
Architecture Document

SWEN90007

Marketplace System

Team: Team0

In charge of:

Lai Wei Hong 1226091

Sean Wong 885710

Andrew Liu 1084190

Xi Zhao 1184353



SCHOOL OF
COMPUTING &
INFORMATION
SYSTEMS

Revision History

Date	Version	Description	Author
20/09/22	01.00-D01	Initial draft	Wei Hong
21/09/22	01.00-D02	Review	Andrew
21/09/22	01.00-D03	Review	Sean
23/09/22	01.00-D04	Added Domain Class Diagram	Andrew
27/09/22	02.00-D01	Added High-Level System Overview	Wei Hong
28/09/22	02.00-D02	Added Concurrency System Sequence Diagrams	Sean
30/09/22	02.00-D03	Improved Physical View Section	Xi Zhao
1/10/22	02.00-D04	Improved Development View Section	Wei Hong
5/10/22	02.00-D05	Added JWT System Sequence Diagrams	Andrew
5/10/22	02.00-D06	Improved UI Layout Section	Sean
6/10/22	02.00-D07	Review	Xi Zhao
7/10/22	02.00-D08	Added Concurrency Issues Section	Wei Hong
8/10/22	02.00-D09	Improved Concurrency Issues Section	Sean
10/10/22	02.00-D10	Improved JWT Token Section	Andrew
11/10/22	02.00-D11	Added Testing Strategies Section	Wei Hong
12/10/22	02.00-D12	Improved Testing Strategies Section	Xi Zhao
13/10/22	02.00-D13	Improved Design Pattern Section	Sean
19/10/22	02.00-D14	Review	Andrew

19/10/22	02.00-D15	Review	Wei Hong
19/10/22	02.00-D16	Review	Xi Zhao
19/10/22	02.00-D17	Review	Sean

Contents

1. Introduction	7
1.1 Proposal.....	7
1.2 Target Users.....	7
1.3 Conventions, terms, and abbreviations	7
2. Architectural representation.....	7
3. Architectural Objectives and Restrictions.....	8
3.1 Requirements of Architectural Relevance.....	8
4. Logical View.....	9
4.1 Entity Relationship Diagram.....	9
4.2 High-Level System Overview.....	11
4.3 Class Diagram (Domain Objects)	14
4.4 Class Diagram (Lock Manager).....	15
4.5 UI Layout.....	16
5. Concurrency Issues.....	17
6. Development View.....	28
6.1 Architectural Patterns.....	29
6.1.1 Data Mapper	29
6.1.1.1 Implementation	30
6.1.1.2 Operation (Insert, update, delete).....	31
6.1.1.3 Operation (Find).....	32
6.1.2 Unit of Work, Identity Map & dependency injection	32
6.1.3 Lazy Loading.....	37
6.1.4 Foreign Key Mapping and Associated Table Mapping	39
6.1.5 JWT Tokens.....	39
6.2 Source Code Directories Structure.....	42
7. Physical View.....	42
7.1 Production Environment.....	42
8. Testing Strategies.....	43

1. Introduction

This document specifies the system's architecture Marketplace, describing its main standards, module, components, *frameworks*, and integrations.

1.1 Proposal

The purpose of this document is to give, in high level overview, a technical solution to be followed, emphasizing the components and *frameworks* that will be reused and researched, as well as the interfaces and integration of them.

1.2 Target Users

This document is aimed at the project team, with a consolidated reference to the research and evolution of the system with the focus on technical solutions to be followed.

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
Component	Reusable and independent software element with well-defined public interface, which encapsulates numerous functionalities, and which can be easily integrated with other components.
Module	logical grouping of functionalities to facilitate the division and understanding of software.
CRUD	Acronym for create, read, update and delete within the database

2. Architectural representation

The specification of the system's architecture Marketplace follows the *framework* "4+1" **Error! Reference source not found.**, which defines a set of views, as shown in Figure 1. Each of these views approaches aspects of architectural relevance under different perspectives:

- The **logical view** shows the significant elements of the project for the adopted architecture and the relationship between them. Between the main elements are modules, components, packages, and the application main classes;
- The **process view** shows the concurrency and synchronization aspects of the system, mapping the elements of the logical view to processes, *threads*, and execution tasks;
- The **development view** focuses on aspects relating to the organization of the system's source code, architectural patterns used and orientations and the norms for the system's development;
- The **physical view** shows the hardware involved and the mapping of the software elements to the hardware elements in the system's environment.
- The **scenarios** show a subset of the architecturally significant use cases of the system.

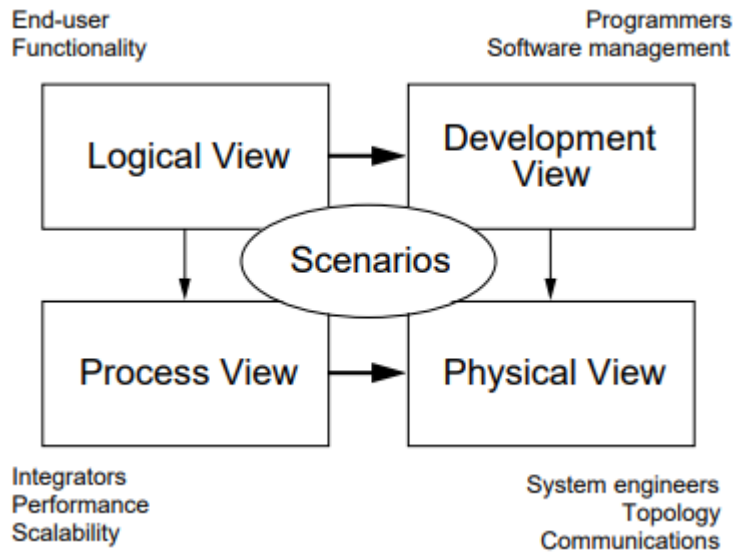


Figure 1. Views of *framework “4+1”*

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

3. Architectural Objectives and Restrictions

With the objective of building a marketplace system, the defined architecture should be easily extensible, where new features can be developed and introduced to the users at a rapid pace. The defined architecture should be able to handle many requests during its operation. Given that this system is developed as part of a team effort, the architecture should be easy to understand. The front-end user interface should be easy to navigate for a new user and there should be a clear separation between the interfaces of different users of the system.

However, several restrictions were placed on the team when developing the system. Several tools / libraries were restricted in this project. The team is, therefore, required to implement and test several important architectural patterns from the ground up.

3.1 Requirements of Architectural Relevance

This section lists the requirements that have impact on the system’s architecture and the treatment given to each of them.

Component	Impact	Treatment
CRUD operations	Defines the way data is written to the database	Data mappers were used to reduce the coupling between the business logic in the service layer and database interactions.
Session Management	Defines the resources that can be accessed by a user	JWT tokens to validate the identity of the user accessing a resource in the back-end server
User Role Identification	Affects the response time of the server when retrieving frequently read data	Role and group ID (for sellers) is incorporated into the JWT tokens

Data Retrieval	Defines the way data retrieval is done, as a large variety of SQL queries needs to be accommodated.	Utilize dependency injection to inject SQL queries to the Find operations in runtime.
----------------	---	---

4. Logical View

This section shows the system’s organization from a functional point of view.

4.1 Entity Relationship Diagram

Figure 2 illustrates the entity relationship diagram that the team has used when modeling our database.

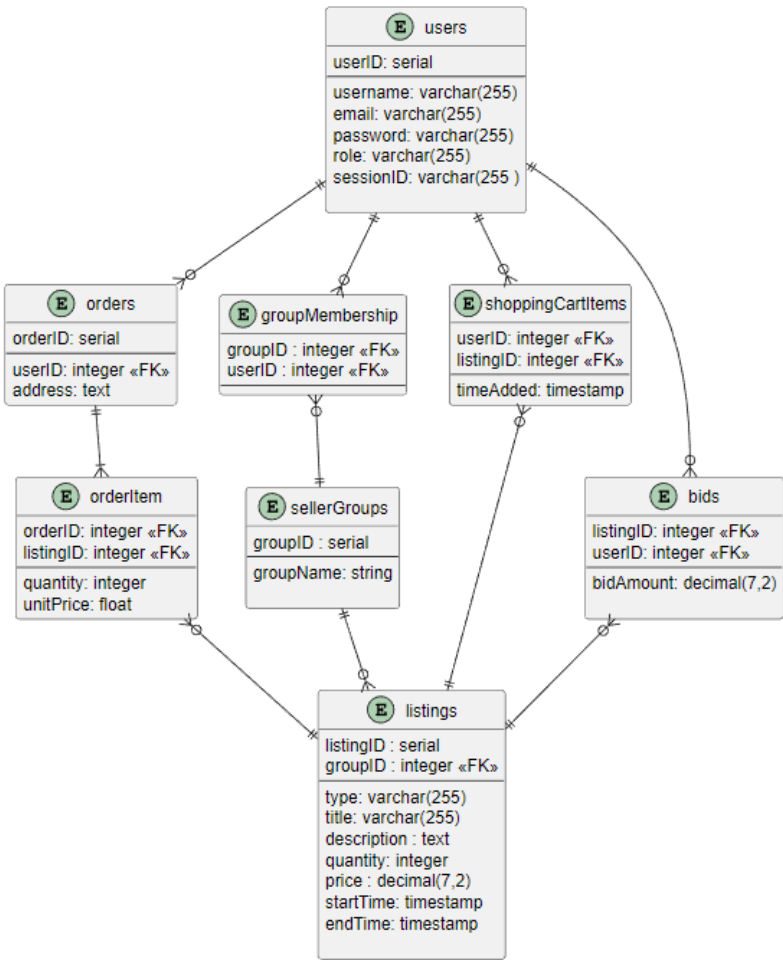


Figure 2: Entity Relationship Diagram

4.1.1 Architectural Discussions (Entity Relationship)

When deciding the entries of the entities in the database, we first look at one criterion, if all entries in the database are unique or not. In our implementation, we decided that the following tables must be unique,

1. Users
Users are an entity that must be unique. If two different users, User A and User B share the same user ID, then User A can act on behalf of User B without the latter's approval. This presents an issue to the system, hence the entries in the User class need to be unique.
2. Orders
Orders represent purchases that a customer performs when purchasing a good. It presents an instance when a business transaction occurs. Every purchase instance is unique from one another, therefore all entries in the Order table must be unique.
3. Seller Groups
Seller groups represent organizations that are selling goods in our marketplace application. Different organizations sell different products, so all entries in the seller group table should be able to differentiate them.
4. Listings
Listings represent the posts created by seller groups to sell their products. A listing, typically, entails a unit of stock that a seller group holds in their possession. Since every unit of stock is unique, the listing for this stock must be unique as well.

Now that we have considered the one-to-one relationships within our database, we need to consider the one-to-many relationships and many-to-many relationships among the entities in our database. Some of these relationships are:

1. Orders -> OrderItems : One-to-many
In every order, there can be more than one item checked out.
2. SellerGroup -> SellerMembership : One-to-many
There can be more than one seller selling for a seller group.
3. User -> Bids: Many-to-many
One user can bid on many auction listings. At the same time, an auction listing can be bid on by many users.

In the instances where one-to-many mapping is present, we will use foreign key mapping. In the case of Orders -> OrderItems, the Order ID will be the foreign key used by OrderItems to help identify the Order that it belongs to. A similar reasoning applies to the Seller Group. By using the Order ID or SellerGroup ID, we can retrieve the list of associated entries.

However, the many-to-many mapping presents issues to the foreign key mapping since there is no single-valued end. Here, we will be utilizing Association table mapping, where we created a Bid table to represent this many-to-many mapping. In this Bid table, we have two foreign keys, ListingID and UserID which help represent the relationship between the User and Auction Listing.

4.1.2 Why use Foreign Key Mapping and Associated Table Mapping over storing everything in the table with the Identity field

If we naively store everything in the 4 tables with Identify fields, we will end up with a table with a lot of duplicate values.

GroupId	GroupName	UserId
1	Team0	Bob
1	Team0	Alice

Table 3: Seller group if GroupMembership does not exist

By utilizing foreign key mapping and associated table mapping, we can reduce the effect of the issue mentioned above. This leads to a much more optimized use of database storage.

4.1.3 Conclusion to architecture design

To summarize our database design, we have used the following patterns to represent the relationships between the entities in our database.

Identity Field	Foreign Key Mapping	Associated Table Mapping
<ul style="list-style-type: none"> - Users - Orders - SellerGroups - Listings 	<ul style="list-style-type: none"> - OrderItems - GroupMembership 	<ul style="list-style-type: none"> - Bids

4.2 High-Level System Overview

The foundation of our system architecture stems from the Model-View-Controller architecture. The view renders the webpages that the user looks at.

