Architecture Document

SWEN90007

Marketplace System

Team: Team0

In charge of: Lai Wei Hong 1226091

Sean Wong 885710

Andrew Liu 1084190

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

Contents

OBJ

OBJ

OBJ

OBJ

OBJ

OBJ

OBJ

OBJ

OBJ

OBJ

OBJ

OBJ

OBJ

OBJ

OBJ

OBJ

OBJ

OBJ

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.

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.

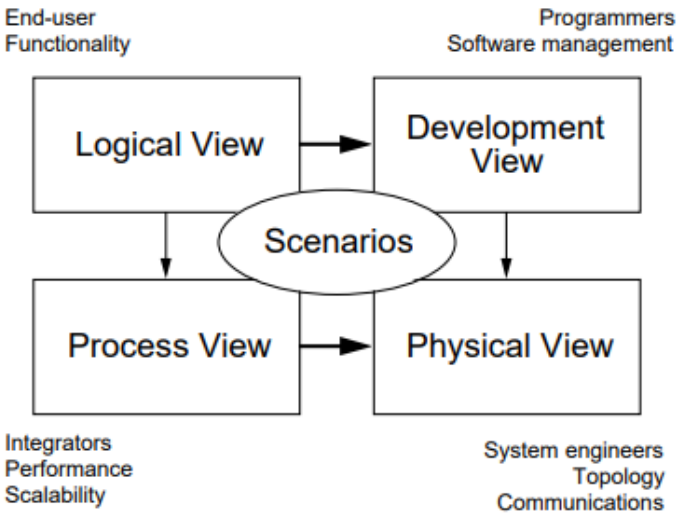


Figure 1. Views of *framework* "4+1"

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

3. Architectural Objectives and Restrictions

The defined architecture's main objective is to make the system...

Since we are 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 rate. The defined architecture should be able to handle many requests during its operation. The front-end user interface should be easy to navigate for a new user and has a clear separation between the interfaces of different users of the system.

However, several restrictions were placed on the team when developing the system. A restriction is placed on the tools/libraries that can be used when developing the system. This forces the team to implement and test several important architectural patterns from the ground up. This will increase the lead time of the development process.

3.1 Architectural Significant Components

This section outlines components within the system that has architecturally significant components.

Component	Impact	Treatment
Data Insertion and Modification	Defines the way data is written to the database	Data mappers for each table in the database will be implemented to reduce the coupling of the business logic and the database interactions
Session Management	Requires a way to keep track of the user's session	JWT tokens are implemented to indicate that the user has been logged in
User Role Identification	Role and seller group data for a user is frequently read	Role and group id (for sellers) is incorporated into the JWT token, reduce the number of duplicate reads from the database

User Input Data	Objects received from the front end may be null	Lazy loading is implemented, so that the entity object fetches relevant data if the object id is set.
Data display at the front-end UI	Defines the way data is presented to the user	Lazy loading on the database reads using the LIMIT and OFFSET clause in all reading operations
Data deletion	Defines the behaviour when a primary key is deleted from the database	Foreign keys that reference the primary key are deleted on cascade, so that there will not be any dangling keys in the database
Finding and Deleting data	There is a need to accommodate a large variety of SQL queries when finding data	Utilize dependency injection to inject SQL queries in runtime to the find operation in the data mappers

4. Logical View

This section shows the system's organisation from a functional point of view. We will be specifying the relationship between different entities in our database. A high-level component diagram specifying the different interactions between components will also be specified. This component diagram will then be broken down further into modules and its corresponding functionality.

4.1 Entity Relationship Diagram

Figure 2 Illustrates the entity relationship diagram that the team has used to model our database.

