Submission 2 Report

MS-Quokka

SWEN90007 SM2 2022 Project

Name	Student ID	Unimelb Username	Github Username	Email
Edward Bond	994614	erbond	erbond994614	erbond@student.unimelb.edu.au
Haoxin Li	911806	haoxinl1	haoxin-li	haoxinl1@student.unimelb.edu.au
Ray Chen	977720	juitengc	rayjtchen	juitengc@student.unimelb.edu.au
Yifei Yu	945753	yifeiy9	Danielyuuuuu	yifeiy9@student.unimelb.edu.au

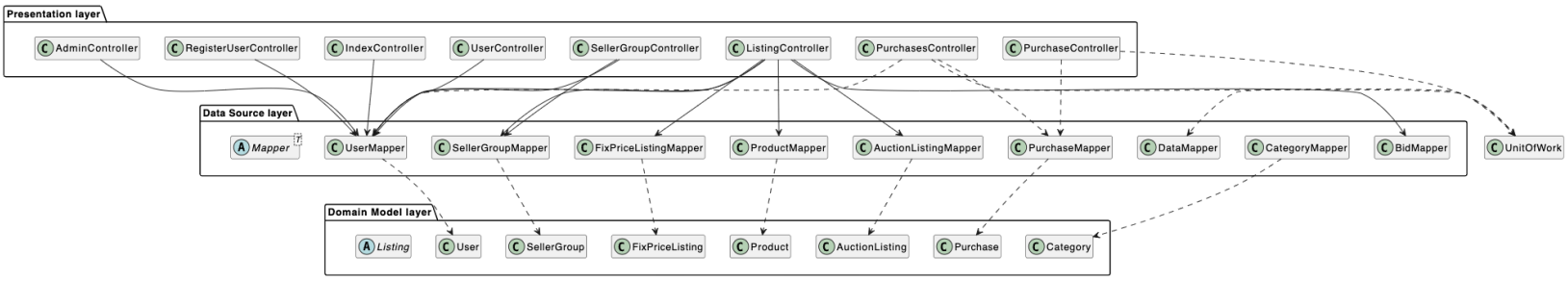
Table of Content

The Class Diagram

Due to the large number of classes that we have, we have split the class diagrams into three sections for readability.

Section 1: Relationships between classes in different layers.
In the diagram below, we can observe that:

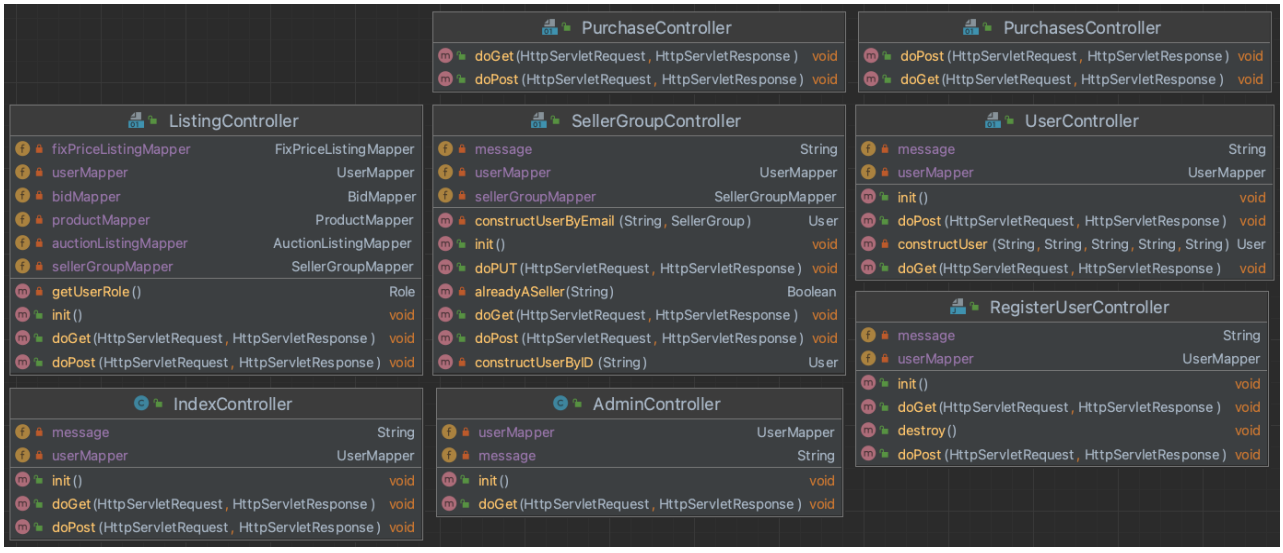
- The Controller classes either have a Mapper class as its attribute or use a Mapper class (e.g. as an input parameter in a function).
- The Mapper classes use the classes in the Domain Model layer.
- All the concrete Mapper classes inherit from the abstract Mapper class. For readability reasons, we choose to not draw the connection between each concrete Mapper and the abstract one.
- Both the AuctionListing as well as FixPriceListing inherit from the abstract Listing class. For readability purposes, we have omitted the inheritance relationship in this view as well.



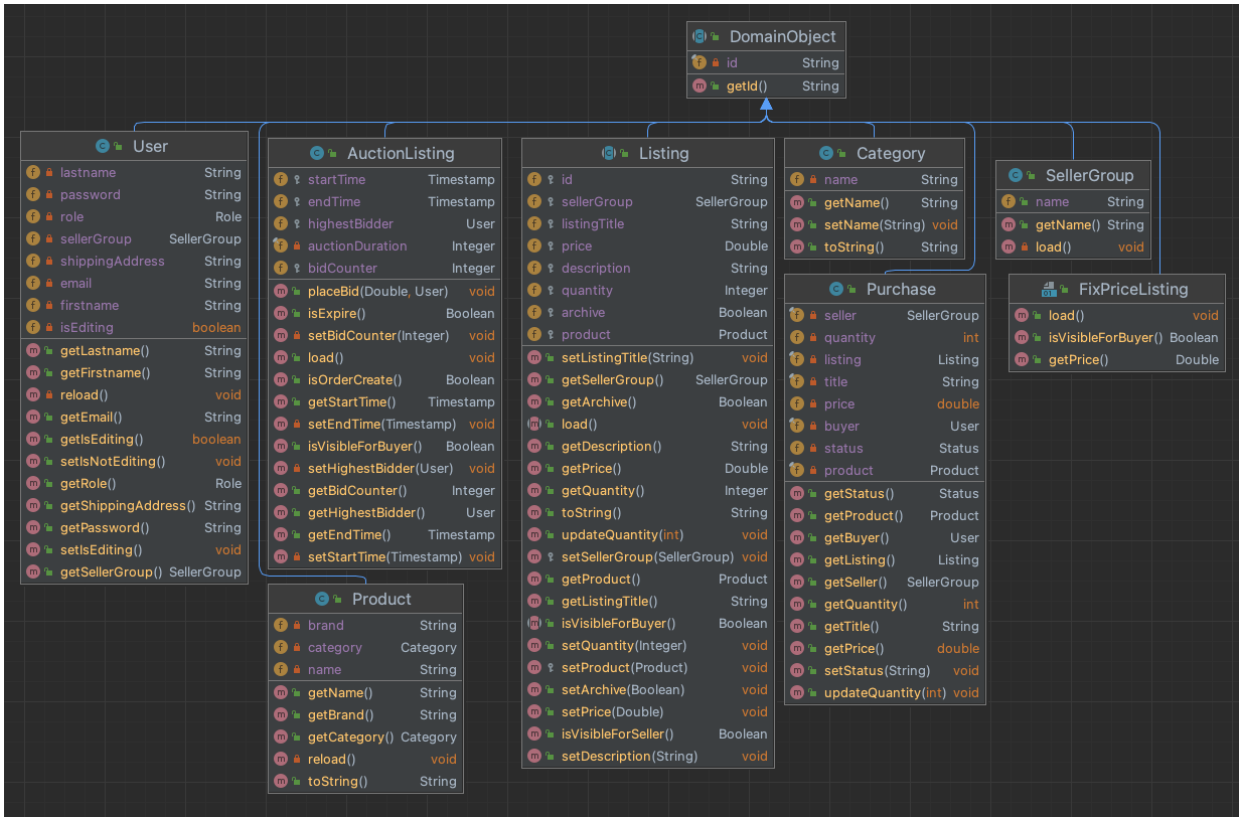
Section 2: Relationships between classes in the domain layer
See **Figure 1** in the domain model pattern.