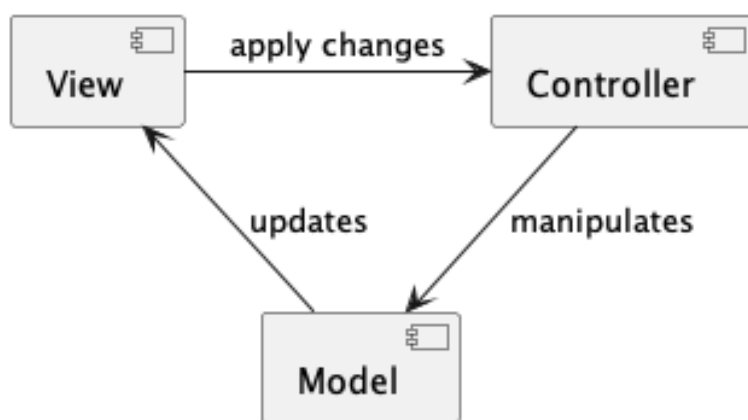


Figure 3: Generic Model View Controller

However, some modifications had to be made to accommodate the architectural requirements of our system. Figure 4 illustrates the high-level system overview that outlines the interactions between different major components in the system

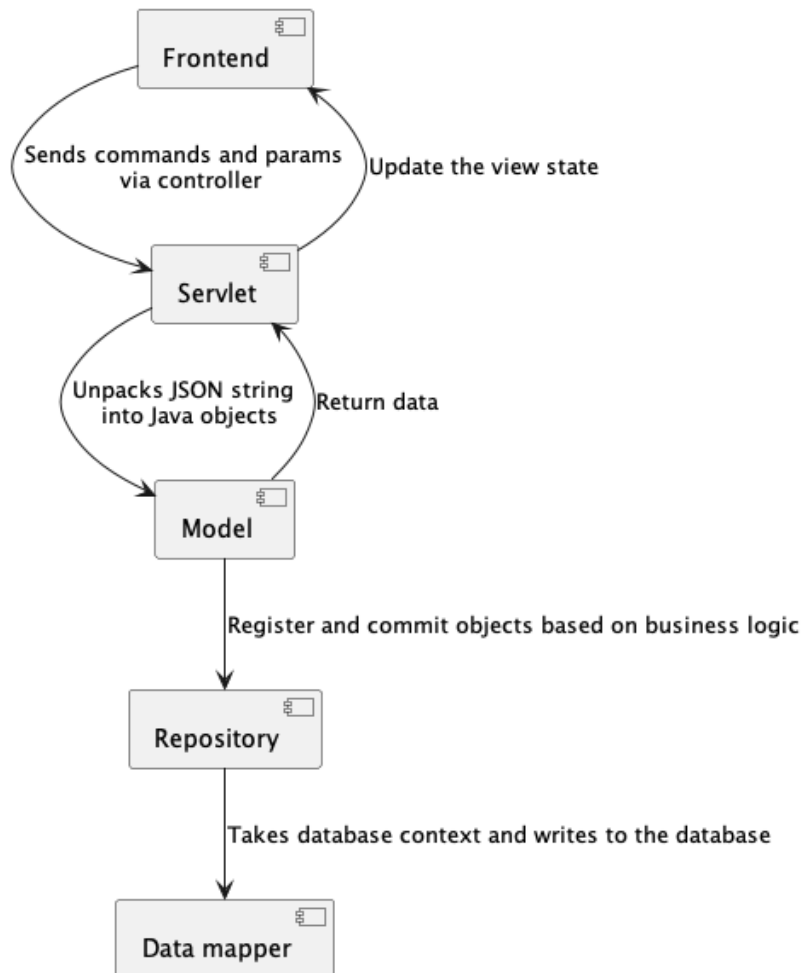


Figure 4: Marketplace System Overview

The frontend will be developed in React while the rest of the components will be developed in Java EE. The frontend will not store any business logic and will only be utilised to display data stored in the backend.

4.2.1 Architectural Discussion: Rich Domain Model vs Anemic Domain Model

The feedback given to the team in Part 2 made us realize the existence of Rich Domain Models and Anemic Domain Models.

A Rich Domain Model is a design pattern where the business logic is part of our domain objects, along with data. With a Rich Domain Model, entity objects reference each other to perform the business logic. This enables the entity object to hide data from the outside of the object, allowing the object to be presented in a comparable manner to an interface.

An Anemic Domain Model is a design pattern where the domain object acts like a data structure, where no business logic is implemented in them. In this instance, the implementation of business logic lies in the service layer. Doing so promotes the separation of data and business logic.

In Anemic Domain Models, the service layer executes all the business logic. This morphs the design to be a lot more procedural when compared to the Rich Domain Model. With the centralization of business logic, the code developed can be much more easily understood and developed. This property also enabled the team to implement unit tests to validate our implementation a lot more easily. Since our business logic is implemented as services, we can isolate components and mock testing conditions to validate the correctness of the implementation.

However, the Anemic Domain Model is widely regarded as an anti-pattern in the Object-Oriented Programming world, when compared to the Rich Domain Model. One of the major drawbacks of the Anemic Domain Model is the need for a developer to be aware of the implementation of functionalities in the service in question. This could be an issue as bugs are prone to occur if there are any misunderstandings over the implementation of the service. On the same note, specific business logic must be implemented, which could lead to a bloated service layer class. However, we would like to contend that with proper refactoring, we are able to keep the service layer classes lean.

4.2.2 What that entails for our system

The decision for the team to incorporate a React front-end plays a huge role in deciding the eventual domain model adopted by the team. The separation of the front-end and back-end has led to the team identifying, either incorrectly or otherwise, the back end as a service that the front-end queries for data. This has led to the architecture design trend towards microservice architecture.

With reference to diagram at 4, the servlets act as REST API endpoints that the front-end interacts with. The endpoint that the servlet is hosting will determine the service that will service the request. A unit of work will also be instantiated at the same time as the service. By the end of the business transaction, changes will be made to the database, typically to the corresponding database. As a result of this, the domain objects are relegated to be pure data structures, leading to us adopting an Anemic Domain Model.

4.2.3 Pros and cons of our architecture and should we have used a Rich Domain Model

As we soon realized, Anemic Domain Model is considered as an anti-pattern in the OOP world, as the way the architecture much closely resembles a procedural-coding paradigm. What led us to choosing this architecture in the first place?

Ease of understanding is one of the key factors contributing to this decision. Since all the business logic is implemented in the service layer, it would be simple for a team mate to understand and develop upon it. This has helped in reducing the lead times for development.

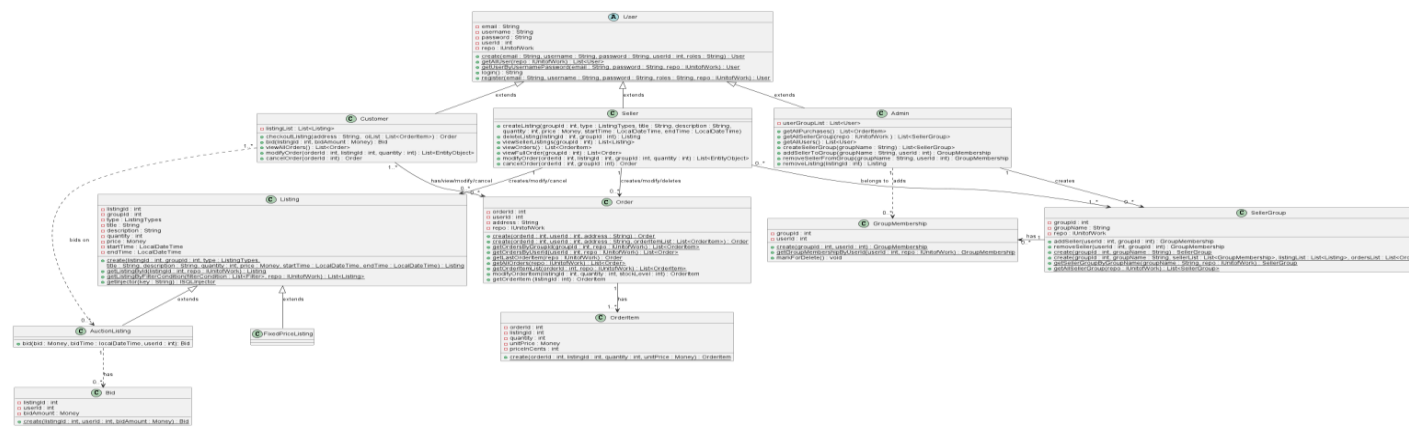
The ease of setting up automated testing is another contributing factor to the choice of such an architecture over the Rich Domain Model. In Rich Domain Model, testing requires mocking all the classes that a domain object may rely on. As an example, if Object A uses Object B and Object C to execute functionA(), then Object B and Object C may need to be mocked specifically for the test. This makes the overhead of test creation to be high. Contrast this with our architecture where we would simply mock the data mapper layer to achieve the same outcome.

Criticism for Anemic Domain Model rings true. The architecture starts morphing into a transaction script, bringing along its cons. Duplication starts becoming an issue, since the business logic can be understood in isolation, our architecture lacks an overarching view of the architecture that ties all the transactions together.

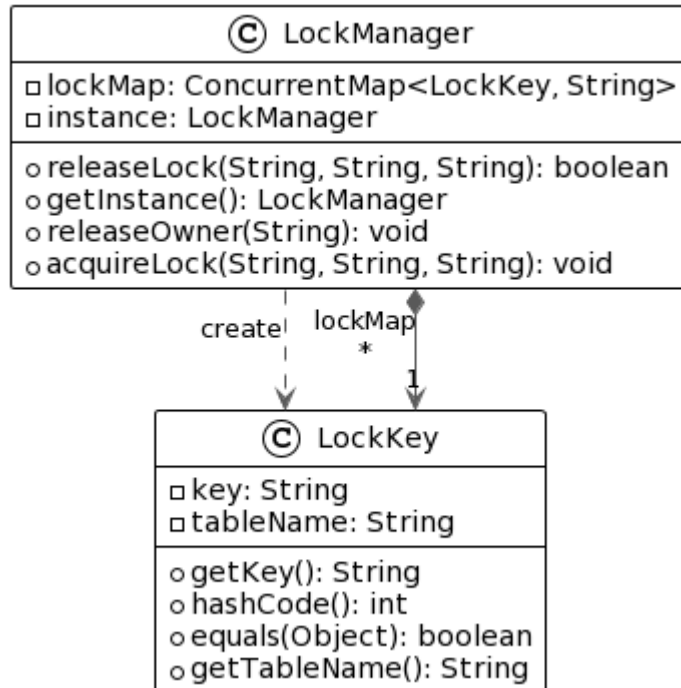
4.2.4 Final decision

Considering the feedback received and the pros and cons, we decided to re-architect our solution, so that it matches that of the Rich Domain Model.

4.3 Class Diagram (Domain Objects)



4.4 Class Diagram (Lock Manager)



The lock manager and its respective lock key is a new component in our system introduced in part 3. They were implemented as part of the pessimistic locking strategy, to help handle concurrency issues.

In our implementation, we have decided to go for an in-memory approach by using a Concurrent Hash Map to hold records of our locks. The data structure is both synchronized and thread-safe, so it is the perfect option to manage lock records while dealing with multiple threads. This method also reduces the requests made to the database, potentially increasing system performance.

We also found that keys can exist in multiple tables so to tackle this, we constructed composite keys using the key and table name to represent a row in our database. This is to ensure uniqueness for every lock acquired by a user.

4.5 UI Layout

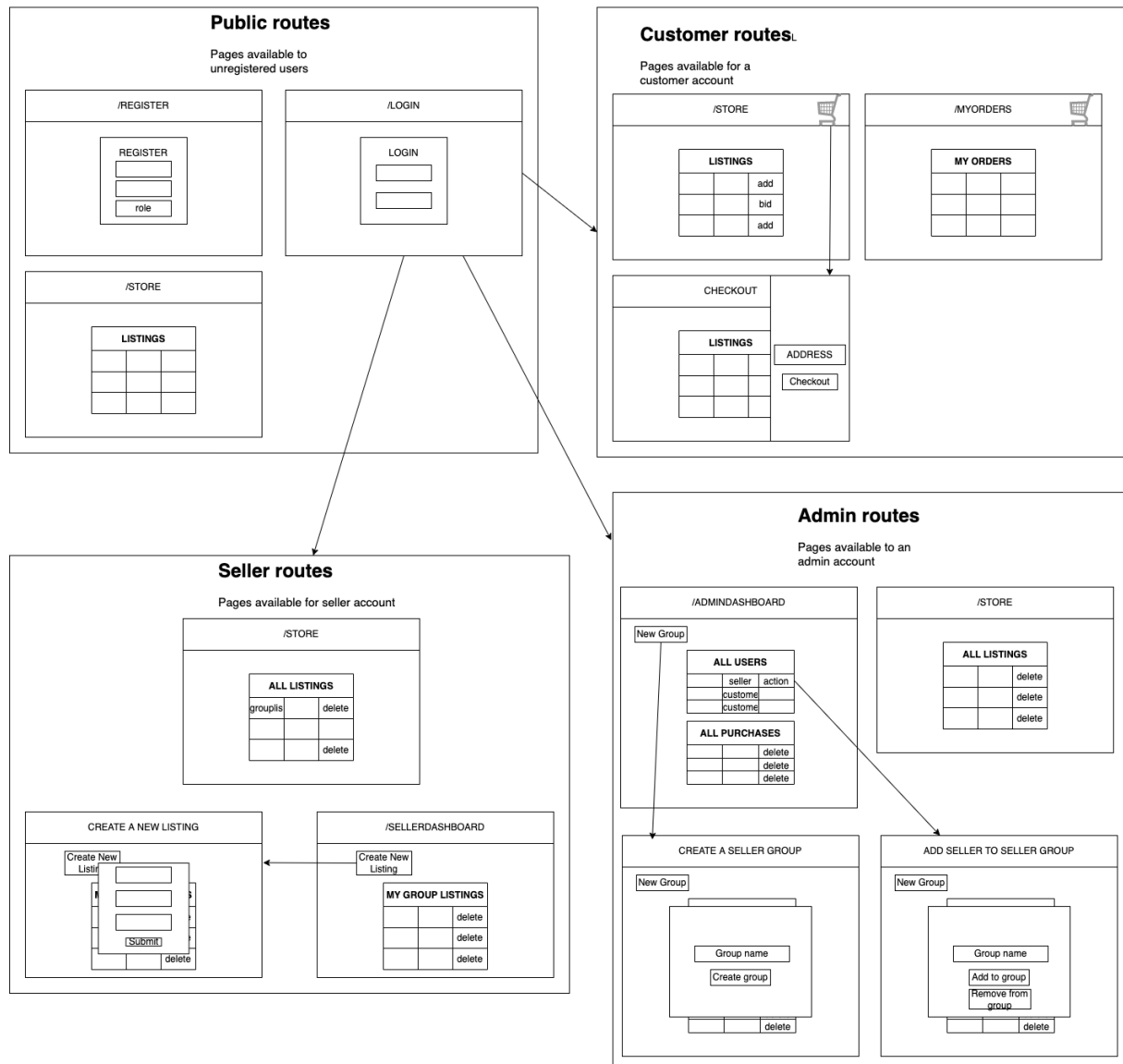


Figure 5: UI Layout

5. Concurrency Issues

5.1 Overview of Concurrency Issues

Concurrency issues arise when multiple transactions are executed concurrently with little control over it. Tomcat, the web application server which our system is built on, accepts data from the front-end in a multi-threaded manner. Therefore, when the traffic to our web server increases, it is without a doubt that concurrency issues will occur as well.

