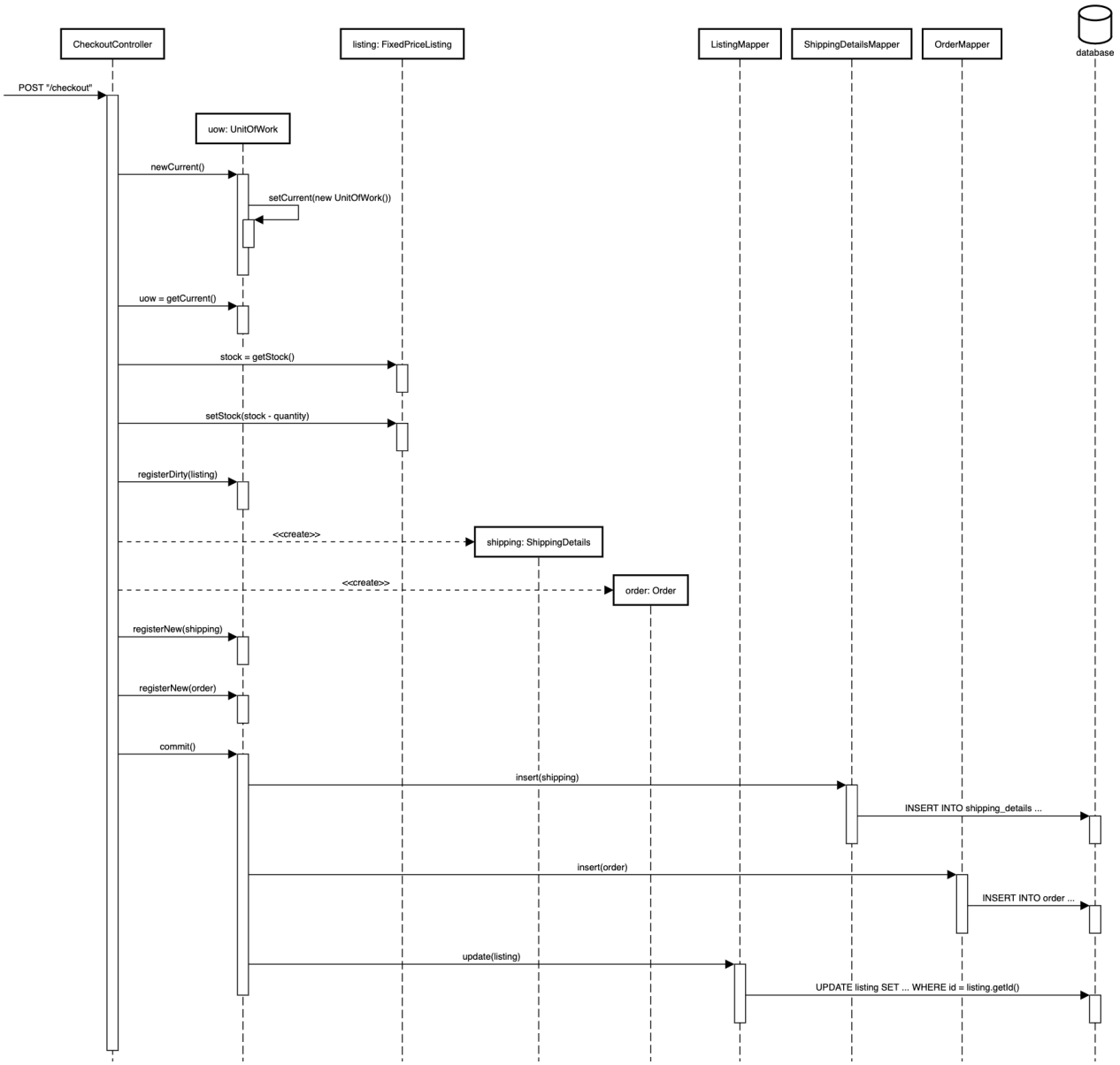

Figure 3.10.1. Sequence diagram for placing a new order for a fixed price listing (via checkout)