Section 3: The specific implementation of individual classes (except for the classes in the domain layer as they have been covered in Section 2)
Note: Please ignore the blue line in the figures below

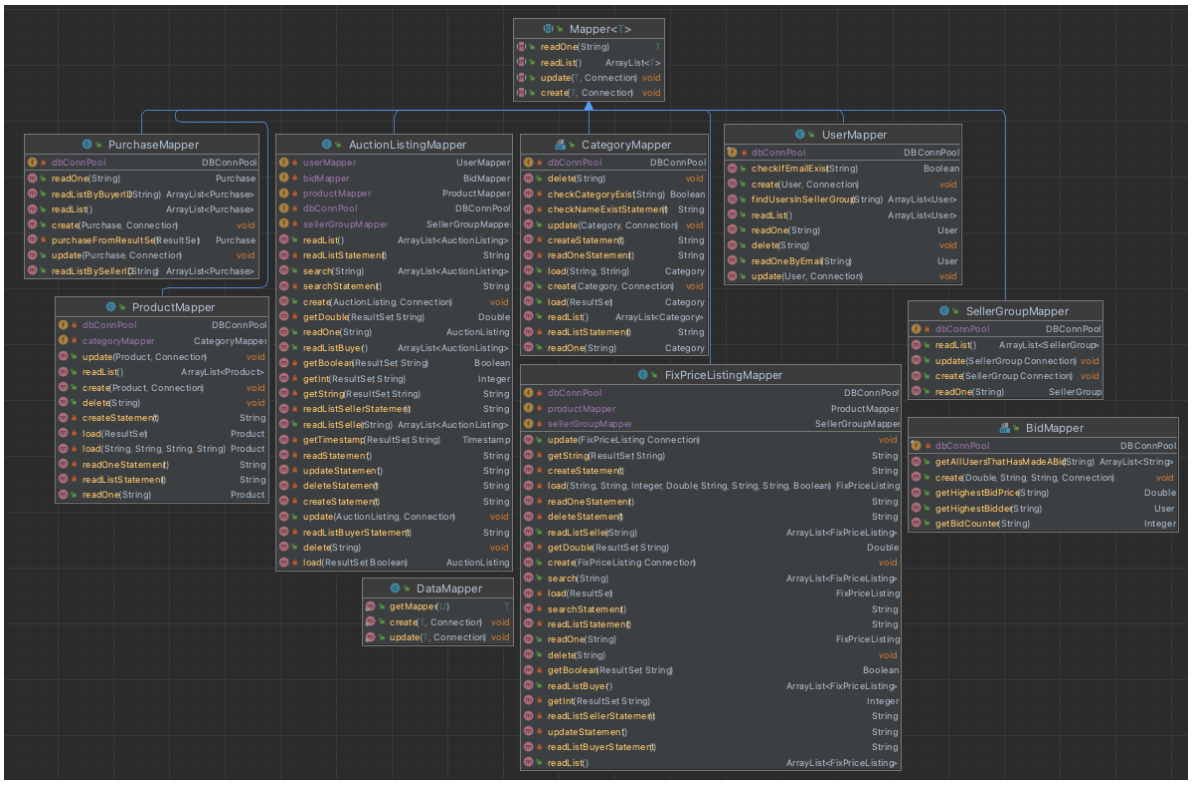
Controller



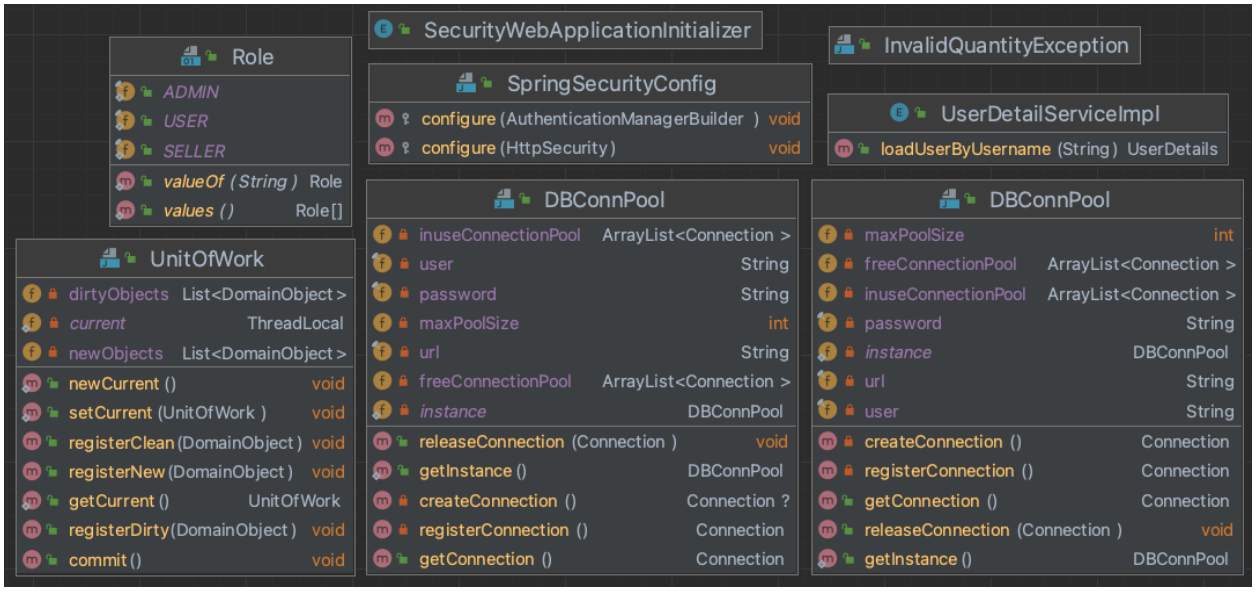
Domain



Mapper



Other



Pattern: Domain Model

We used an object-oriented approach to construct a model for the business domain in the MarketPlace System. We abstracted the objects in the business domain, such as a buyer, a seller, an order, a listing, into Java classes that have their associated attributes and methods.

Figure 1 illustrates how the abstraction is achieved. For example, buyer, seller and admin are abstracted into a User object with attributes of id, email, firstname, and lastname, and the user type is captured by an enum called Role (BUYER, SELLER and ADMIN). The company that uses the MarketPlace System to sell goods are modelled as SellerGroup objects with name and id, and which sellers work for this company is captured by the SellerGroup attribute in the User object. Selling items on the website is abstracted into a Listing object, and the different approaches that a company can sell items is abstracted into two children classes, FixPriceListing and AuctionListing, that inherit from the Listing. An order that has been placed by a buyer is captured as a Purchase class.

The interconnected objects can interact with each other through methods. As we adopted the Model View Controller pattern for the system, the Controllers are responsible for the interaction between objects, rather than the Model (i.e. Java classes). Therefore most of the classes shown in Figure 1 do not have methods other than getters, setters and constructors.

Note that there is a one-to-many relationship drawn between AuctionListing and a User. This is to reflect that we have chosen to add a highestBidder field (an User object) in the AuctionListing object. In our design, AuctionListing and User have a many-to-many relationship where an

auction can be bid by many users, and a user can place many bids on many auctions. We used the Association Table Mapping pattern to model this many-to-many relationship. The use of this pattern removes the many-to-many association relationship between the AuctionListing and User classes, therefore it is not drawn in the diagram below.