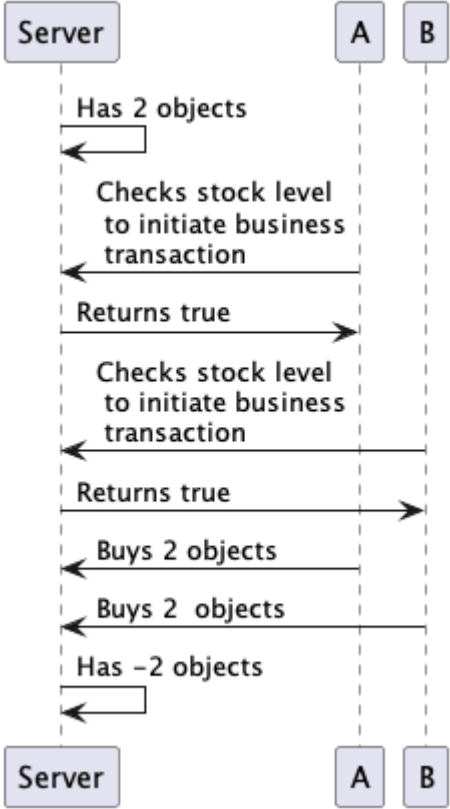
In general, we consider the following concurrency problems to be prevalent in our system.

5.1.1 Incorrect Summary Problem

We define an incorrect summary problem to be one where a transaction is applying some aggregate function on a row in the database while the same row is being updated by another transaction

5.1.1.1 Regions affected

The region where this issue is affecting are regions where the validation of a business transaction relies on the state of an entry in the database. The table below outlines operations that are affected by this problem.

Checkout	<p>To ensure that checkout executes correctly, we need to ensure that the listings have sufficient stock to fulfil the order. However, in a multithreaded environment, this assumption cannot be enforced. We could be in a case where two transactions pass the stock check, but the resultant combined orders of the two exceed the stock level.</p>  <pre> sequenceDiagram participant Server participant A participant B Note over Server: Has 2 objects A->>Server: Checks stock level to initiate business transaction Server-->>A: Returns true B->>Server: Checks stock level to initiate business transaction Server-->>B: Returns true A->>Server: Buys 2 objects B->>Server: Buys 2 objects Note over Server: Has -2 objects </pre> <p>Figure 6: Checkout failing due to lack of concurrency checks</p>
Modify Order	<p>To modify an order, we need to ensure that</p> <ol style="list-style-type: none"> 1. The order exists 2. The listing of the ordered item exists and can fulfil the new quantity

	Again, these two assumptions do not necessarily hold in a multi-threaded environment. A seller could delete a listing while the server is processing the customer's request to modify an order.
--	---

5.1.2 Lost Update Problem

The Lost Update Problem refers to updates to the database lost because of it being overwritten by a more recent update.

5.1.2.1 Region affected

This problem affects regions where we want to avoid duplicate entries in our database. The table below outlines the operations that are affected by this problem.

CreateSellerGroup	Since a seller group represents an organization selling goods and services in our system, all seller groups must be unique to ensure that the customers will be able distinguish different sellers. However, multi-threaded environments expose our system to issues where two admin sessions could lead to duplicate seller groups created.
RegisterUser	Every user has his own list of orders, so the uniqueness among users should be enforced to prevent third parties from making unauthorized changes to a user account.
SellerOnboard	At the current stage of the system, every seller should only be present in one seller group. This restriction cannot be enforced if multiple admin sessions are adding the same seller to different seller groups at the same time.

5.1.3 “org.postgresql.util.PSQLException: FATAL: sorry, too many clients already” Error

In the single-threaded version of our system, connection to the SQL database is obtained by creating a new connection and closing it at the end of the business transaction. However, when this approach is infeasible in a concurrent setting. The database couldn’t create and close connections fast enough to handle the traffic since all our business transactions are short, leading to the “too many clients already” error.

5.2 What is unaffected by concurrency issues

5.2.1 Read-only operations

Read-only operations refer to business transactions that only perform read operations and do not affect the database. These operations include `getAllPurchases()` and `getAllSellerGroup()`. The only issue is that the user could be reading stale data if data has not been fetched after some time. They could then perform transactions using the stale data. However, this is only a minor issue in our system since validation is performed at the server side. While it holds true that the user experience suffers from it, it is a compromise that we are willing to settle with.

5.2.2 Delete-only operations

There are two issues that could arise in delete-only operations, such as seller removal from seller group and listing deletion. They are deleting non-existent entries and deleting the wrong entry.

Handling the case for non-existent entries is straightforward. When the database is queried to delete a non-existent entry, it will simply do nothing, and the state of the database remains unchanged. Since the users can only delete objects that they own, seller -> listing etc., passing the JWT token along with the delete request allows the system to be able to validate the user’s ownership of the object, which

prevents a user from deleting another user's data. This allows the delete operations to be atomic, hence circumventing the concurrency issues.

5.3 Design pattern used and implementation

5.3.1 Design considerations

When designing our solution, we are aware that a lot of our issues can be resolved by utilizing specific SQL clauses, such as UNIQUE (lost update) or FOR SHARE (incorrect summary). However, doing so would go against the spirit of this subject. Therefore, we chose to implement pessimistic offline lock to handle our concurrency issue.

When handling the “too many clients already” error, we implemented a connection pool that reused existing connections. This helps overcome the issue of the expensive connection creation cost. This approach also integrates well with the existing architecture since it doesn't hamper the ability for the Unit of Work class to roll back in the event of failure, since rollback reverts to the last commit and not to the beginning of the connection.

5.3.2 Pessimistic Lock

The choice of pessimistic lock against optimistic lock boiled down to the choice of liveness (something good eventually happens) and safety (something bad never happens). Considering that our system is a marketplace system, we need to ensure that transactions do not deviate from its specifications, so that it cannot be misused by external parties. Since pessimistic lock locks a record for exclusive use until the end of a business transaction, it presents itself as the most desirable solution. This need for safety in exchange for tolerable amounts of latency is the reason pessimistic lock is used to coordinate concurrent transactions in our system.

Optimistic locking is a less desirable option for this system since some of the regions affected are regions with a high rate of contention (checkout and registration). One might argue for the use of Optimistic locking in the other low contention regions suffering from concurrency issues (modify order, seller onboard etc). We acknowledge that, but we find the benefits to be miniscule since the business transactions are short. It also introduces more bugs in the process. This line of reasoning led the team to abandon the implementation of optimistic locking in our solution.

5.3.3 Lock Key comparisons

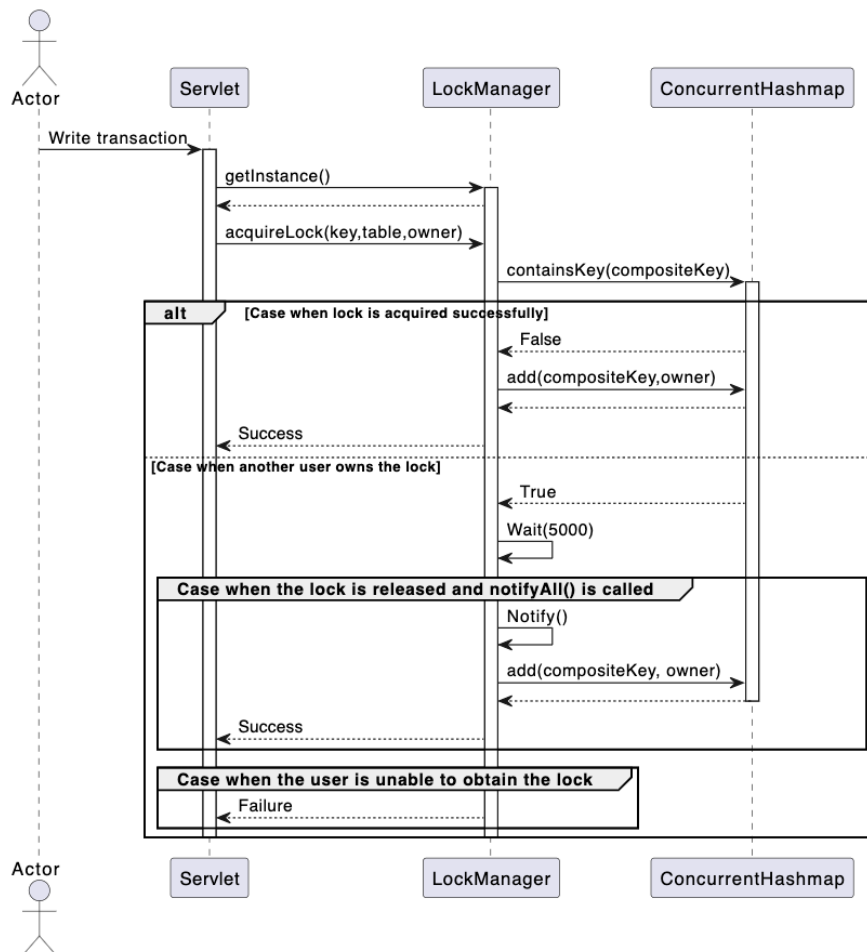
To lock records, the need for a way to differentiate different records is now necessary. Without this, the pattern fails.

One of the ways of differentiating the Lock Key is to see which table the record belongs to. Now, the task becomes a lot simpler, since the data is now categorized into distinct groups. Once we can differentiate the data from their respective data, we can simply reuse the identifiers for the records. As an example, for records in tables with the Identity field, we can incorporate the primary key into the Lock Key. In tables with one-to-many and many-to-many relationships, we can take the concatenated foreign keys as identifiers. To put it simply, the Lock Key would return the following when compared,

$$\text{Lock Key} = \text{hashCode}(\text{table} + (\text{primary_key} \parallel (\text{foreign_key} + \dots + \text{foreign_key})))$$

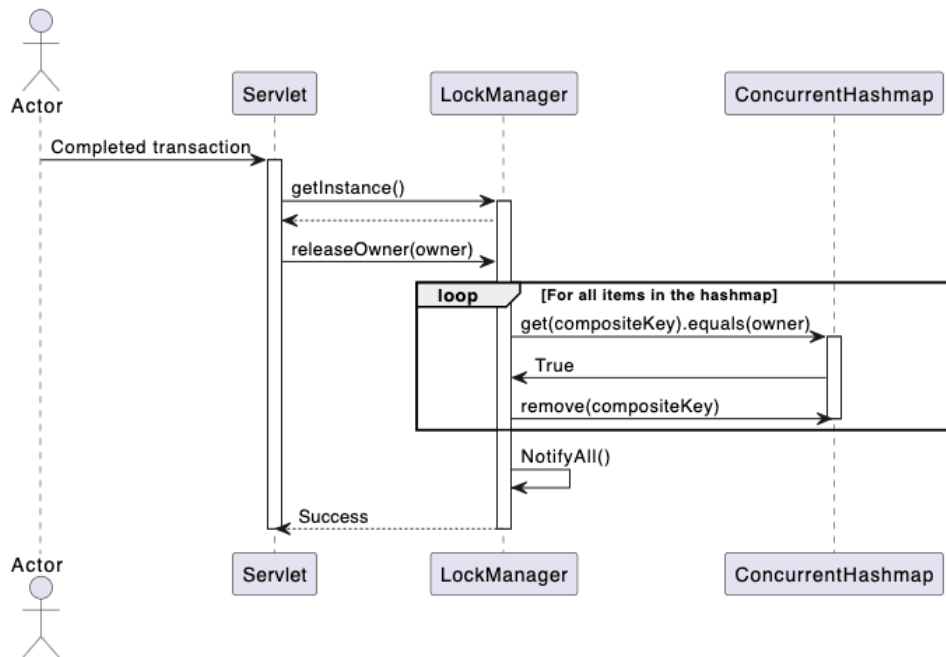
Using our approach, we are confident that we will be able to correctly lock relevant records from being improperly accessed