3.11. Lazy load

Lazy load is an object to relational design pattern that aims to reduce the amount of data being read from the database by only reading a subset of the data initially, and then loading other fields only when they are needed. The main benefit of lazy load is that it helps to improve performance of the system, especially on pages that load a large number of objects at once. Another advantage the team discovered is it can help to improve security by avoiding loading any sensitive data unless it is needed. A major drawback of lazy load is added complexity, which increases the risk of introducing bugs and can make the code more difficult to extend. Furthermore, if the domain makes use of inheritance, lazy load can introduce confusion when there are specific fields within the database that are needed to determine which type of object to create, and those fields are not loaded. There is also the potential for ripple loading and hence the trade off between loading the entire collection from a single query or making successive calls to the database needs to be carefully evaluated when deciding whether or not to use lazy load. There are four main ways the lazy load pattern can be implemented - these are lazy initialization, ghost, virtual proxy and value holder. The team utilised both lazy initialization and ghost within the Agora Marketplace.

Implementation

Listing

Ghost is implemented in the *Listing* class to improve performance on pages that load multiple listings at once. These include the home page where users can view all active listings and search for specific listings, as well as the page where sellers can view their own listings. Only the data required for the list of results is loaded initially and the remainder of the data is loaded when the user clicks on a specific listing to view the listing details.