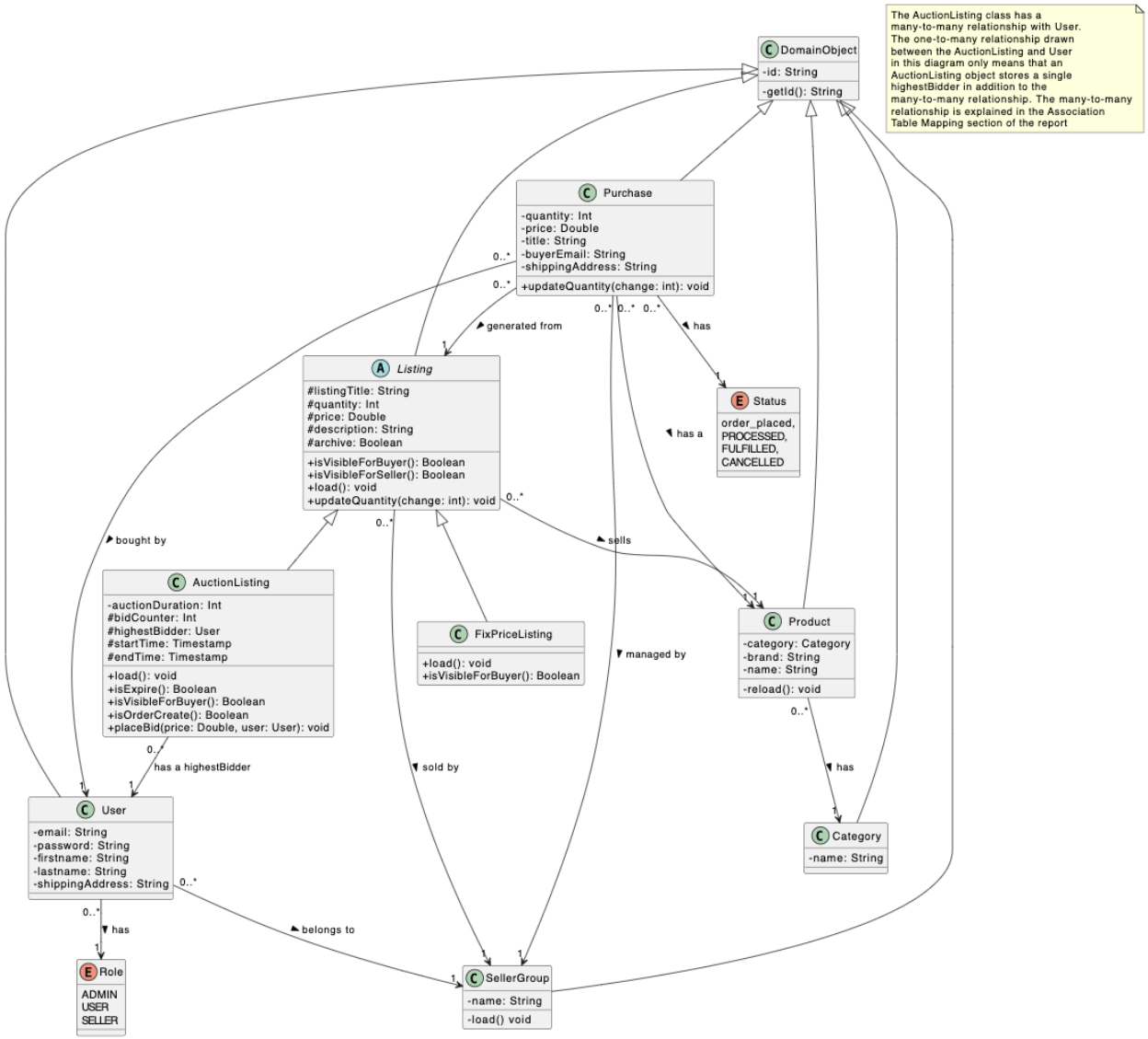


Figure 1: class diagram of the application, with only the domain layers drawn to highlight the abstraction of the business domain.

Pattern: Database Mapper

The database mapper pattern was used to enable communication between the controller and database. Its sole purpose is to fetch or update the data from the database. It does not contain

any domain logic - which is handled by the controller. This way the controller layer is separated from the database mapper layer.

There is a mapper class corresponding to each of our database tables. For example, there is a PurchaseMapper class for the domain class of Purchase, and there is a UserMapper class for the domain class of User.

The following diagram illustrates the communication between classes when a user would like to update their user profile (authentication and authorization details omitted - see the AUTH section for more details).

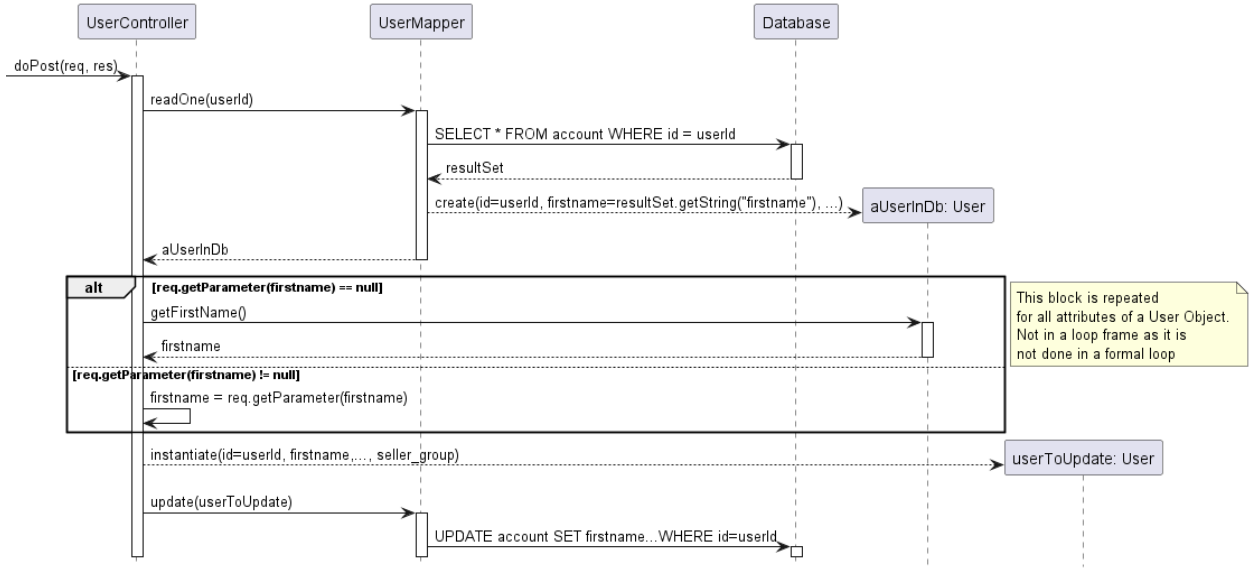


Figure 2: update user details sequence diagram

The diagram above can be divided into two steps:

1. Step 1: Instantiate a user object in memory, ready to be used as a parameter by the UserMapper to update the database.
2. Step 2: Update the database with the latest information.

Logic in Step 1

As the UserMapper implements the Mapper interface, the update method takes a User object as a parameter (for more information, see Inheritance section). This means that we need to instantiate a User object which represents the latest record of the Mapper class.

However, the challenge comes as the client might not update all fields on the frontend. For example, if the client only updates their password and leaves other fields empty, we won't know the values of the unchanged attributes (they will be empty). Therefore, we need an initial call to the database to fetch the existing information of the user, with the following steps:

1. Firstly, the frontend sends a POST request to the UserController, which then delegates the task to UserMapper to find the user with the userId (i.e. calling readOne(userId)).

2. The UserMapper will query the database, instantiate a User instance (i.e. aUserInDb) and then send this object back to the UserMapper.
3. The UserMapper will then return the aUserInDb instance back to the calling class. At this point, the user object is loaded in memory, ready to be used.

Note that if all the user accounts are needed, an array of Users objects will be returned from the UserMapper (via the readList() method).

The decision to either update an attribute or keep the database record is completed in the if/else block in the diagram above. If the request contains the field to be updated (i.e. value is not equal to NULL), then the attribute value will be set to that. Otherwise the value of that field in the database will be kept.

Logic in Step 2
Finally, a new User object (userToUpdate variable) can be instantiated in memory, and is used as a parameter in UserMapper's update() method to update the record in the database.

Pattern: Lazy Load

The Lazy Load logic is implemented in the getters of the domain classes. For example, quantity is a private attribute of the Purchase object, and when the JSP receives a Purchase object in the request, to display it, getQuantity() method will be used to retrieve the quantity attribute inside the purchase object. If a requested attribute is null, then the getter method will execute a load() method to load the full objects from the database by using the PurchaseMapper.

Figure 3 shows the communication between classes when a client has requested to view all of their purchases. The allPurchases ArrayList should only have the necessary information to distinguish between purchases on the user interface (title, quantity, etc). Our approach also loads the ID of attributes that are foreign keys (e.g. for Purchase, this involves seller group and product). If the user selects a Purchase item to look into, we can use the already-loaded ID to query the database and load that object (e.g. Seller Group) in memory.

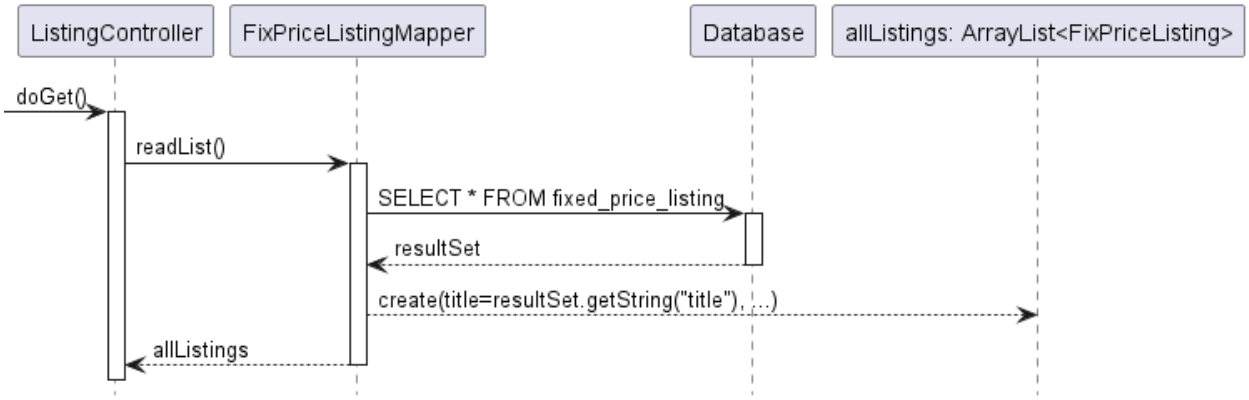


Figure 3: Client requested to view all Purchases that they are authorised to view.