5.2.2.1 Sequence diagram showing a user attempting to acquire a lock



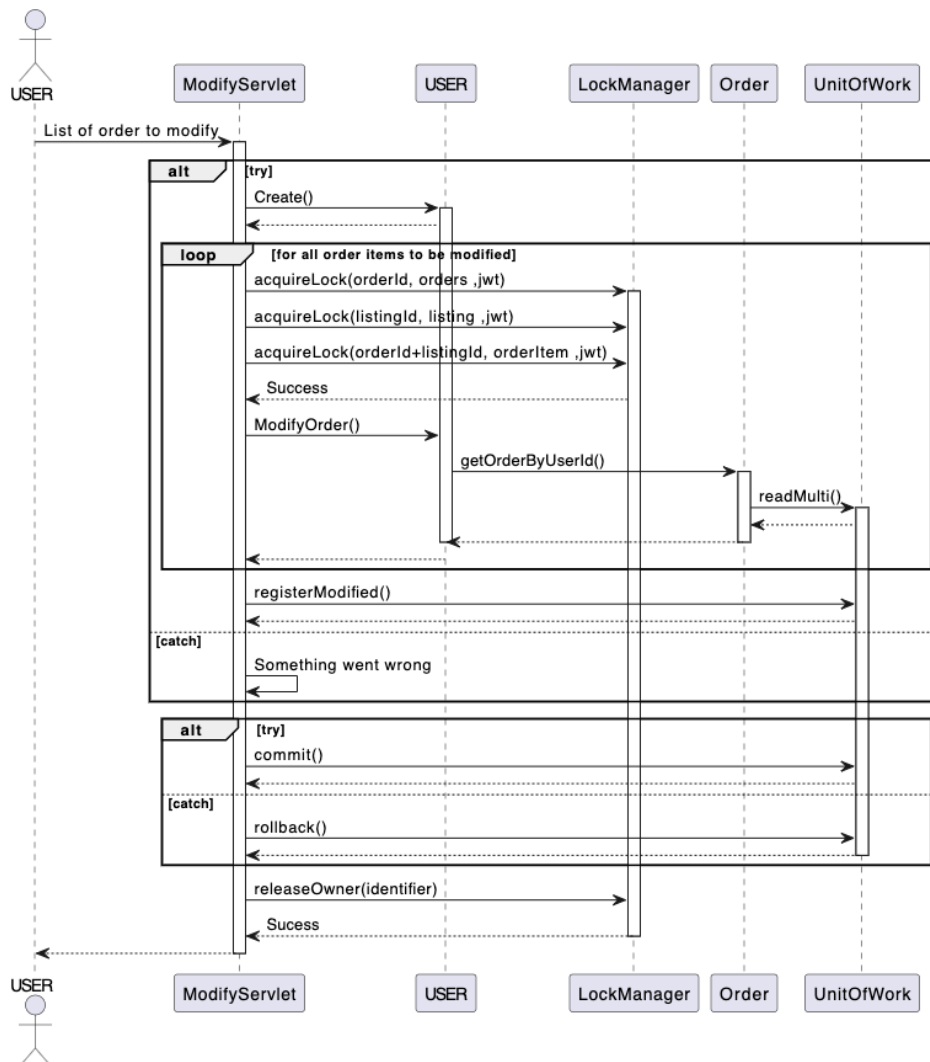
When a user attempts to acquire a lock, they first check for the existence of the key in the Concurrent HashMap before acquiring the lock. If the key does not exist, the user obtains the lock by adding the composite key and owner identifier as an entry into the map. If a key is present in the map, the user will wait for five seconds before returning an exception if the lock is not released.

5.2.2.2 Sequence diagram showing how a user releases a lock



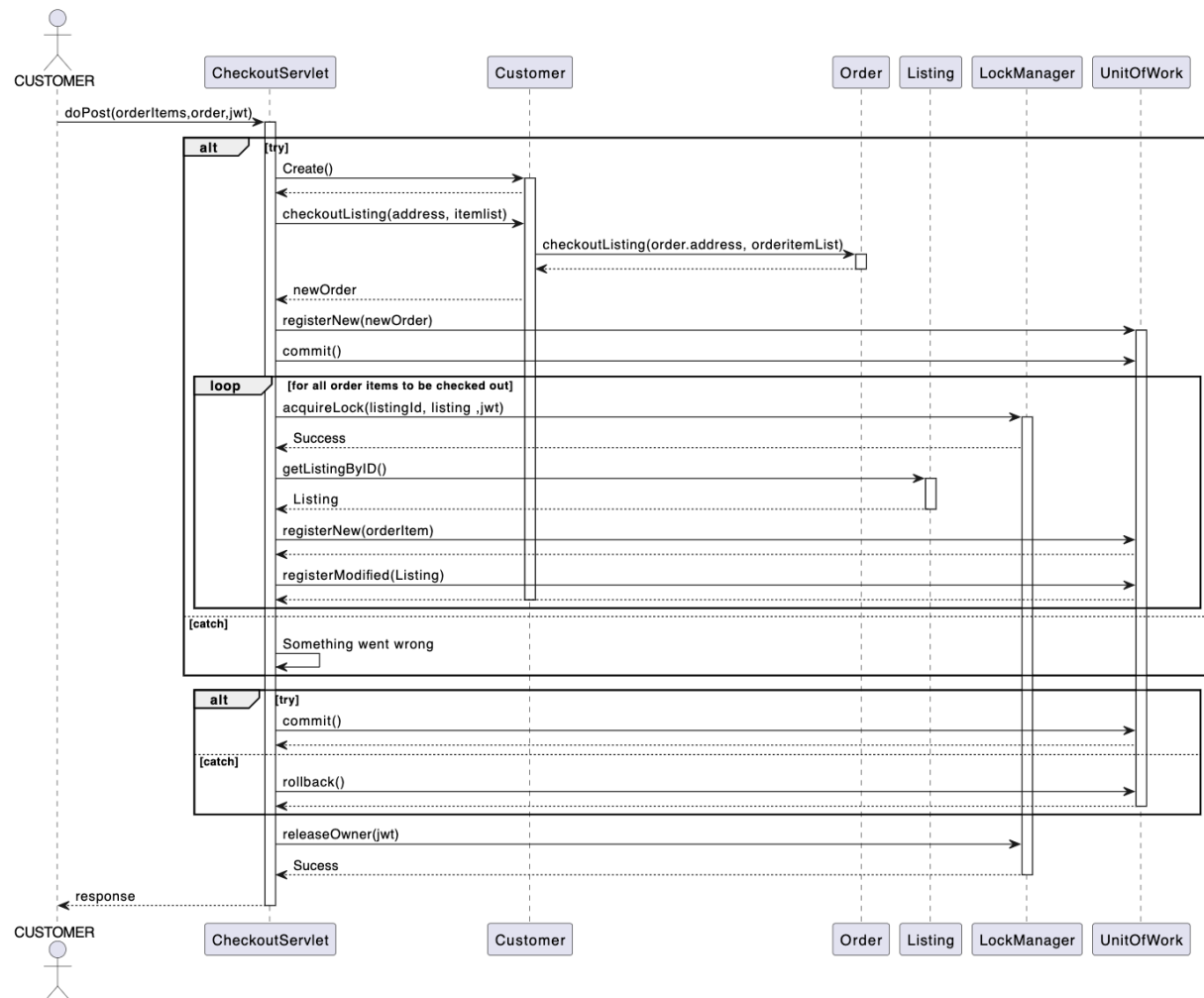
The process above shows how we release all locks for an owner. The HashMap will be searched for any locks belonging to the owner and release them. This is useful especially for complex transactions as it lowers the chance of forgotten locks and ensures that all locks involved are released at the end of the process.

5.2.2.3 Sequence diagram of user attempting to modify orders



The diagram above demonstrates how locks are implemented in our Servlets. The user will acquire the relevant locks, in this case, for rows belonging to orders, listings and order items. Once all locks have been obtained successfully, the user can continue the transaction. If a user fails to obtain any of the locks, the lock manager will throw an exception which blocks the transaction. The `releaseOwner()` function is called at the end of the servlet to ensure that all locks that belong to the owner are released despite the outcome of the transaction. This is to ensure that no deadlocks can occur, and no locks are left locked.

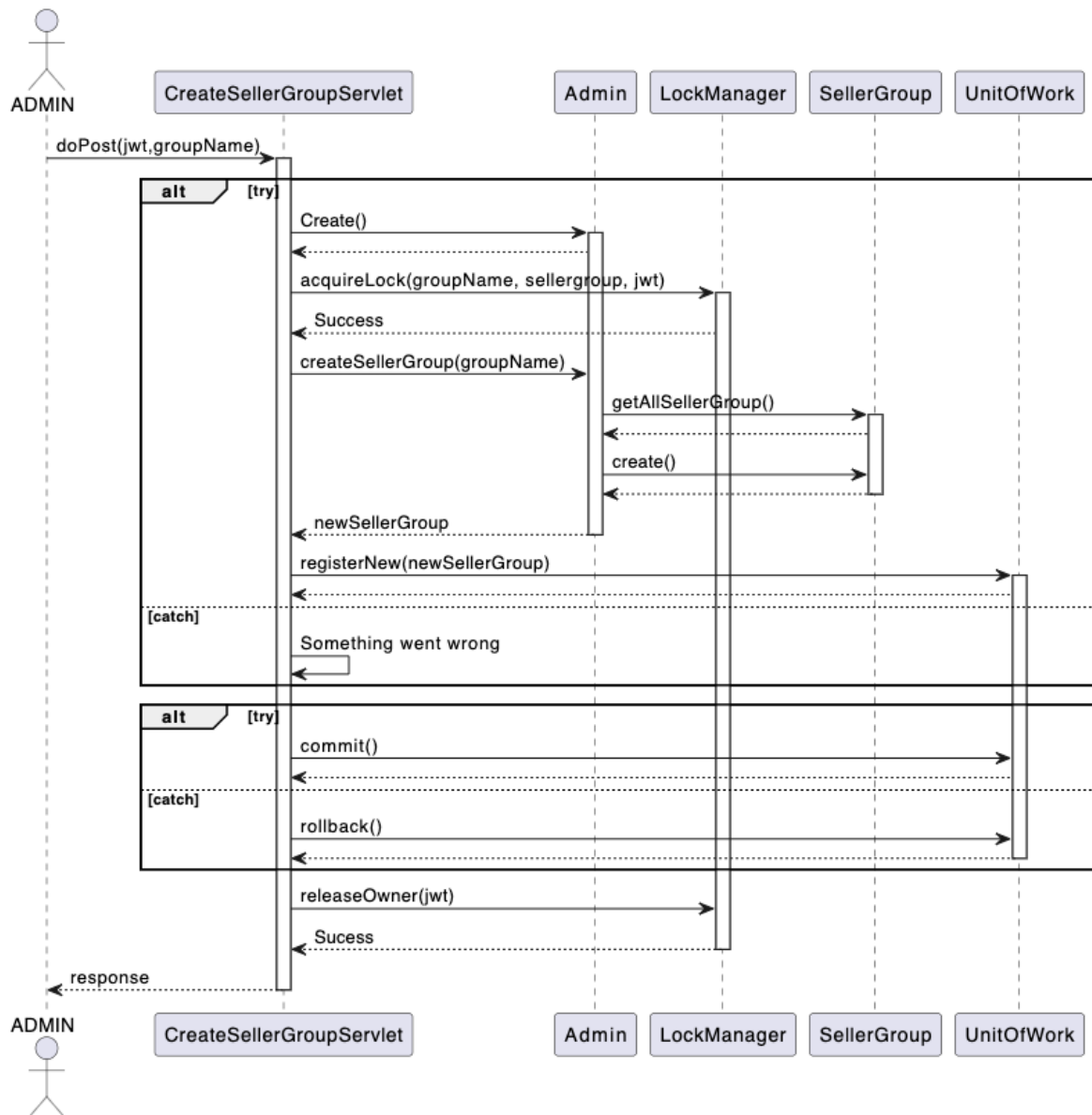
5.2.2.4 Sequence diagram for a customer checking out a list of orders