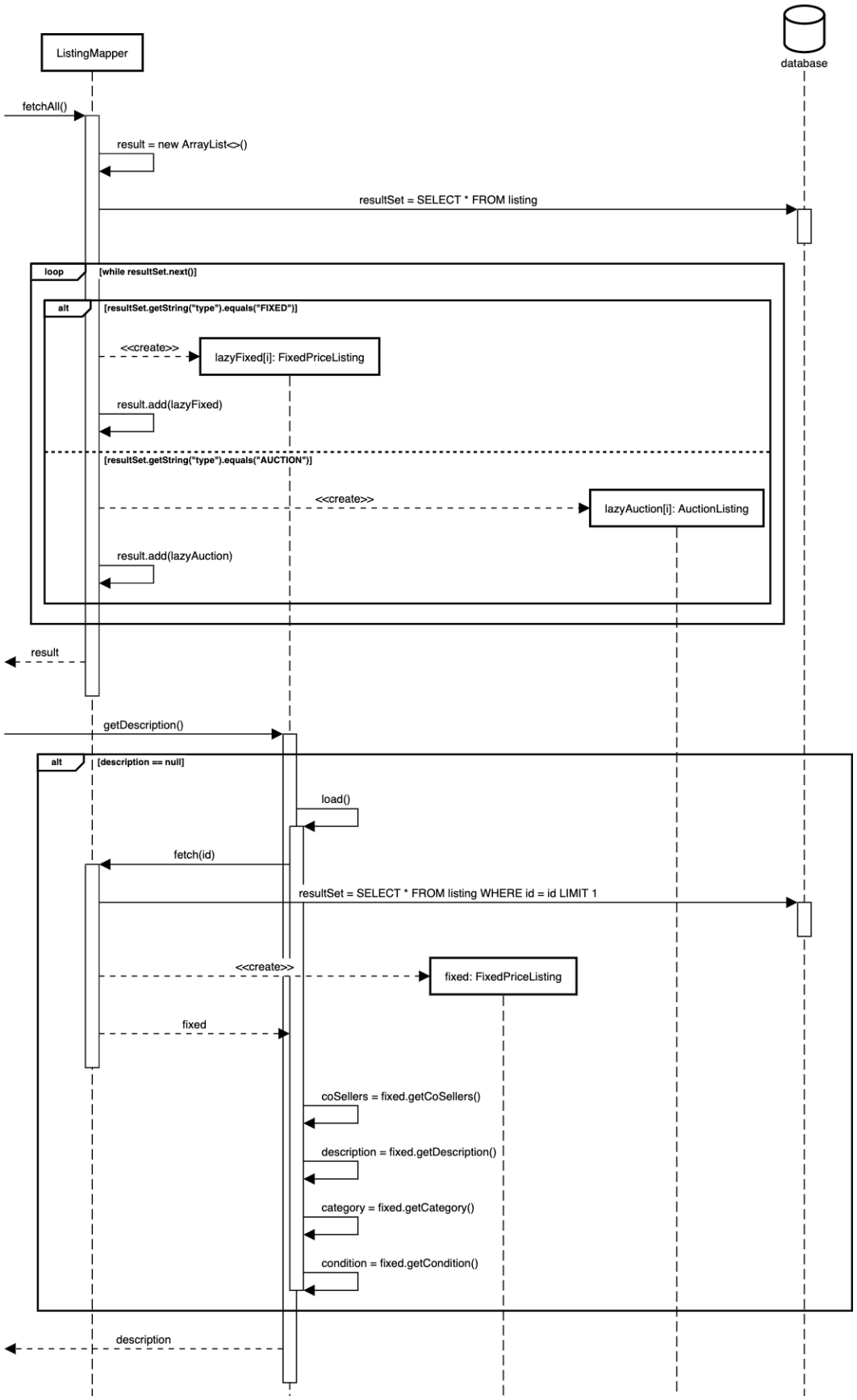

Figure 3.11.1. Sequence diagram for ghost lazy load in *Listing* class

Order

Similar to the justification for using lazy load in the *Listing* class, lazy initialization was implemented in the *Order* class to improve performance on pages that load multiple orders at once. These include the view all orders page, accessible by the admin, and view purchases and sales, accessible by standard users. Since only the *shippingDetails* are lazy loaded, lazy initialization was implemented by loading the data in the *getShippingDetails* method.

User

The *shippingDetails* object is also lazy loaded in the *StandardUser* class since the details are only needed when a user edits their shipping details, or when placing an order. They are not needed for various other use cases that load user objects such as the admin viewing all users on the system and creating new listings (which assigns an owner and co-sellers). Furthermore, to improve security a decision was made to lazy load the *passHash* in the *User* class (which *StandardUser* inherits from), so that it is only retrieved when a user signs in or changes their password. Although there are multiple class attributes being lazy loaded, lazy initialization was selected over ghost so that loading *shippingDetails* does not load *passHash* at the same time, and vice versa.