The figure 4 below gives an example of how the lazy load method is triggered when additional information is needed. If the client is in the listings page and decides to click onto a fixed price listing the system will run the isVisibleForBuyer method to check whether or not the listing can be viewed by the user, thus the getArchive() and getQuantity() methods in the fixed price listing object will be called. In the getters, If the system found out that the required attributes (e.g.,quantity or archive) in the listing object are null, it then calls the load method to retrieve the entire listing object from the database and load the attributes that were previously null.

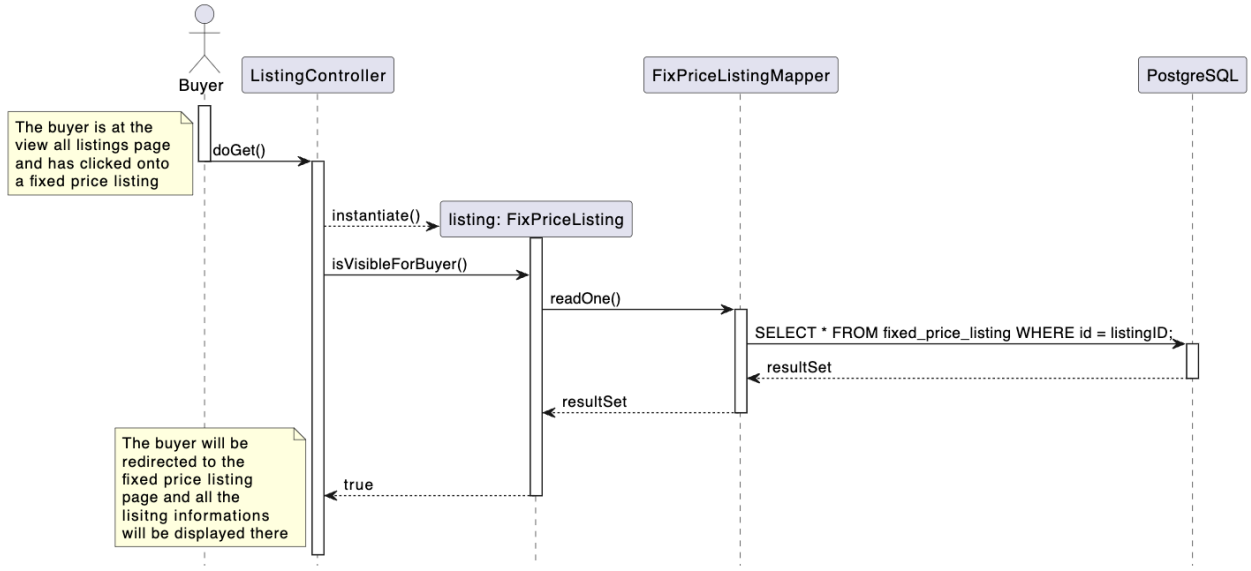


Figure 4: Buyer views a fixed price listing

Design rationale for Lazy Load

The Lazy Load pattern is used for the following use cases:

- When an admin wants to view all users
- When an admin/seller/buyer wants to see a list of listings
- When an admin/seller/buyer wants to see a list of purchases

The reason behind choosing these three use cases are:

When a user would like to see a list of objects, e.g. buyer wants to see all the listings available for sale, which listing item(s) they will click and look into is unknown. It is rather unlikely for the buyer to want to access all listings objects, therefore it is very inefficient to load each of the listing objects fully because:

1. it might involve multiple database queries to fully load one listing object into memory. For example, to fully load a listing object, a query to the Product and Seller Group tables are required. This is possible with a join, however that would break the domain model that we set up.
2. If the data is large (e.g. a video file) then it will take up lots of resources to load the video, and the client might not even need to access that resource.

Our approach allows us to load the necessary information for the user to distinguish the listings on the user interface (e.g. for fixed-price listing object, the listing_titles, quantity, price and description). Querying to other attributes in the listing object (e.g. Seller Group) is only done when the client has requested it.

Pattern: Identity Field

We have decided to use mostly meaningless keys: Every record stored in the database has a meaningless key (i.e. the id field) to distinguish between the records. The User collection is an exception - a user's email is used as a meaningful key. This is useful when a user needs to find another user or enter information about another user. For example, when an admin needs to add a user to a Seller Group, they can complete the task by typing in the user's email address. It is not user-friendly to ask the admin to enter the user's record id stored in the database, therefore we have decided to use email instead. To avoid duplication we have restricted that no two users can have the same email address when they register into our system.

We have also decided to use globally-unique keys as it would provide us with the flexibility to implement Identity Map should we wish to do so.

To generate a new key, we have decided to use the Globally Unique Identifier (GUID) approach, with the exception that we do not compute the algorithm ourselves, but delegate the task to the UUID class of Java. It can be considered as a single machine that is used to generate random, unique keys.

The sequence diagram below illustrates the creation of a Seller Group object and how the Identity Field pattern is used. Firstly, JSP sends a POST request to the SellerGroupController to create a new Seller Group. The SellerGroupController delegates the task of generating a unique ID to UUID class which returns a new randomId to be used to instantiate a SellerGroup object

(i.e. sg) in memory. Finally, the SellerGroupController delegates the Seller Group object creation to the SellerGroupMapper via the Data Mapper pattern that we implemented.

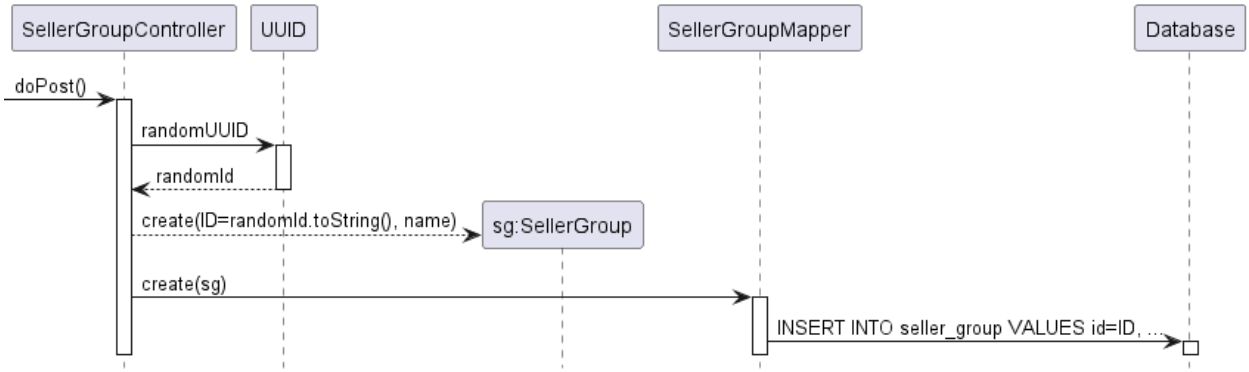


Figure 5: update SellerGroup sequence diagram

Pattern: Foreign Key Mapping

Our system involves objects that contain references to other objects. We use a combination of the Foreigh Key Mapping pattern as well as the Identity Field pattern to capture this information, and handle one-to-many relationships between domain objects.