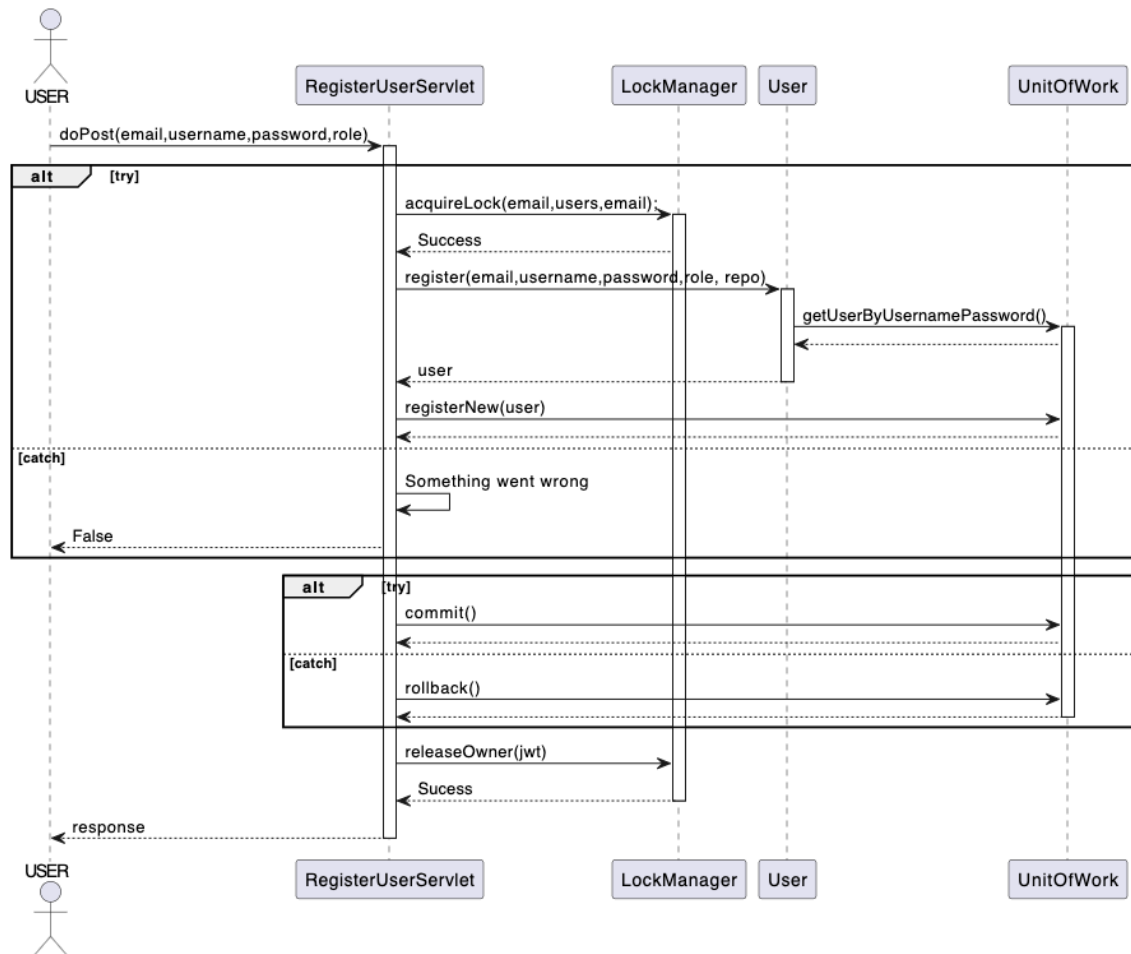
As displayed above, the customer will have to acquire a lock for every listing within their list of orders that they have checked out. This is because the stock of the listings will have to be updated to reflect the customer's order. Acquiring the lock will block any other transactions from modifying the rows to prevent the incorrect summary problem and to ensure that listings' stocks are updated accurately. If the lock cannot be acquired, the transaction will roll back and release any other locks acquired by the owner.

The following sequence diagrams demonstrate how we utilise our lock manager to solve duplicate entries in the database. We are locking the row we are about to create, ensuring that when multiple users attempt to execute the transaction, our system handles the issue pessimistically.

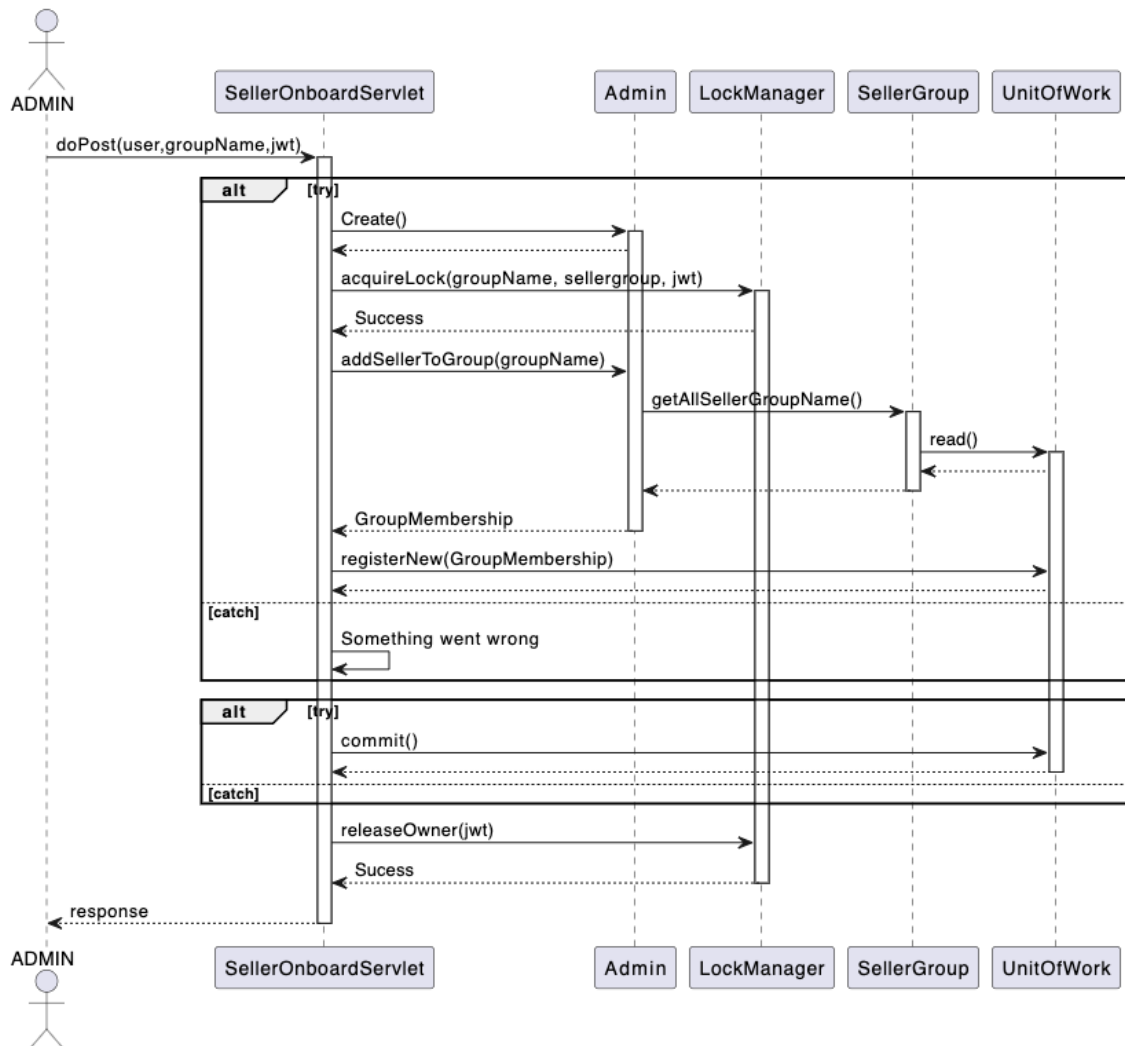
5.2.2.5 Sequence diagram for an admin creating a new sellergroup



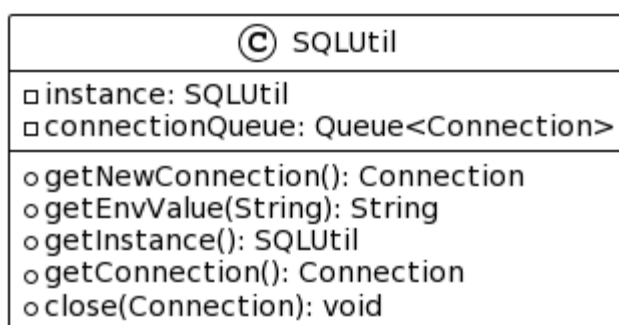
5.2.2.6 Sequence diagram showing user registration



5.2.2.7 Sequence diagram showing admin adding a seller to sellergroup



5.3.4 Connection Pool (Implemented with SQLUtil)



In our implementation, we aim to promote database connection reuse. To achieve that, we have a queue that stores references for existing connections to the database. When a connection is requested, the SQLUtil class checks the queue to see if there are any free connections in the queue. If not, the transaction busy waits until a connection is added back to the pool. We opted to use BlockingQueue as our implementation for the queue storing connections, due to its ability to allow for waits when inserting and removing elements from the queue.

5.3.4.1 Design issue

Since we have constrained our system to a set number of connections, it is important that the connection objects are not left dangling as the system could eventually lose the way to connect to the database due to the system having no way of generating new connections. We should ensure that connections are created and closed at the same section of the code where data reads, and writes are required, so that human errors can be minimized.

So, the question is now narrowed to where that place could be. What part of our code serves as a gateway for all read and write operations in our system? Exactly, the Unit of Work. Our implementation entails getting the connection at the beginning of a read operation and returning it at the end of said operation. When dealing with writes and updates, we only get the connection at the time of commit. As Unit of Work processes the registered objects to its respective mappers, the database connection object is supplied as well. Finally, regardless of whether the commit is successful or rolled back, the connection is returned to the SQLUtil queue.

5.3.4.2 Operations

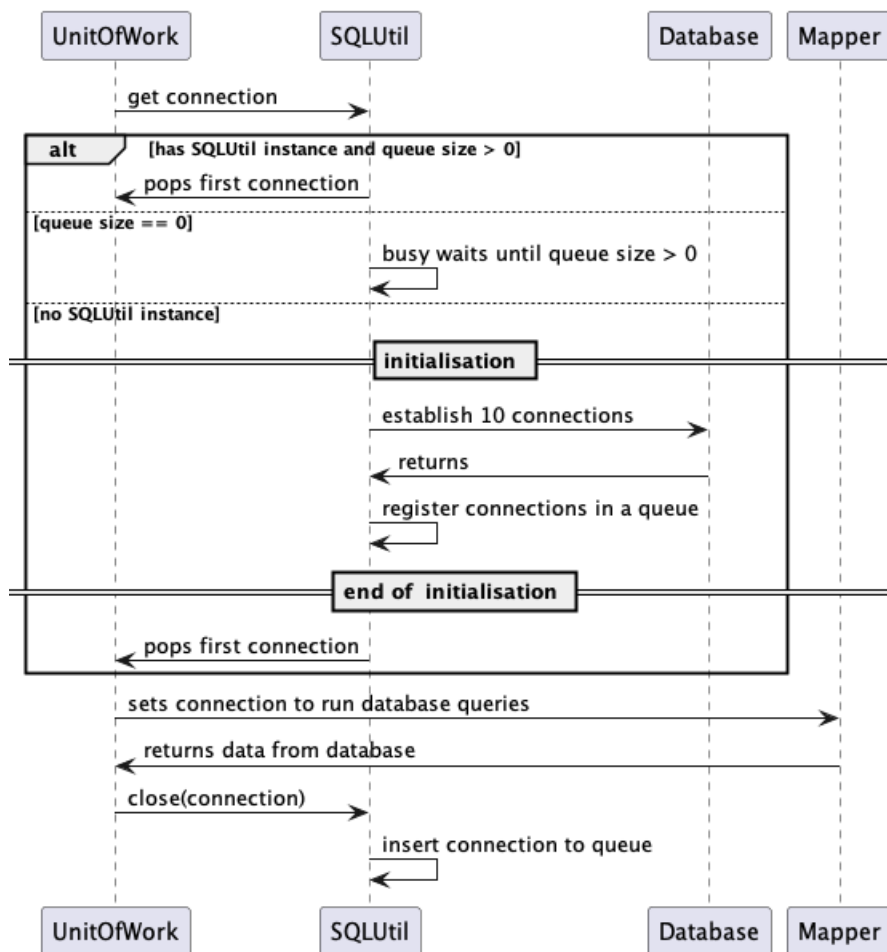


Figure 7 : Connection pool operations

6. Development View

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

6.1 Architectural Patterns

Pattern	Reason
Data Mapper	The data mapper gives us a way to abstract our database logic, typically in SQL queries, from the rest of the system. This enables the system to be portable between different database offerings from different providers.
Unit of Work	The unit of work pattern is used to facilitate the operations of other architectural patterns that we have implemented, namely the identity map and dependency injection. A single unit of work is defined for every business transaction and an “all or nothing” principle is used in the operation of the Unit of Work. If a failure is detected during a business transaction, all changes are rolled back.
Lazy Loading	Lazy load allows us to only load data that is relevant for our use cases.
Identity Map	Identity maps allow us to enforce the fact that only one instance of an object is instantiated throughout a predefined scope (one business transaction, in our case). It also gives us access to a context-specific, in-memory cache that allows data to be retrieved faster than it is fetched.
JWT Tokens	JWT Tokens are adopted by the team as a means of session management. The tokens are also used to store user-specific context, such as user ID, group ID and roles to reduce the number of reads to the database.
Dependency Injection	There is a wide range of read conditions that need to be considered in this project, which limits the ability of the data mapper to provide a generic read function. Dependency injection allows us to inject relevant SQL queries in runtime to the read function. This also helps reduce the coupling between the choice of database and the data mapper.

6.1.1 Data Mapper

Data mappers are used to abstract database logic away from business logic. By decoupling business logic from database transactions, we can switch database providers with minimal code refactoring needed in the business logic.

In our implementation, every table in the database matches with a data mapper. This decision was made as every table has a different structure, which made the implementation of a general data mapper unfeasible.