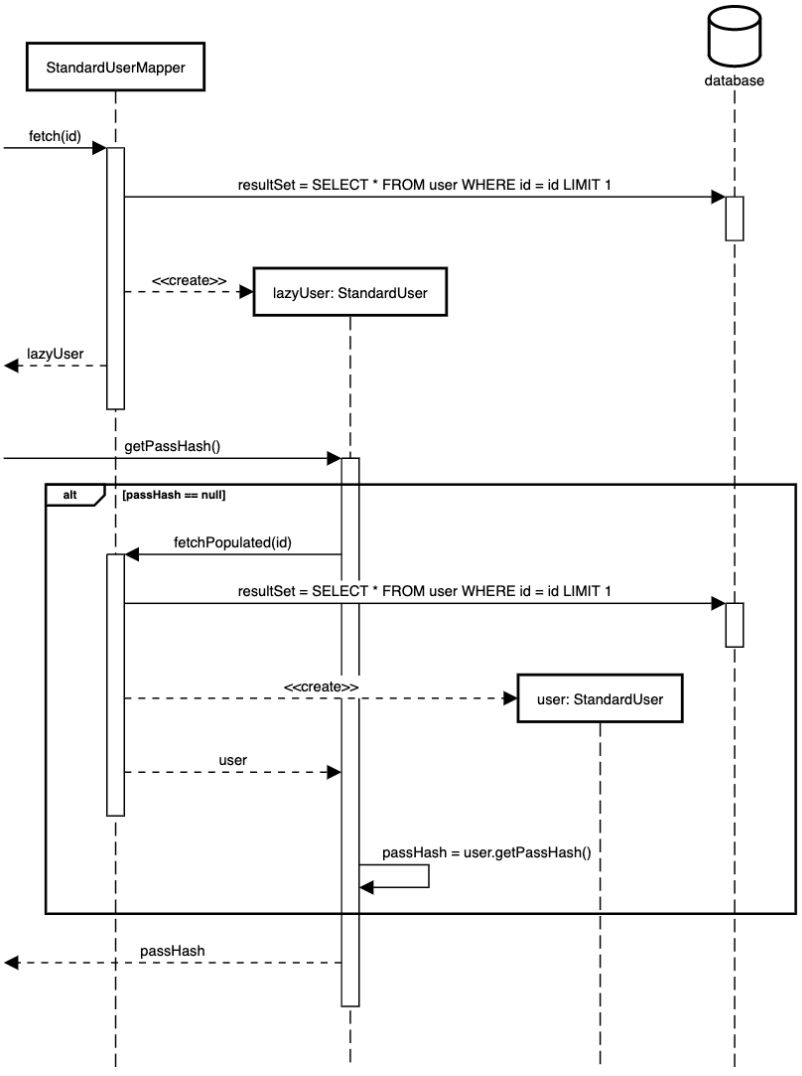

Figure 3.11.2. Sequence diagram for lazy initialization in *User* class

3.12. Authentication and Authorisation

Because this system operates according to a flow of users, with different types and privileges, it is critical that the system has an appropriate level of authentication and authorisation used throughout in order to provide the necessary security. Authentication is the process of ensuring that a user of the system is who they say they are, while authorisation involves restricting access to particular resources or functionality based on the user's role or status.

Implementation

Authentication was implemented by adopting the authentication enforcer pattern. In this pattern, authentication of a given user is delegated to a unique object which specialises in the authentication. The *AuthenticationEnforcer* is called by the page controllers in almost all circumstances, except where authentication is not necessary.

In the implementation of the Agora Marketplace system, during login, the enforcer uses the provided email to fetch a matching user from the database. The enforcer also hashes the user's provided password, using the Pbkdf2 Password Encoder included as part of the Spring Security Framework. The password hash is then compared against the one retrieved from the database (included in the user object), and if it matches, the user's UUID is stored in the *SessionContext* object for that user. As the *SessionContext* is located with the server, the client user instead receives another identifier for their matching *SessionContext* object. This identifier is then used to authenticate the user in future requests.