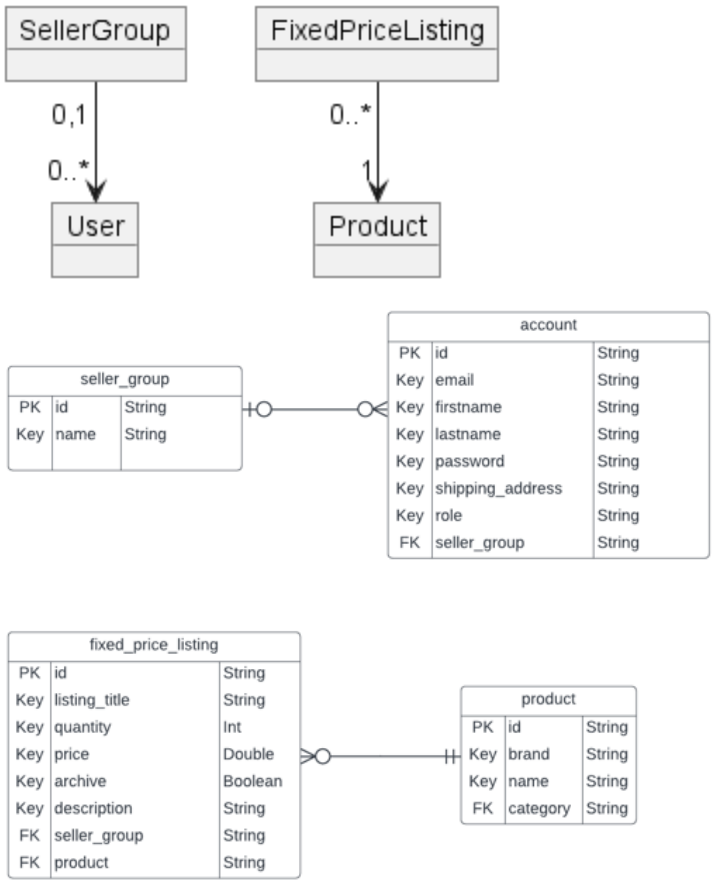


Figure 6 (top): an excerpt from the domain model
Figure 7 (bottom): An excerpt from the ER diagram.

For example, according to Figure 6, a SellerGroup may have zero or many users (i.e. Sellers) associated with it. In situations like this, where an object A (SellerGroup) refers to a collection of object B (Sellers), we follow the Foreign Key Mapping pattern, and place a foreign key referring to the SellerGroup in the collection of User objects instead. Notice that in Figure 7, the User object contains a foreign key which is the primary key of the SellerGroup table.

As part of the SellerGroup management (i.e. admin needs to be able to add/remove users from/to SellerGroups), it is required to display all the users who already exist in a SellerGroup. Figure 8 describes the interaction of classes for this use case. Note that the sgId attribute comes from the request in doGet().

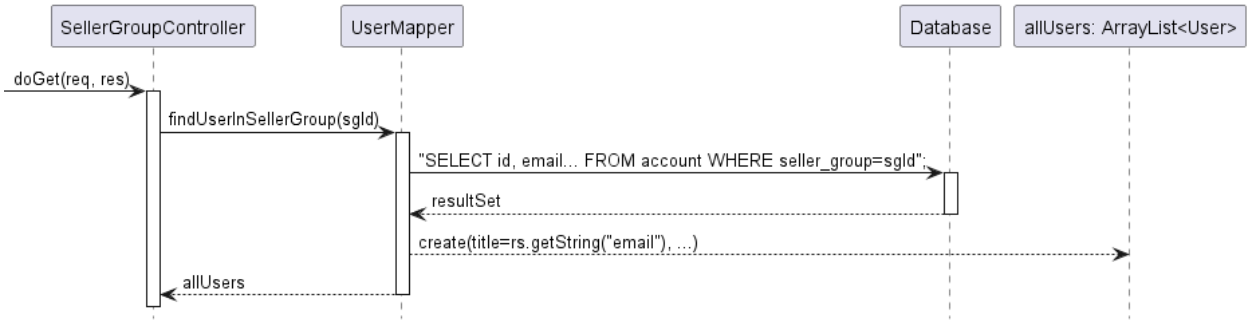


Figure 8

The situation for FixedPriceListing and Product is simpler. A FixedPriceListing must contain a Product attribute (Figure 6). When the client requested a FixedPriceListing object (e.g. a buyer browsing an available listing), the client can obtain information about what the listing is selling (i.e. the Product). In situations where an object (i.e. FixedPriceListing) refers to another object (i.e. Product), we place a foreign key, the id of the Product record, in the FixedPriceListing record.

Pattern: Embedded value

One of the possible approaches was to create a separate class for the shipping address, however we realised that there will never be a time when the shipping address needs to be loaded without a user. User and shipping address is always loaded and modified at the same time. Therefore, instead of creating a separate class for it, we have decided to store the shipping address as a string attribute in the user object. This approach simplifies our design as we do not need to make multiple queries to the database in order to instantiate one object (e.g. the Order object) in memory.

In our design, we make the assumption that a user can have only one shipping address. It is true that a shipping address can belong to multiple users. Therefore, this is not a one-to-one relationship. However, the key deciding factor is that the shipping address information is always loaded and updated with the rest of the User object. Therefore we decided to embed the ShippingAddress object (see Figure 10) as part of the User object. To simplify it further (due to time constraints), we have decided to use one attribute to capture the entire shipping address as a string (see Figure 9).

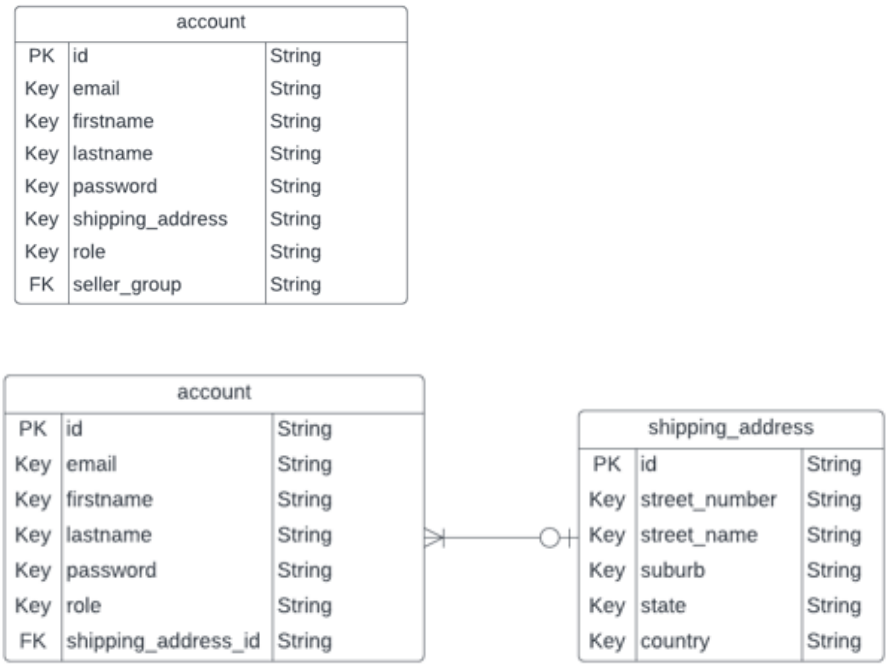


Figure 9 (top): An excerpt of the Entity Relationship diagram (actual implementation)
Figure 10 (bottom): A possible, alternative design that treats the shipping address as a separate class of its own.

Pattern: Concrete Table Inheritance

To map inheritance hierarchies to relational models, we adopted the Concrete Table Inheritance pattern which maps each of our concrete classes, FixPriceListing and AuctionListing, to a database table (as shown in Figure 11). There is a Data Mapper for FixPriceListing, and a separate Data Mapper for AuctionListing. The Listing controller decides which Data Mapper to use depending on the type of the object.

The Concrete Table inheritance pattern was a decision after we analysed each of the three choices. The Class Table Inheritance pattern was not suitable for our application as it would require storing the start_time and end_time fields in the AuctionListing table, and all other fields in the Listing table. In order to read an AuctionListing object from the database, two queries need to be performed (one to the Listing table and another one the AuctionListing table) instead of one. The extra query is unnecessary and inefficient.

There is the potential to use the Single Table Inheritance pattern, as the majority of fields between the FixPriceListing and AuctionListing objects overlap. However, we decided against that. If we use Single Table Inheritance, Listing that is sold with fixed-price will have its start_time and end_time attributes empty/null. If we want to create more fields for a Listing

object sold via auction, we create more empty fields for Listings sold with fixed-price. It is possible to result in two types of Listing objects, each has a certain set of fields empty. We believe that this will result in confusion when we try to maintain the database and take up more space in memory when the object is loaded.