6.1.1.1 Implementation

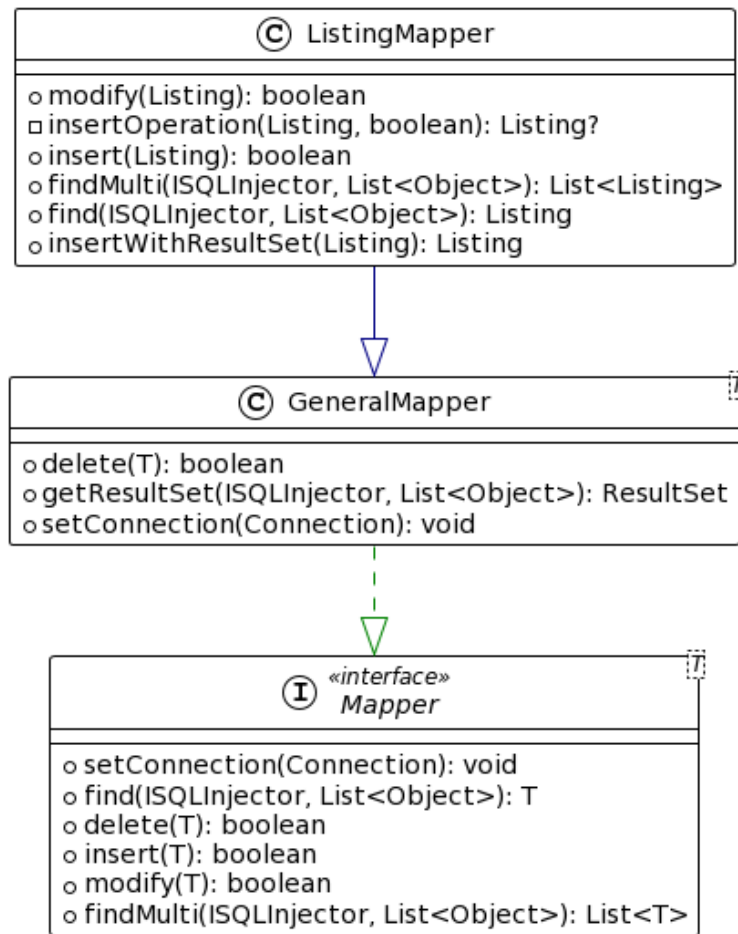


Figure 8: Data mapper class diagram

We will be using the data mapper for the Bids object as an example to explain the implementation of our data mapper. The data mappers implement the Mapper interface that provides the CRUD operations to external classes, namely find, insert, delete, and modify. In addition to that, all implementations of data mappers extend a GeneralMapper class which provides functionality that is shared across all data mappers.

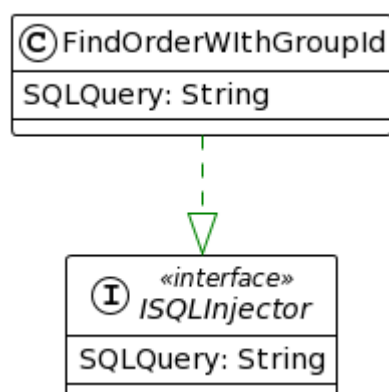


Figure 9: SQLInjector class diagram

Dependency injection is a key component in data mappers. Since we integrated Identity Maps as part of the Unit of Work read operations, we need a generic Find() function in the data mapper so that it can be integrated seamlessly into the Unit of Work class. Relevant SQL queries are injected into the Find() operation via an ISQLInjector class from the Unit of Work class. The database reads the SQL query injected and returns the relevant data. As an example, when the new FindOrderWithGroupId().SQLQuery() is called, “SELECT oi.* FROM orderitems oi JOIN listings l on oi.listingid=l.listingid where orderid=?;” is returned. This will then be used by the mapper to return the relevant data.

6.1.1.2 Operation (Insert, update, delete)

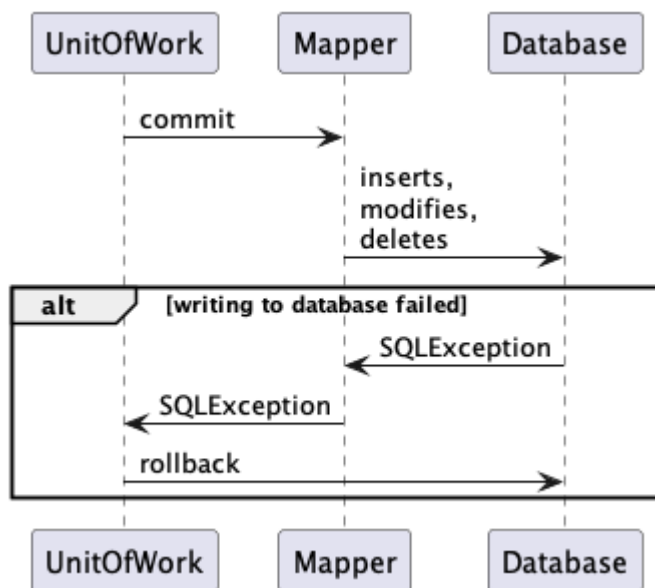


Figure 10: CRUD operations with mappers

The insert, update and delete operations are straightforward processes. When the UnitOfWork commits the change, it calls the mapper to execute the insert, modify delete operation. The exact implementation of the database transaction is implemented in the mapper, which allows it to execute the transaction.

6.1.1.3 Operation (Find)

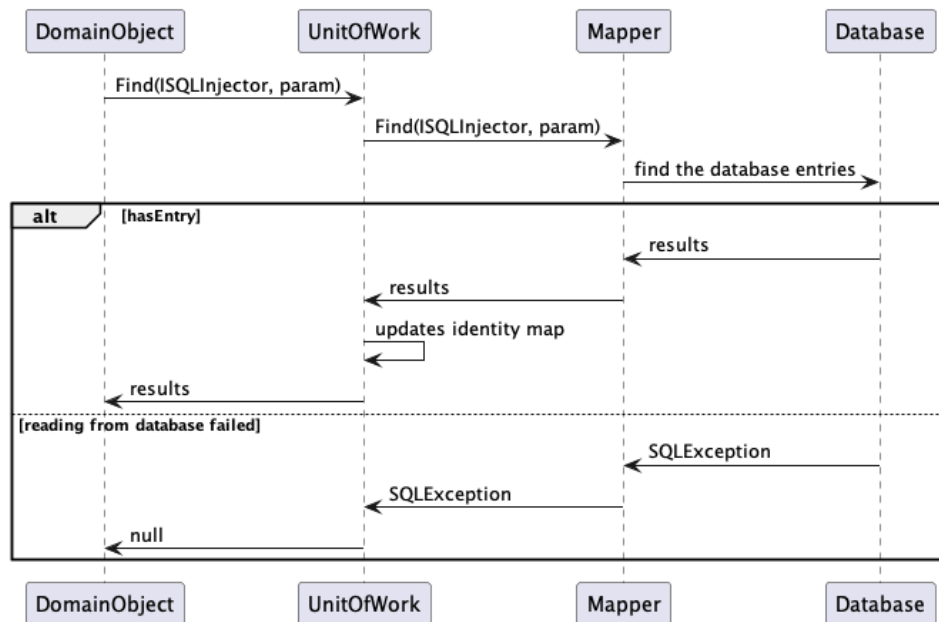


Figure 11: Find operation

To accommodate a general Unit of Work, we need a general Find operation in our data mappers. This is achieved using an SQLInjector that has a specific SQL query. During runtime, this object is injected into the mapper from the domain object via the Unit Of Work. The mapper reads the SQL query and executes it to return the result back to the domain object.

6.1.2 Unit of Work, Identity Map & dependency injection

Unit of work is a design pattern used to group one or more operations into a single transaction, so that all operations either pass or fail as a unit.

Identity maps are incorporated into the Unit of Work implementation as the Unit of Work is the only gateway where data flows from the database to the business logic. The Identity Map serves two functions in our implementation. One, it ensures that there is only one instance of a domain object in the system. This ensures that two different sections of the transaction are interacting / modifying the same domain object, ensuring the consistency of the operation

To accommodate for the identity maps, the UnitOfWork class needs to support read operations. Since a Unit of Work commits for different entity objects, the data mappers must be able to support generic Find() operations. We achieve that by incorporating dependency injection in our implementation. See Section 6.1.1 for more information.

6.1.2.1 Implementation

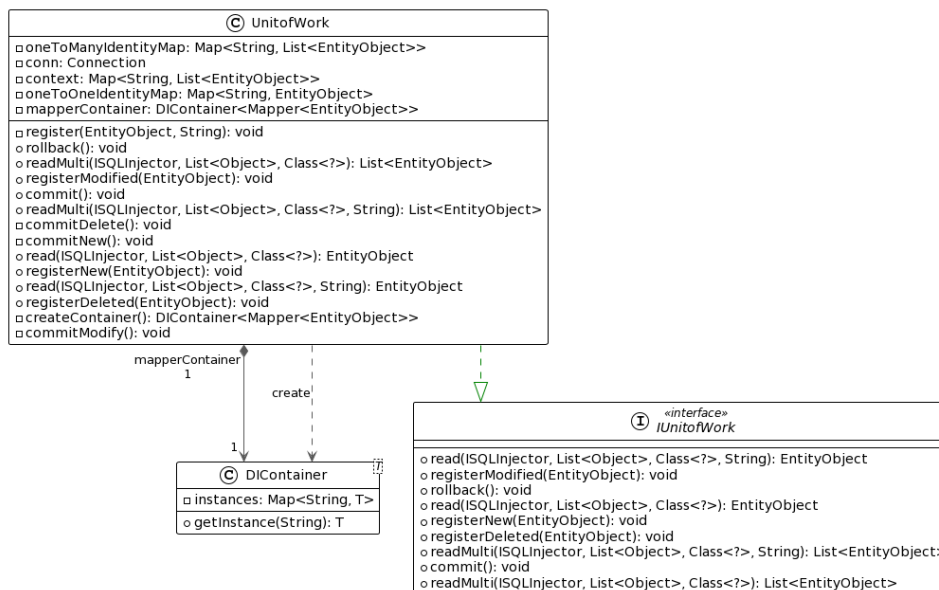


Figure 12: Unit of Work class diagram

The table above outlines the class diagram of our Unit of Work. The use of a dependency injection container allows us to get relevant data mapper instances for our Entity Object, i.e., Listing -> ListingMapper etc. By utilizing this, we can have a generic Unit of Work class that is able to handle the registration and commitment of all Entity Objects.

6.1.2.2 Design decisions

We defined a business transaction to be a unit of work, so we want to coordinate changes to either all the transactions are committed at once or rolled back if failure is detected. To support this operation, the SQL connection passed to the Unit of Work instance should have the attribute setAutoCommit set to false. With this change, changes to the database will only be pushed when commit() is called by the Unit of Work.

6.1.2.3 Operation

The Unit of Work class is governed by four major CRUD operations, Find, Insert, Modify, and Delete. In addition to that, its behavior is also affected by the implementation of a commit and rollback function. In our implementation, one Unit of Work governs one business transaction and is terminated at the end of it.

6.1.2.3.1 Initialization

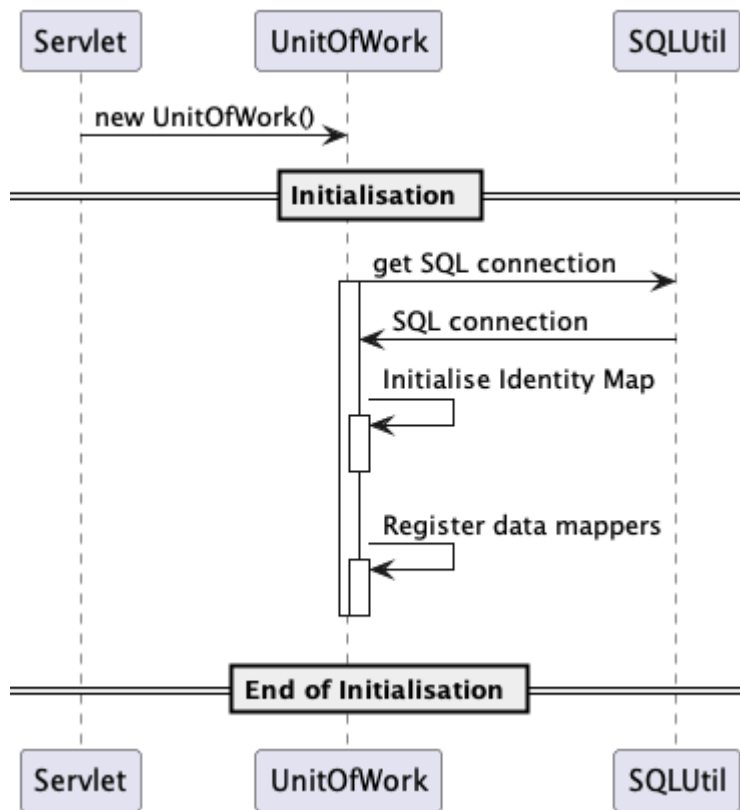


Figure 13: Unit of Work initialization

When Unit of Work is initialized, a SQL connection is acquired, so that the Unit of Work can pass it to the mapper during CRUD operations. The table below outlines the mapping within the dependency injection container in the Unit of Work class, the value holds an instance of a mapper.

Key	Value
Bid.class	BidMapper
GroupMembership.class	GroupMembershipMapper
Listing.class	ListingMapper
OrderItem.class	OrderItemMapper
Order.class	OrderMapper
SellerGroup.class	SellerGroupMapper
User.class	UserMapper

Figure 14: Mapping when data mappers are registered

By implementing such a mapper, we can determine the mapper required by an EntityObject using Reflection in Java.