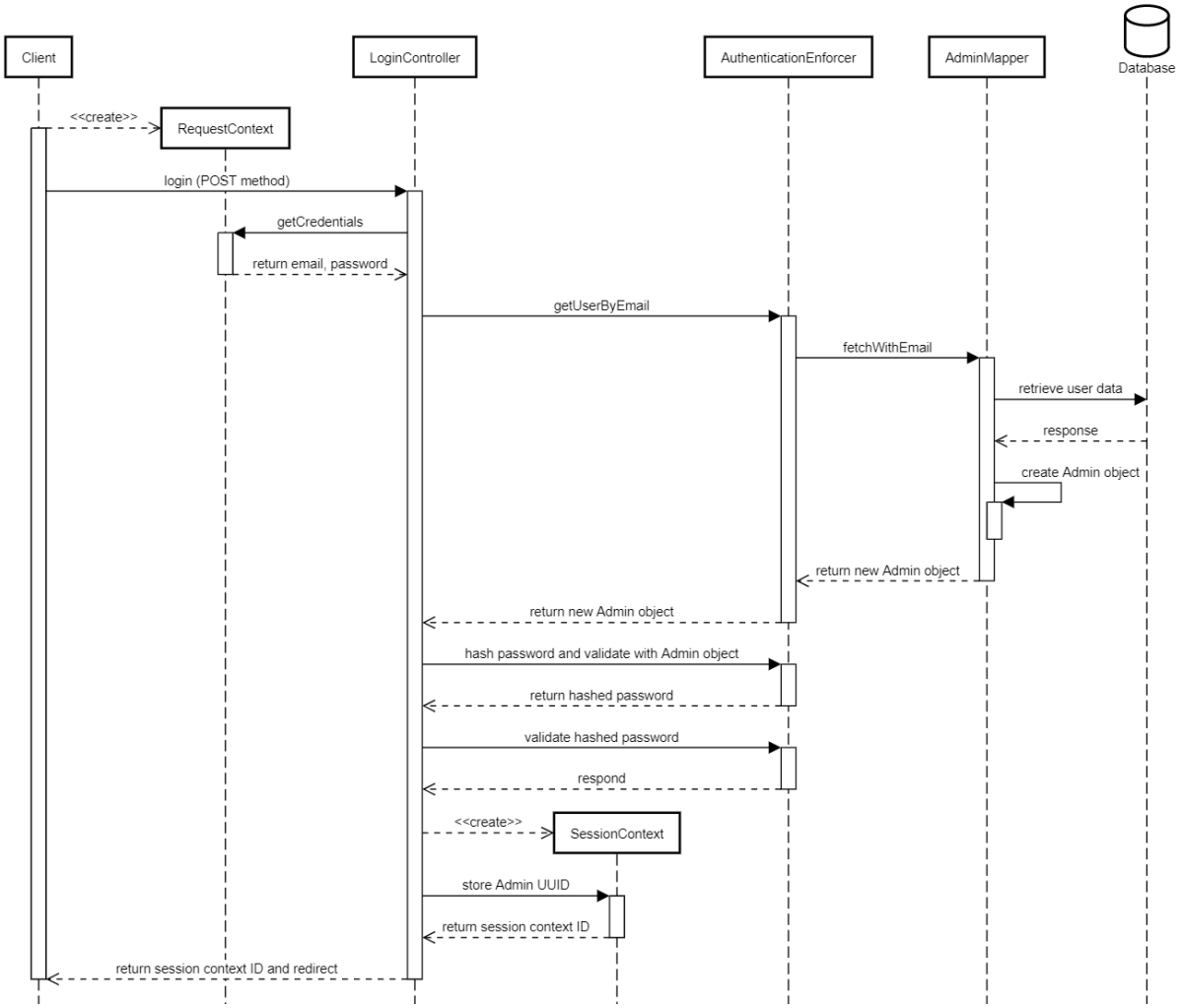

Figure 3.12.1 Login Authentication Sequence Diagram of the Admin User

Once the user has been logged in, and their UUID stored with the *SessionContext*, they can then make authenticated requests to the server. When making a request to a page requiring authentication, their stored UUID is used to retrieve directly the user object from the database, which is then used for authorisation purposes, or to handle the user's specific request. Unlike with authentication, an existing pattern was not closely adhered to; instead, authorisation is handled on a controller-by-controller basis.