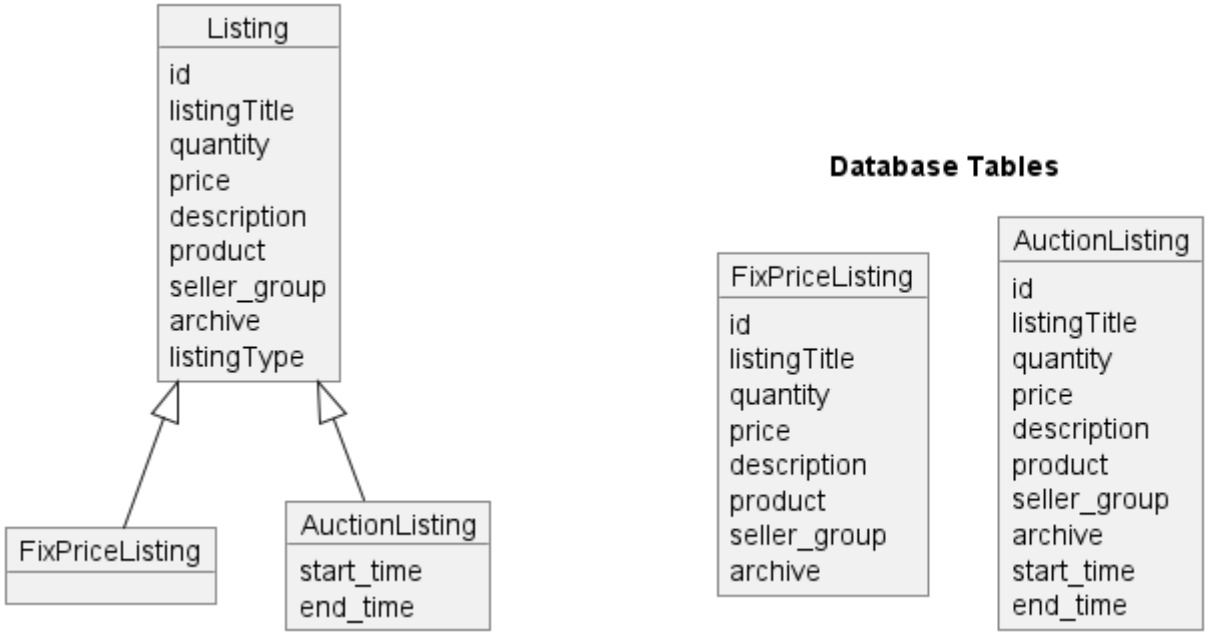


Figure 11. Mapping between domain model classes and database tables for Listings

Pattern: Association Table Mapping

The Association Table mapping design pattern has been used to handle many-to-many relationships when dealing with auctions. In our system, a user could place many bids into an auction while an auction could be participated in by many users. However, relational databases do not support many-to-many relationships; hence in order to handle such a situation, a new “bid” database table has been created and put in between the account table and auction_listing table. As shown in figure 12 below, the bid table contains a pair of foreign keys from the tables it is associated with, which turns the many-to-many relationship into one-to-many relationships between the account and bid tables as well as the bid and auction_listing tables.

Figure 12: ER-diagram for association table mapping

There are several design decisions the team has made when implementing the bid table.

1. The team decided to store the bidding price field in the bid table; As by storing the price in the bid table, the system could easily identify the highest bidder for the auction. In fact, without this key piece of information, there is no way to determine the highest bidder. Therefore we must store the bid_price attribute in the bid table.
2. Since it is possible for a user to place multiple bids on the same auction, the combination of foreign keys in the bid table (i.e. buyer, auction_listing) is no longer unique and can not serve as the primary key for the bid table. Therefore a separate id field has been used.
3. The bid table has no corresponding in-memory object. The table only serves as a record holder that connects the account table and auction_listing table together. Given a user id (i.e. the buyer field in the bid table is fixed), we can find out all the auctions that the user has participated in. On the other hand, given an auction_listing id, the admin can find a list of users who had placed a bid on this auction if they are curious about the history. We can also find out the highest bidder by querying the bid table with the known auction_listing id and locate the user who has the highest bid price. Finally, if we are interested in the total number of bids an auction receives, we can use the bid table to achieve that too - it would require a query with a given auction_listing id and counts the number of records. The interactions involving the bid table with the system are shown in the sequence diagram below.

The buyer could initiate a bid by sending a POST request to the ListingController, then after receiving the request a new bid record will be inserted into the bid table with the buyer’s id, the auction listing id and the price.

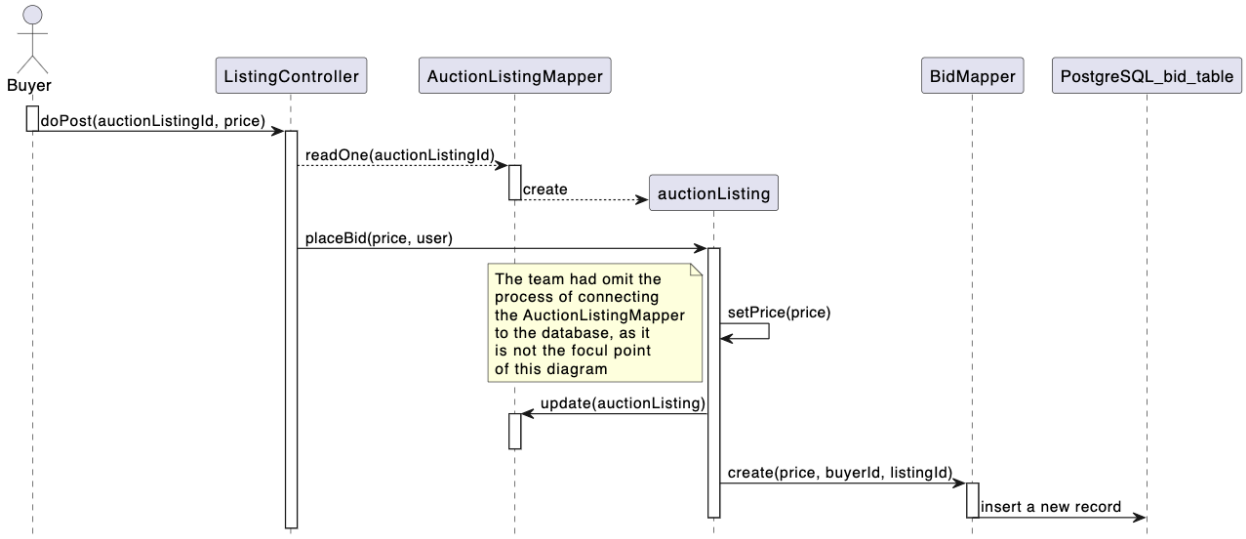


Figure 13: Classes' interactions when a buyer places a bid

The bid table contains the bidding history of all the auctions, so the system is able to retrieve the highest bidder and the number of the bids for a particular auction and display those information to the seller

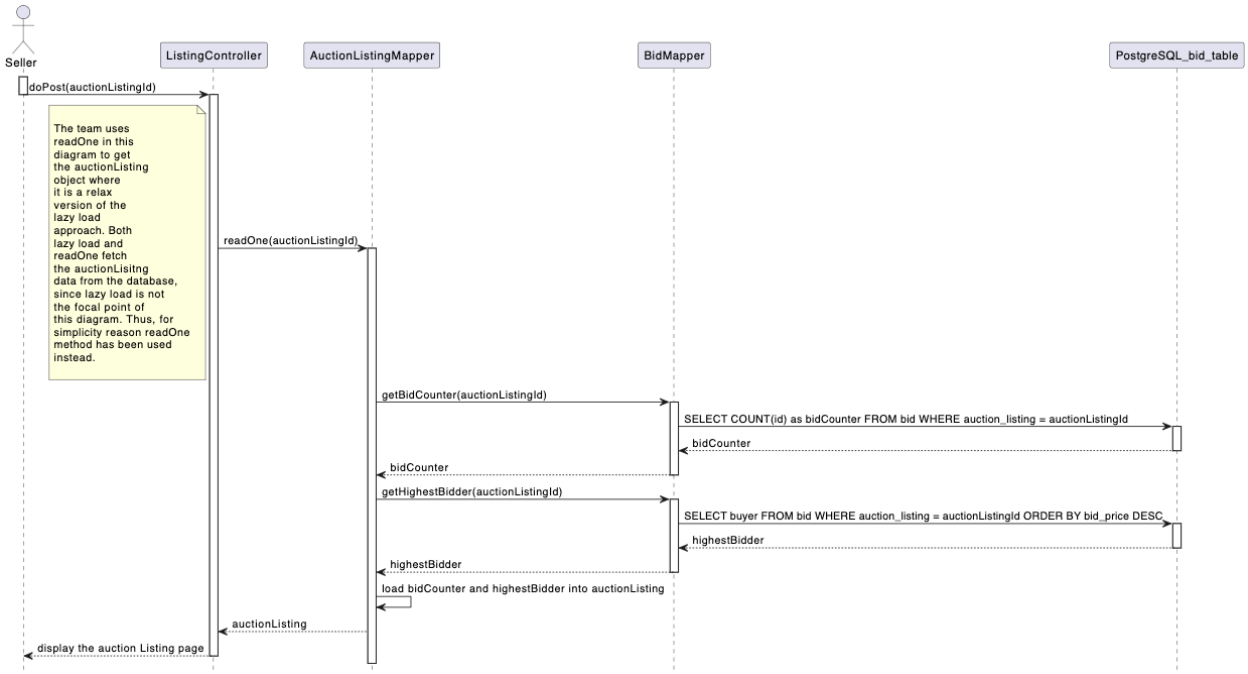


Figure 14: Classes' interactions when a buyer wants to retrieve all the information of an auction listing

Pattern: Authentication and Authorization

The Spring Security framework was used to implement authentication and authorization for our application. The purpose of authentication is to identify the individual user that is trying to gain access to the marketplace system. The purpose of authorization is to control the user's access rights to the system resources.