6.1.2.3.2 Find

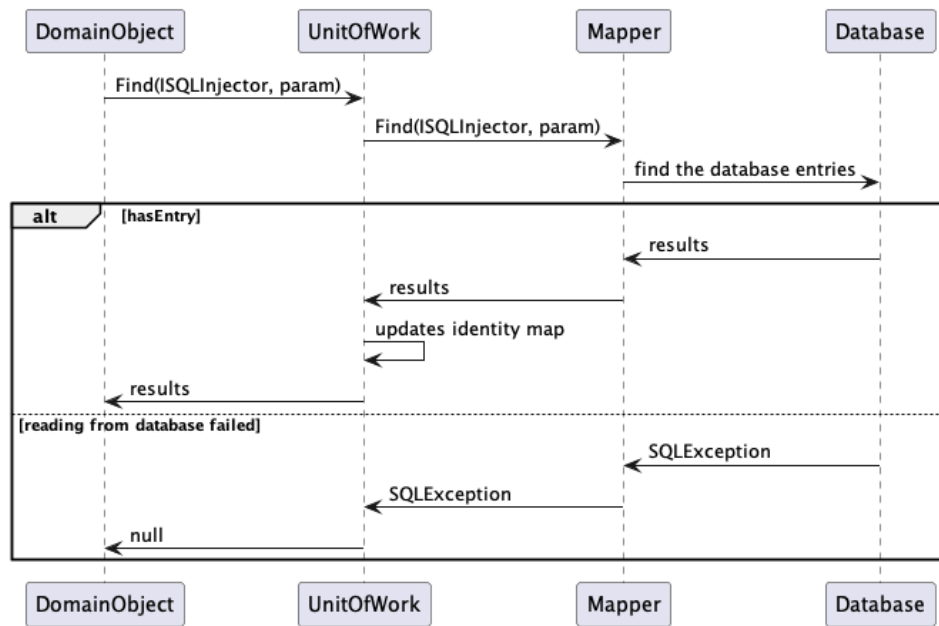


Figure 14: Find operation

When the Find () operation is invoked, we first look at our Identity Map to see if an instance of the domain object has been instantiated. If not, the injected SQL query and its corresponding parameter is sent to the data mapper's Find() operation to retrieve the entries.

6.1.2.3.3 Insert, Modify and Delete

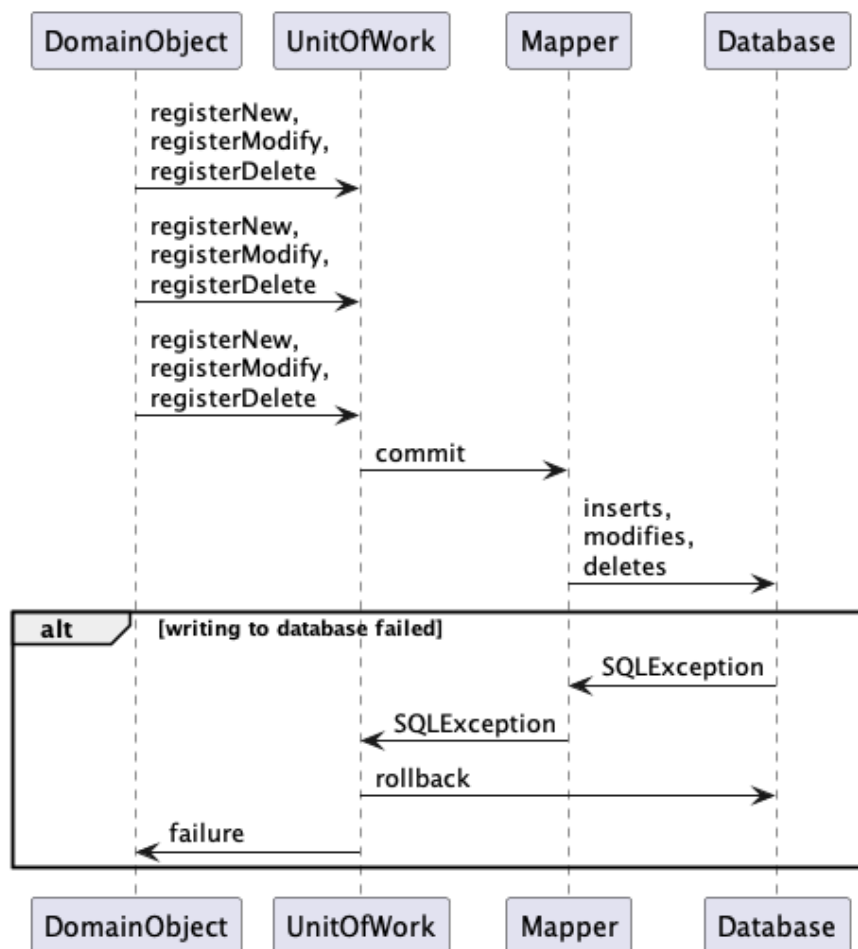


Figure 15: Insert, modify, delete

After the UnitOfWork is initiated at the beginning of the business transaction, all inserts, modifications, and deletions are registered in the Data Context map within the Unit of Work class. The Data Context map keeps track of the changes that need to be committed at the end of the business transaction. If a failure occurs during the business transaction, the transaction is aborted and rolled back. Otherwise, all the changes will be committed to the database.

6.1.3 Lazy Loading

Lazy loading is implemented, so that relevant data is loaded only when it is needed. This reduces the startup time during initialization, which helps with improving the responsiveness of the system. Data is loaded only when it is requested.

6.1.3.1 Design Decision

There are four types of Lazy Loading, namely

- Lazy Initialization
Objects are set to null with lazy initialization. When the object is requested, it is fetched from the database and returned.
- Virtual Proxy
A virtual proxy is an object that shares the same interface as a real object. It is loaded when one of its methods is called.
- Ghost
A ghost is the object that is loaded in a partial state, where only the identifier is loaded. The rest of the object is loaded when one of its properties is accessed.
- Value Holder
A generic object that is placed in data fields to handle lazy load behavior

We opted to implement lazy initialization as our lazy loading strategy. Since in all our use cases, null is not a legitimate value to be held in our database. However, it is worth noting that should null fields be allowed in the database, a different lazy loading strategy must be introduced. Otherwise, the object that is being loaded could be stuck in an infinite loop where it is fetching a null from the database.

6.1.3.2 Implementation

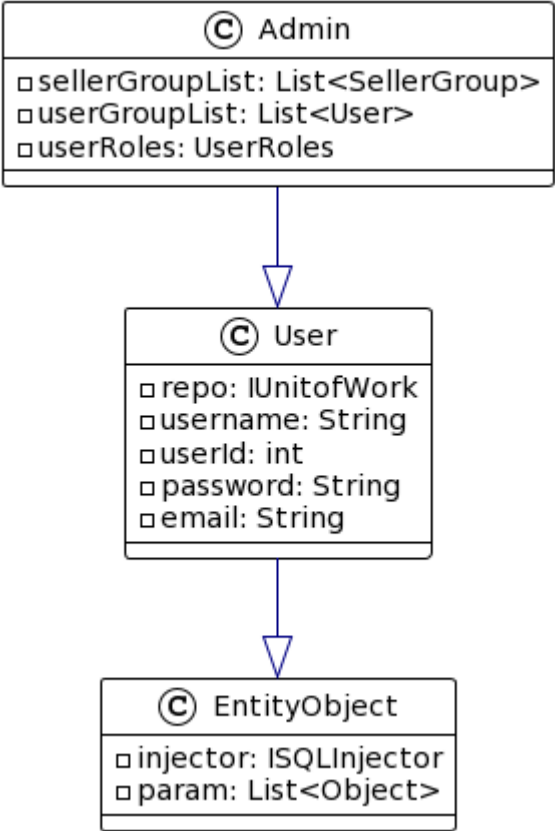


Figure 16: Admin class diagram

Using the Admin domain object as an example, on startup, username and userid will be set since it is crucial for differentiating the admin from the rest of the users. The other fields, such as sellerGroupList, userGroupList are set to null and will be initialised only when these values are requested.

6.1.3.2.1 Lazy loading Data Retrieval

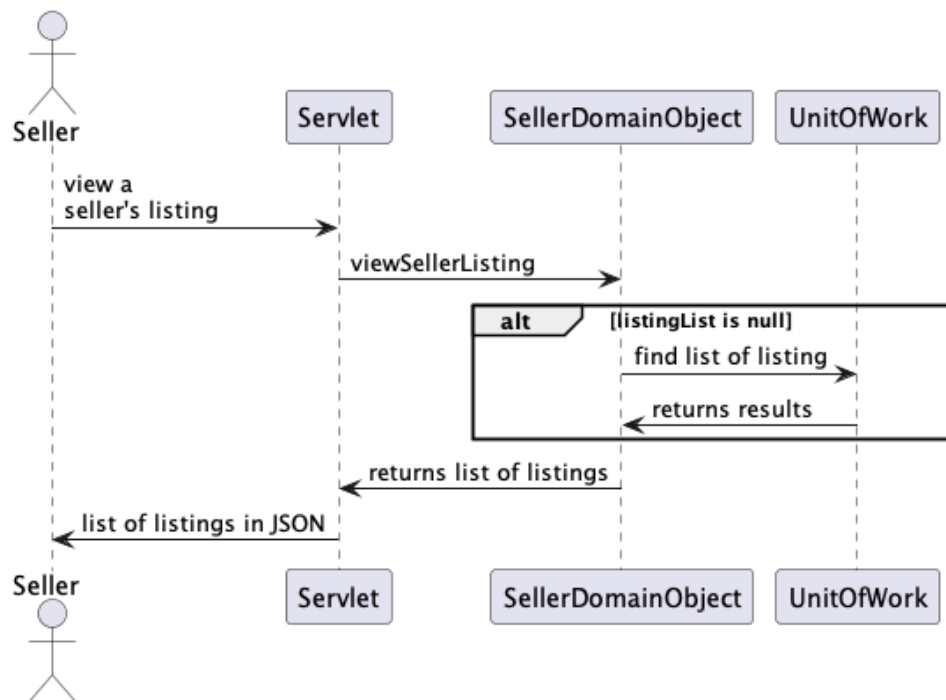


Figure 17: Lazy loading

Take a seller viewing the list of listings that he controls as an example; lazy loading happens when the Seller domain object is looking to retrieve a list of listings but finds it to be null. Upon realizing that the list of listings in the Seller domain object is null, the domain object initiates a database read to get database records of the list of listings that he controls via the Unit of Work.

Otherwise, if the list of listings is already initialized from an earlier operation, then the results will simply be returned without the database being read.

6.1.4 Foreign Key Mapping and Associated Table Mapping

Refer to 4.1 for detailed discussion on where foreign key mapping and associated table mapping is used.

6.1.5 JWT Tokens

JWT Tokens are used as a means of managing sessions in our system. The decision to utilize JWT Tokens, instead of session cookies is driven by two main reasons.

One, the stateless nature of the JWT token. After a successful login, a JWT token with user data embedded in it is generated. At every subsequent request, the server simply needs to validate the secret key used to sign the JWT token. This means that we no longer need to maintain sessions in the server, which contributes to reduced reads onto the database, leading to a lower response time.

The second reason for the use of JWT tokens is the ability to embed claims into the token. This enables us to include commonly read data, such as user ID, role, and group ID into the token during

token generation. This further reduces the number of reads performed on the database. If the secret is secured, any changes made onto the token can be detected and the token is invalidated.

6.1.5.1 Implementation

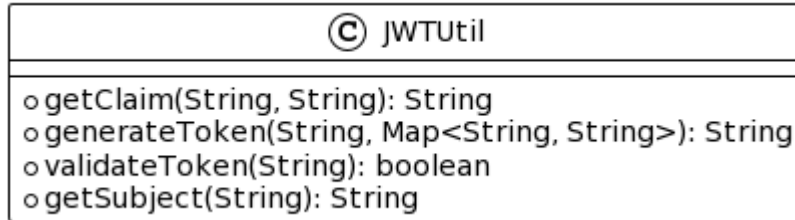
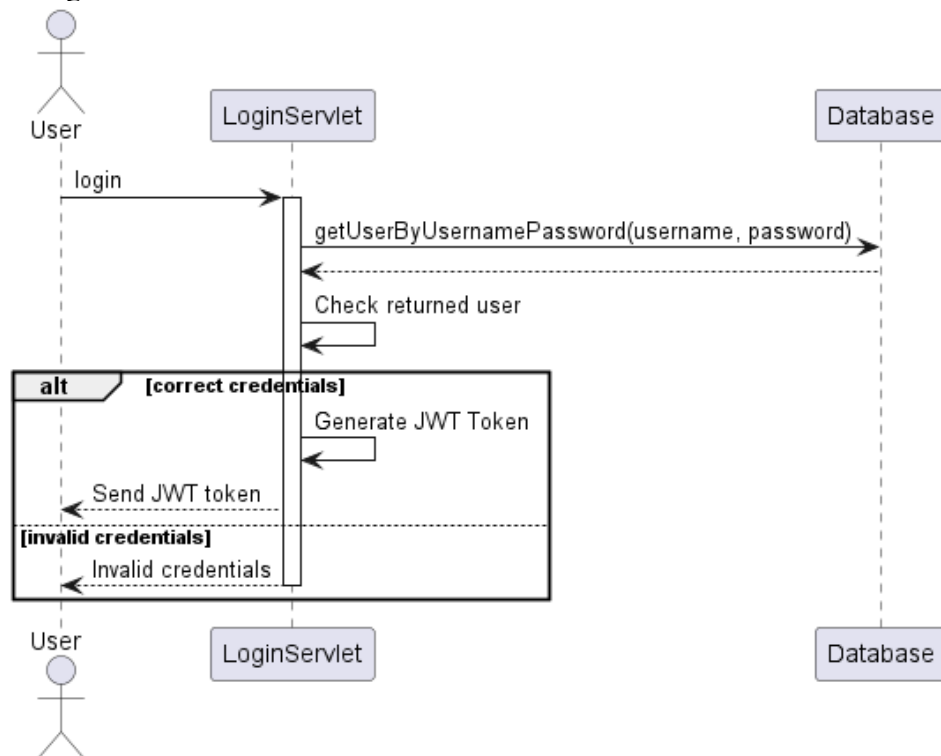


Figure 18: Class diagram of JWT service

JWTUtil is implemented as a service object whose only purpose is to support operations for the generation, validation, and data extraction from JWT Tokens. Since the generation of the token relies on the secret key stored in the server and has no need for an instance, JWTUtil is implemented as a singleton.