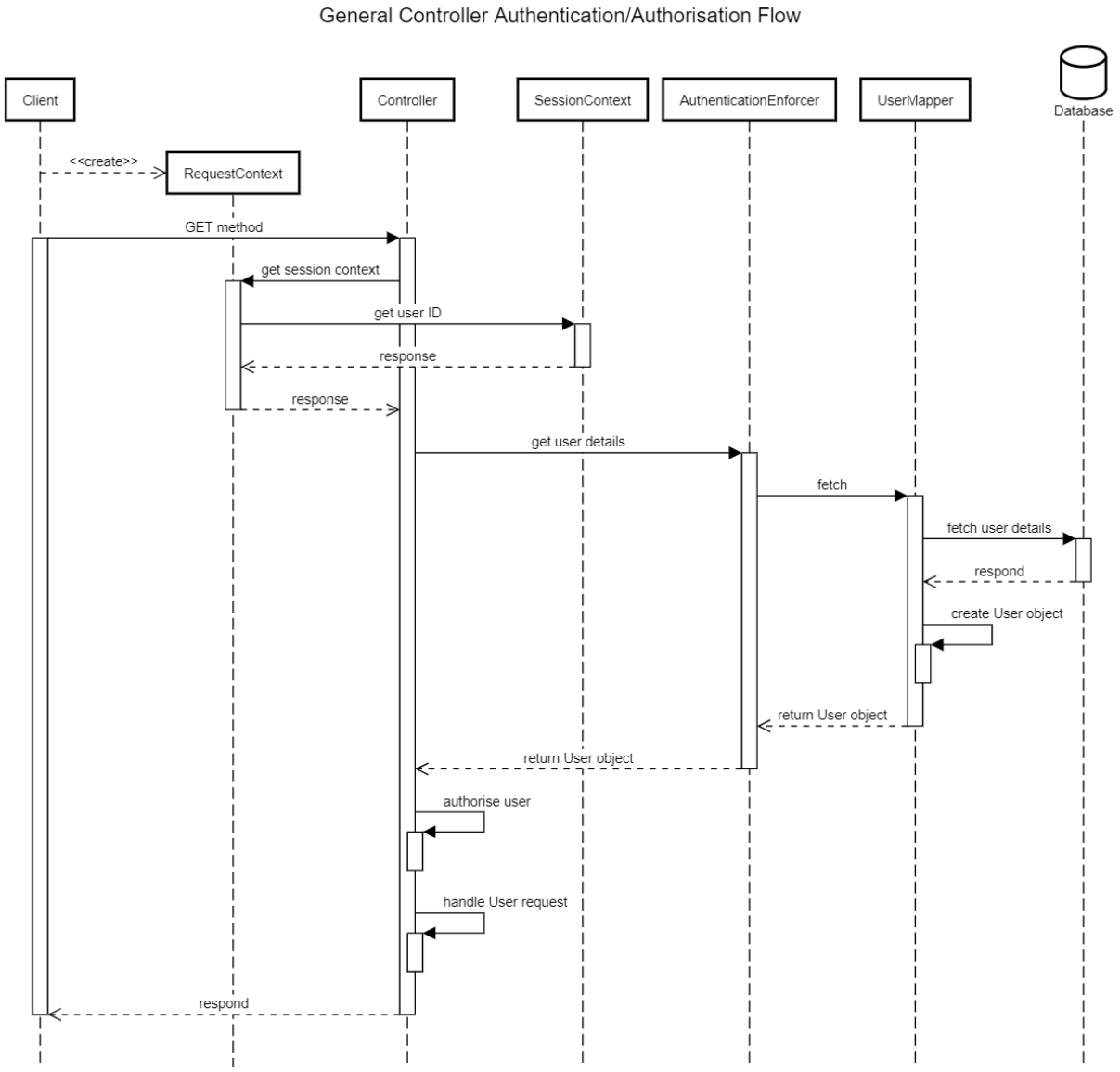

Figure 3.12.2. Sequence Diagram for General High-Level Authentication and Authorisation Flow

Justification

Due to the relatively simple nature of the system, an advanced security system was not considered necessary. The authentication enforcer pattern was chosen due to meeting the requirements of basic page authentication, where the same authentication system could easily be used in a global context.

As for authorisation, no pattern was specifically implemented due to the highly varying nature of resource and functionality privileges. Because, in many circumstances, pages and methods could be accessed in a nuanced manner, it was decided that delegation would not necessarily save on reducing complexity. For example, accessing the page to edit a listing could be performed by either the owning

user or any co-sellers, but the level of privilege they were given differed often significantly; hence, it made sense while developing to handle such circumstances on a case-by-case basis.

4. Source Code Directories Structure

src
|-------- *main*
|--------*java*
|--------*controller*
|--------Controllers
|--------*datasource*
|--------Data Mappers
|--------*domain*
|-------- Domain Model
|--------*util*
|-------- Unit of Work
|--------*webapp*
|--------Views

5. Libraries and *Frameworks*

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

Library / *Framework*	**Reason**	**Version**	**Environment**
Spring Security	Used for password hashing	5.4.0	Production and Development
Bootstrap	Used for styling the website	5	Production and Development

6. Development Environment

The development environment will be formed by:

- IntelliJ Professional Edition Version 2022.2
- Apache Tomcat Version 9
- PostgreSQL Version 14.5
- Git Version 2.37.3
- Heroku

7. References

The University of Melbourne. (2022). *Software Design and Architecture Subject Notes for SWEN90007*.