Figure 15 below shows the interaction between different objects when a buyer tries to login. The AuthenticationManager class is a built-in class from the Spring Security framework, which is used to authenticate the user. When a buyer enters the email and password in the login page, after clicking on the ‘login’ button, the AuthenticationManager will ask the UserDetailsService to fetch the user credentials in the database via the UserMapper. After receiving the user credentials, the AuthenticationManager will encode the password that was entered by the user, and compare it with the encoded user credential from the database (which is contained in the userDetails). If it is a match, the AuthenticationManager will return ‘authentication==true’ and the user will be redirected to the corresponding webpages. If the password entered is incorrect, the AuthenticationManager will throw an AuthenticationException, and the error message will appear on the login page.

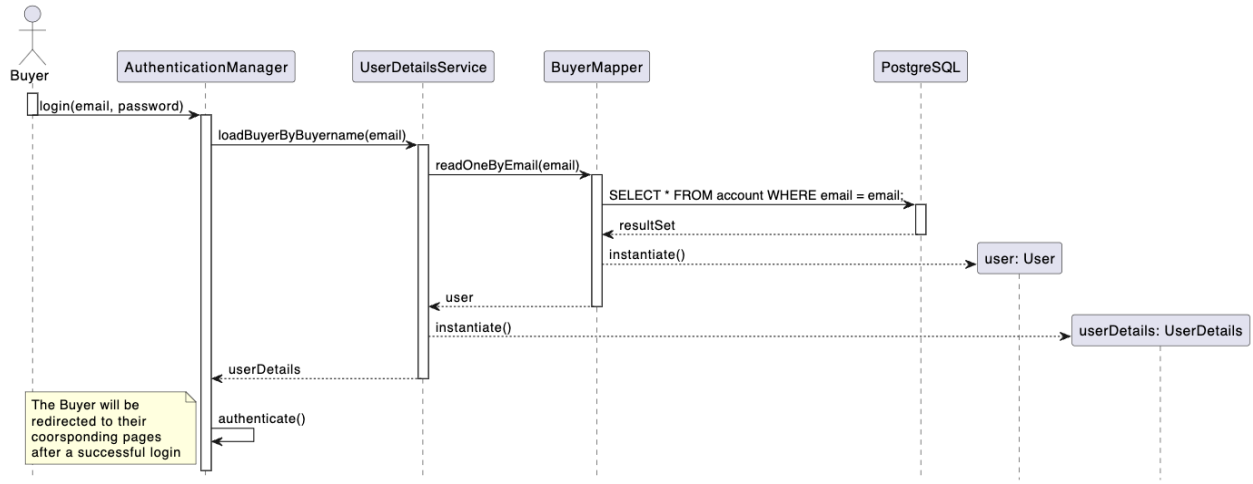


Figure 15: User making a login request

The figure 16 below shows the interaction between different objects when a user tries to register a new account. When a user tries to register an account, the request will be handled by the RegisterUserController. The RegisterUserController will check if the email address already exists in the database by using UserMapper. If the user does not exist, it will encode the password using the Pbkdf2PasswordEncoder and then create that account in the database by calling the UserMapper again. After the account is created, the user will be redirected to the login page.

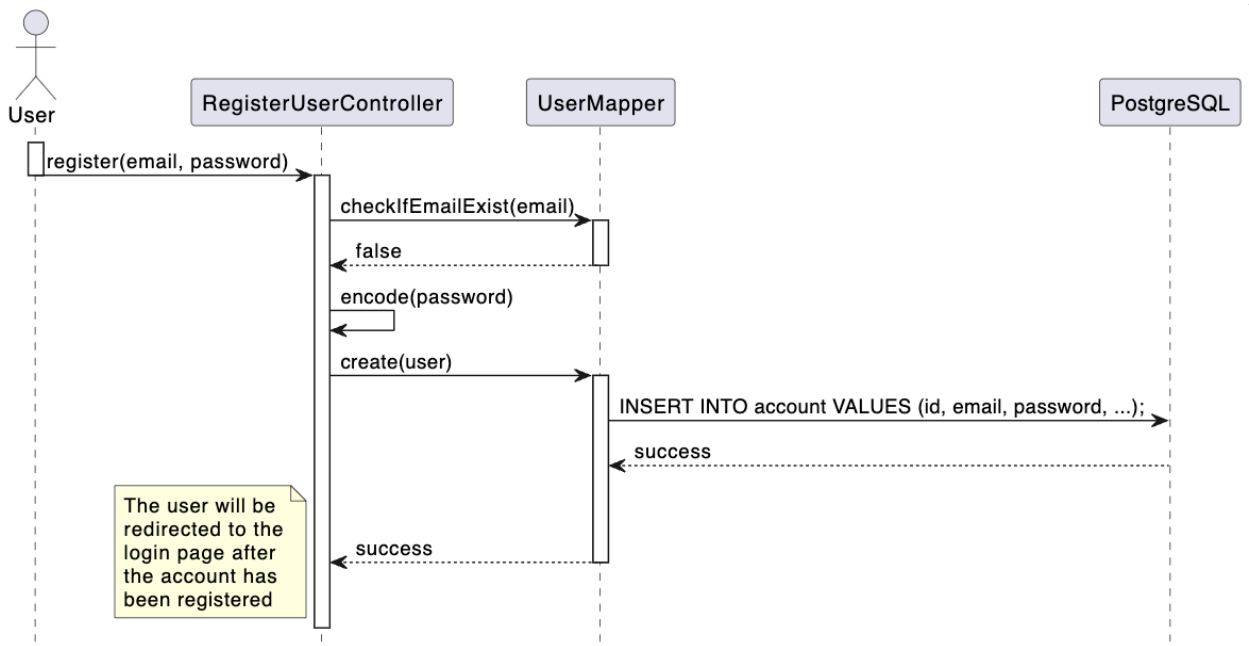


Figure 16: Creating a new account

Authorisation is handled by the AccessDecisionManager which is another built-in class of Spring Security. It makes sure that the client has the appropriate role to request the specified resources. For example, The figure 3 below shows the interaction between different objects when an admin tries to view all users in the system. When the admin clicks on the view all users button, an API request will be sent to the AdminController. Before the API request can reach the AdminController, it will be filtered by the AccessDecisionManager. The AccessDecisionManager will check if the user calling the api is logged in and has the admin role. If those two conditions are met, it will allow the api call to be executed. And the admin will be redirected to the page that shows all the users. Otherwise, the request will be rejected.

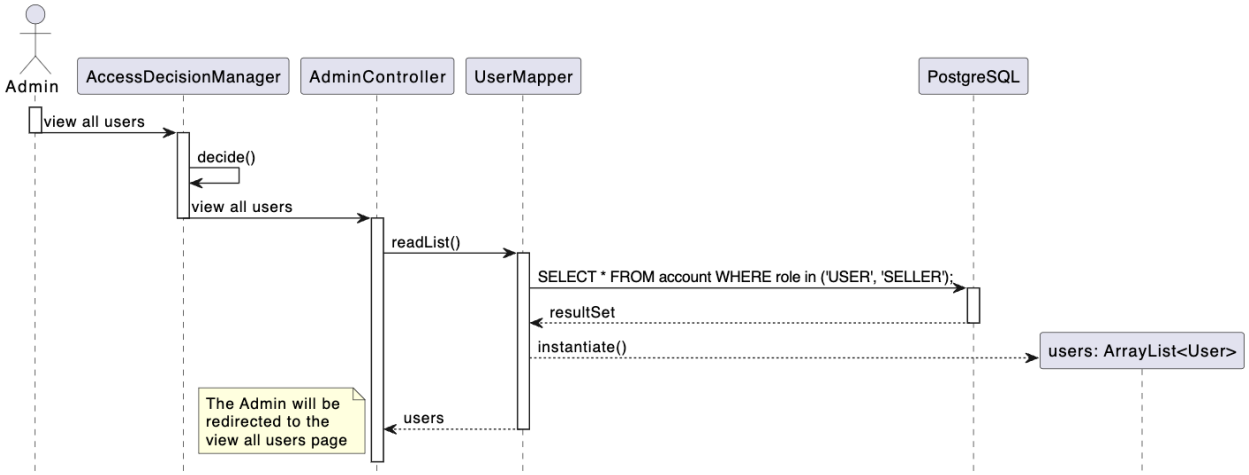


Figure 17: Admin view all users

Pattern: Unit of Work

Design rationale for Unit Of Work

We have implemented Unit of Work in two main cases:
creating a purchase, and
updating the quantity of a purchase

Both cases require updates to two different database tables to keep the quantities consistent. In the first case, creating a purchase involves the creation of a new record in the purchase table in the database as well as updating the listing record from which the purchase was created. The listing's quantity needs to decrease based on the number of items purchased. In the second case, updating a purchase record's quantity field will update the corresponding listing's quantity field as well.