6.1.5.2 Operation

6.1.5.2.1 Setting JWT Tokens



Claims	Description
Issuer	The UID of the server issuing the token
Subject	The user ID of the token bearer
Issued at	Time of issue (microseconds)
Expiration	Token expiration time

Role	User role
Group ID	Sellergroup ID (if relevant)

Table 2: JWT claims

The table above outlines the claims that need to be set in a JWT token. In our implementation, tokens automatically expire an hour after it is issued.

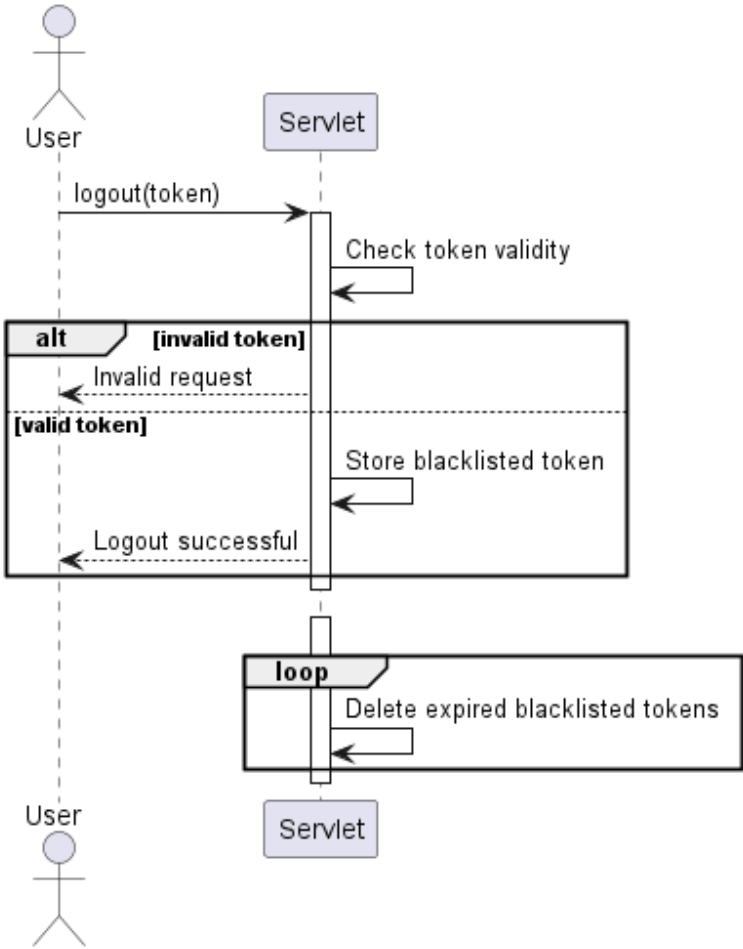
6.1.5.2.2 Authentication / authorization

We authenticate / authorize actions in our system by attempting to parse the JWT token with our Secret key. If the JWT token cannot be parsed properly, an exception is raised, and a failure is notified by the Unit of Work class. Eventually, all changes are rolled back.

6.1.5.2.3 Claim usage

The JWT class offers the ability to retrieve claims from the token via the getClaim() function. If the token is valid, a claim is returned. Otherwise, an empty string is returned. This functionality is used to record the user’s id, role and seller group id if the user is a seller. Doing so reduces the number of trips to the database, enhancing the responsiveness of our application.

6.1.5.2.4 Logout

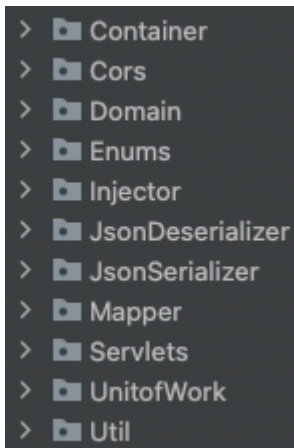


Since JWT Tokens are stateless, we cannot revoke the validity of the token once it is issued, unless the token expires. When a user logs out, the front-end interface removes the

token from local storage. In most cases, this is sufficient. However, to accommodate for cases where a token is reused, the JWT service additionally blacklists the token until it is expired.

6.2 Source Code Directories Structure

The source code in this project is arranged based on its functionality, as shown in the diagram below.



Folder	Description
Container	The dependency injection container used in Unit of Work
Cors	The definition of the CORS filter used by Tomcat
Domain	The domain object used in this project
Enums	The Enums that represent a group of constants used throughout the project
Injector	The SQL query injectors used in the find and delete operations
JsonDeserializer	Deserializers to deserialize JSON string to custom classes
JsonSerializer	Serializers to serialize custom classes to JSON strings
Mapper	Data mappers used
Servlets	The REST API end points consumed by the front end
UnitOfWork	The Unit of Work implementation
Util	Utility functions used in the project.

7. Physical View

The section describes the deployment strategy that the team has adopted for this project.

7.1 Production Environment

The team has opted for two separate Heroku deployments, one for the frontend and another for the backend.

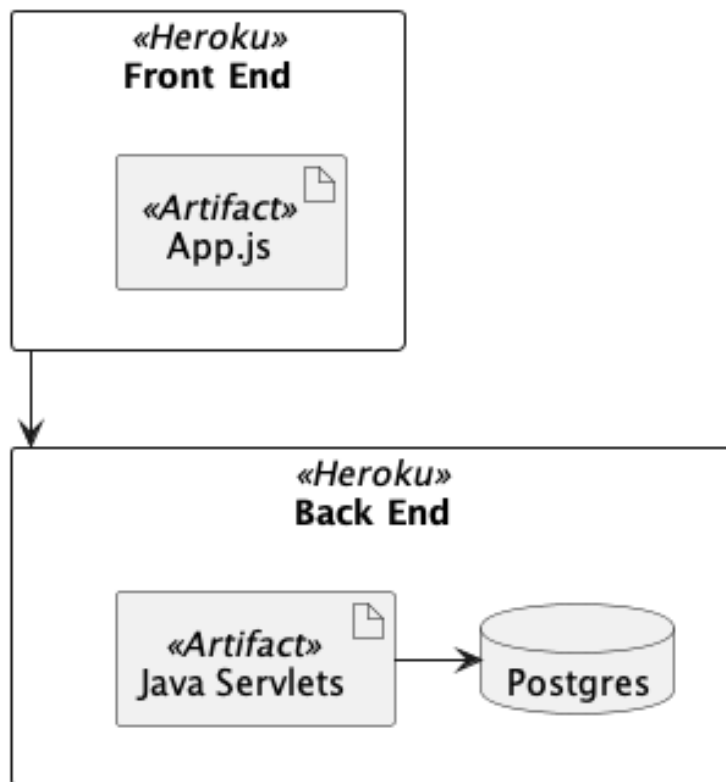


Figure 19: Production environment of the system

8. Testing Strategies

8.1 Testing Strategy

8.1.1 What do we want to test

There are two aspects of our system that we aim to test, the correctness of the software, along with the ability of the software to accommodate large numbers of requests.

The correctness of the software ensures that the system does not deviate from the specified business logic. However, since the system needs to accommodate concurrent requests, we need to maintain the correctness of the implementation in a concurrent environment as well. There is also a need for the system to be able to accommodate for a large flow of traffic into the site. We need to ensure that the requests are handled correctly and if a failure occurs, it fails gracefully so that the rest of the system can continue providing services to other requests.

8.1.2 Overall Scope

Since the frontend simply takes data from the backend and displaying it with no business logic implemented there, concurrency testing will not be performed there in this project, due to the complication of having to setup two different testing setups in a short time frame. Instead, manual testing will be performed there.

The backend, on the other hand, holds the business logic and controls the flow of data into the database. It is also affected by concurrency issues, so it is subjected to all two stages of the testing outlined below.

8.1.3 How do we plan on testing it

There are two main steps to us testing our system.



Figure 20: Steps in testing

First, we test the system without concurrency (single-threaded) with unit tests via JUnit 5. Doing so allows us to validate the correctness of the implementation without having to worry about the complications caused by concurrency.

Once we are confident of the correctness of our business logic, we will perform load and integration tests to test the system. Using JMeter, we aim to stress test our system, as well as to ensure that transactions are handled correctly in a concurrent environment.

8.1.4 Unit Testing Scope

The main aim of the unit testing will be to ensure that business logic is implemented as specified. Therefore, a lot of unit tests will be performed on the domain objects. For the other software patterns, such as the mappers and unit of work, they hold little business logic in them. Any implementation errors will be left to be discovered in the integration testing phase.

8.1.5 Load and Integration Testing Scope

We introduce concurrency in integration testing here. Since implementation logic is dealt with in the unit testing stage, the goal of this round of testing is to ensure the stability of the system when handling traffic as well as the test the behavior of the system. We also aim to use load testing to ensure that the concurrency issues outlined in Section 5 have been addressed.

To cover our bases, all API interfaces hosted by the servlet will be manually tested by a team member using Postman, as well as through the front-end interface. This ensures that the software patterns that were not tested in the unit tests are able to operate as specified.

At the current stage of the project, several use cases will be skipped from load testing. These are use cases that were outlined in Section 5.2 (operations unaffected by concurrency). In addition to that, creation operations without a duplicate object check, such as `createListing()` and `createBid()` will be excluded as well. This decision is driven by the much lower consequence of an incorrect entry. An incorrect listing can be removed by the owner, while a lower-priced bid would simply be ignored by the system. On top of that, a large amount of data could be written to the dev database by these create

functions. This presents an issue to developers with laptops on lower specifications as test data quickly fill up their database.

Instead, we aim to specifically validate the behavior of the 5 use cases suffering from concurrency issues, as outlined in Section 5.1. JMeter will be the tool of choice for running the tests. At least one request will be true. The same holds true for an invalid request. This enables us to observe the behavior of the system when dealing with correct and incorrect requests interleave with each other. This could expose more bugs that the team needs to fix.

8.2 Results

8.2.1 Results from unit test and multithreaded unit test

The section below outlines the result of the testing that was done on the system.

8.2.1.1 Jacoco Report





























Element	Missed Instructions	Cov.	Missed Branches	Cov.
Listing.java		43%		23%
User.java		45%		35%
Order.java		79%		61%
SellerGroup.java		52%		0%
GroupMembership.java		22%		n/a
OrderItem.java		47%		0%
FixedPriceListing.java		26%		0%
AuctionListing.java		53%		16%
Admin.java		81%		71%
EntityObject.java		17%		n/a
Bid.java		75%		n/a
Customer.java		95%		66%
Filter.java		65%		n/a
Seller.java		98%		69%
Total	667 of 1,942	65%	102 of 181	43%

Figure 21: Jacoco report for tests around our domain objects

Overall, we have covered 65% of the instructions and 43% of the branches within our business logic. A lot of missed instructions and missed branches are from the getters and setters that hold little to no business logic.

8.2.1.2 Final Maven test report

```
[INFO] Tests run: 17, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.379 s - in AdminTest
[INFO] Running SellerTest
[INFO] Tests run: 15, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 s - in SellerTest
[INFO] Running CustomerTest
[INFO] Tests run: 13, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.02 s - in CustomerTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 45, Failures: 0, Errors: 0, Skipped: 0
```

Figure 22: Maven test report

For a more detailed description of the tests run, refer to the testing document.

8.2.2 Results from JMeter test

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received K...	Sent KB/sec	Avg. Bytes
Register	110	2	1	51	7.17	0.00%	4.8/min	0.02	0.03	267.0
customer	110	6	1	128	17.71	0.00%	4.8/min	0.04	0.03	455.0
checkout	110	5	1	115	14.81	0.00%	4.8/min	0.02	0.05	267.0
modifyOrder	110	4	1	105	14.09	0.00%	4.8/min	0.02	0.05	267.0
Add seller t...	110	5	2	107	12.17	0.00%	4.8/min	0.02	0.05	267.0
add seller ...	110	5	1	114	15.23	0.00%	4.8/min	0.02	0.04	267.0
admin	110	1	0	14	2.19	0.00%	4.8/min	0.04	0.03	455.0
seller	110	1	0	13	1.65	0.00%	4.8/min	0.04	0.03	455.0
delete listing	110	7	3	117	11.78	0.00%	4.8/min	0.02	0.04	267.0
get purcha...	110	44	22	159	17.16	0.00%	4.8/min	0.02	0.02	321.0
TOTAL	1100	8	0	159	17.58	0.00%	47.5/min	0.25	0.35	328.8

Figure 23: JMeter test results

Parameter	Value
Number of threads (users)	100
Ramp-up period	1
Loop count	2

Figure 24: JMeter parameters

Overall, the system does not have any long running requests, with get functions taking up the longest time on average. No errors have been reported when running with the parameters outlined above. A more detailed description of the tests done in JMeter can be found in the testing document.