The reason why we used Unit Of Work for these two cases was that we want to avoid inconsistent data in the database. It can happen if one of the updates is successful while the other fails. Unit of work is used to track multiple database changes so that they can be made at the same time as part of a single transaction. Multiple changes being part of a single transaction means that either all of the changes go through or none of them go through so that the system doesn't end up with inconsistent data in the database due to an incomplete transaction.

Having Unit of Work built into the system also allows for a centralised location to control database updates for the purpose of concurrent updates which needs to be implemented in the future. It also allows for easier implementation of a shopping cart in the future as a shopping cart could easily make use of Unit of Work's registering of objects to track the changes made by the purchases in the shopping cart.

How it was implemented

The sequence diagram shown below illustrates how Unit Of Work is implemented in our system. We used the caller registration approach where the calling object (i.e. the controller) is responsible for registering the item in the Unit Of Work. Shown below in the figure, when the client wants to make a purchase, the PurchaseController registers the updated Listing object in the dirty list, and then the new Purchase object is registered in the new list. Finally, the commit() method is called to update both tables in the database. If there is a problem during the commit() method, there will be a rollback and none of the actions will be completed. When the commit() method runs, only one database connection is used.

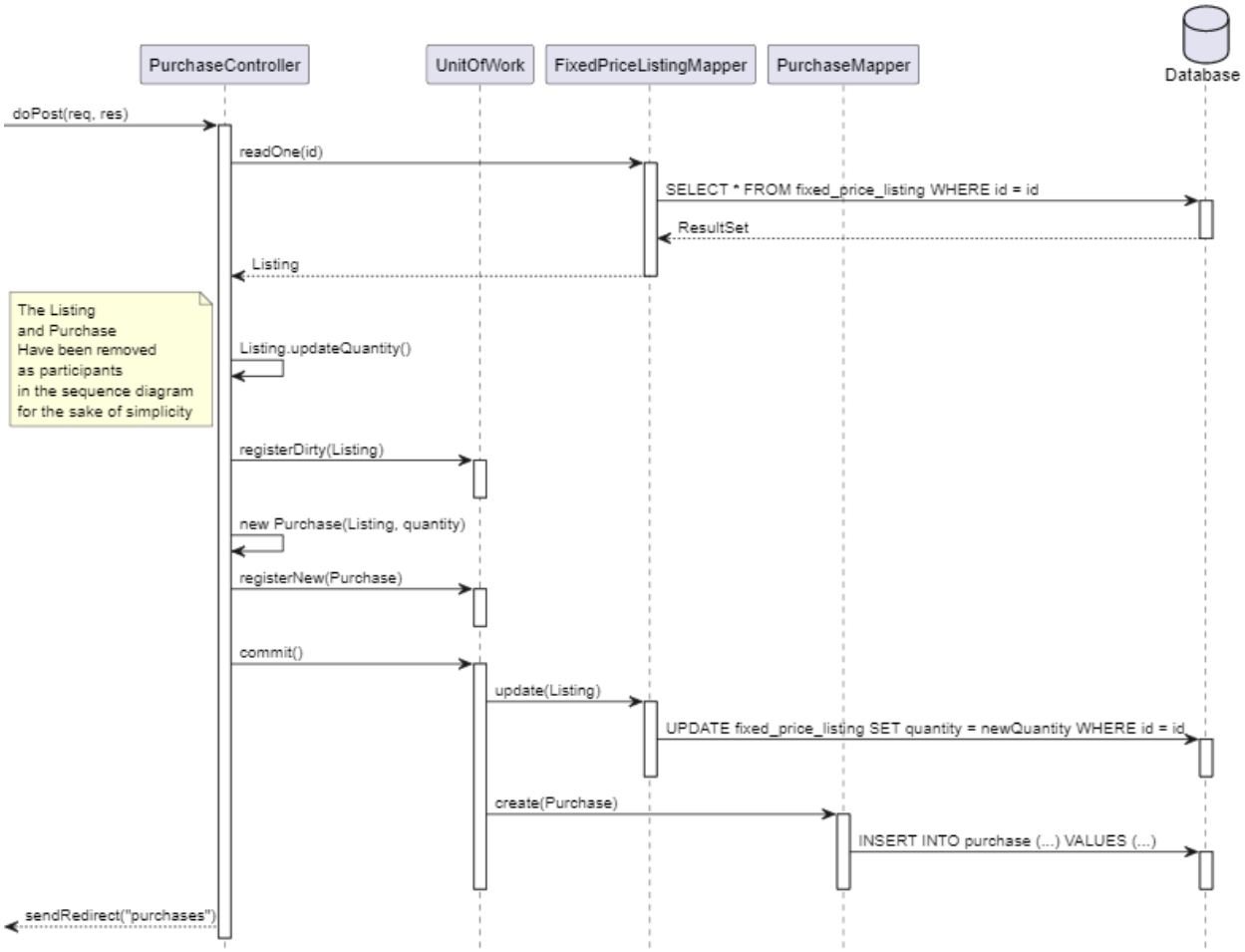


Figure 18: Create Purchase sequence diagram using Unit of Work

SQL Schema Diagram:

The figure 1 below shows the database schema of our application.

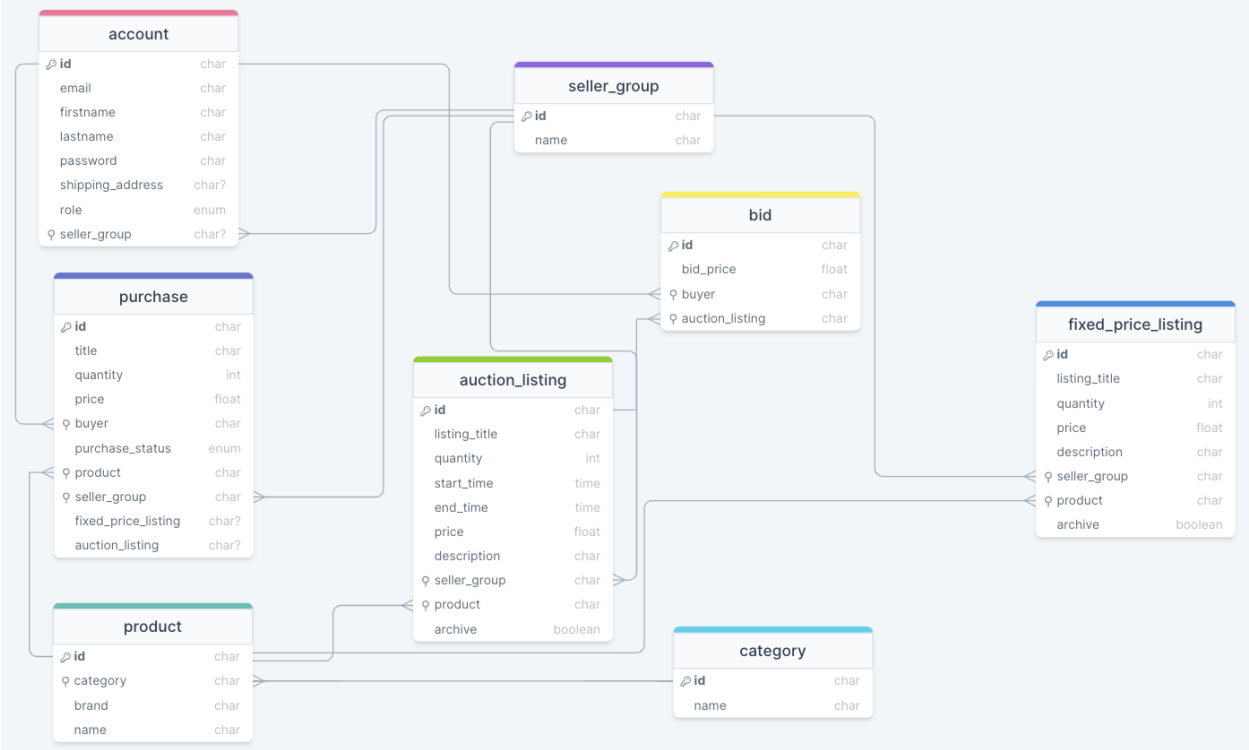


Figure 19: database schema