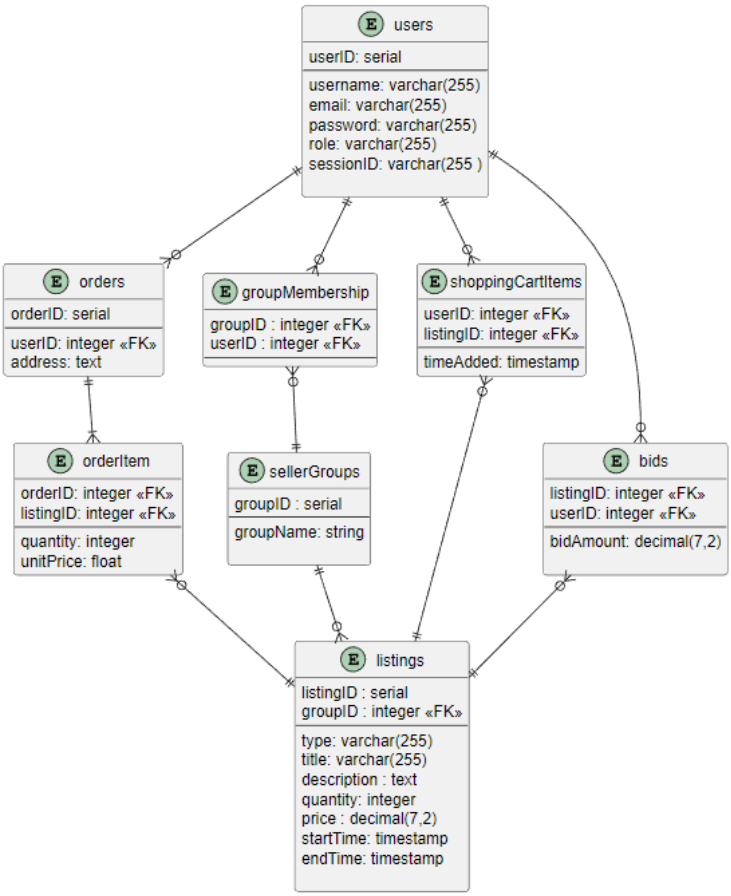


Figure 2. Marketplace Entity Relationship Diagram

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. The controller are the buttons that are present in the webpage that allows the user to interface with the system. The controller will update the model based on the inputs from the controller. When the model has been updated, it pushes a change to the view based on the updated state.

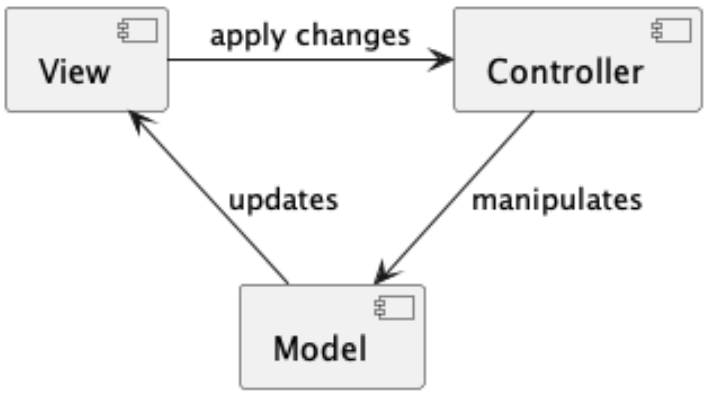


Figure 3. Generic Model View Controller Architecture

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.

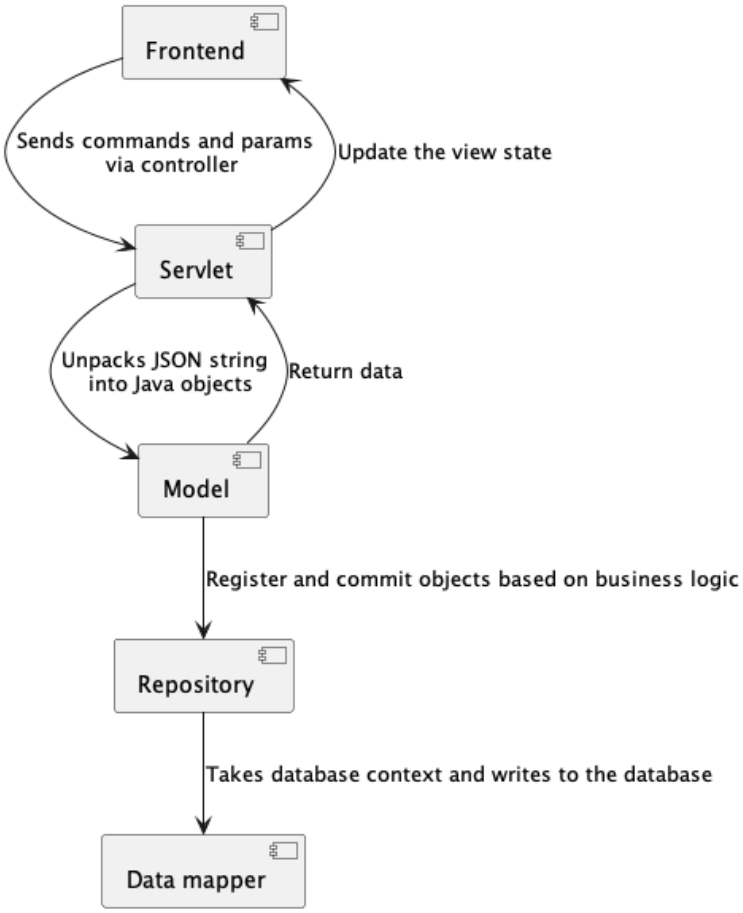


Figure 3. Marketplace System Overview

The following subsections outline the responsibilities held by each component in the system and the components it interacts with.

4.2.1 Front End Interface

Responsibilities	The front-end interface is used to render the components that the end user will be interacting with. It also renders buttons, which acts like a controller that allows the user to interface with the backend system.
Interacts With	Servlets

4.2.2 Servlet

Responsibilities	The servlets act as a medium between the front-end controllers and the model. It is responsible for the serialization of JSON objects.
Interacts With	Model, Front-End Interface

4.2.3 Model

Responsibilities	The model is where the business logic for the transactions lies.
Interacts With	Repository, Servlet

4.2.4 Repository

Responsibilities	The repository registers changes made by the business logic in the model component. This can either be a create, update, or delete. At the end of a business transaction, the changes are committed into the database via the data mapper.
Interacts With	Model, Mapper

4.2.5 Mapper

Responsibilities	The mapper acts as an abstraction layer to separate database logic from the rest of the system. It handles all the reading and writing of data into the database
Interacts With	Repository

4.3 Class Diagram

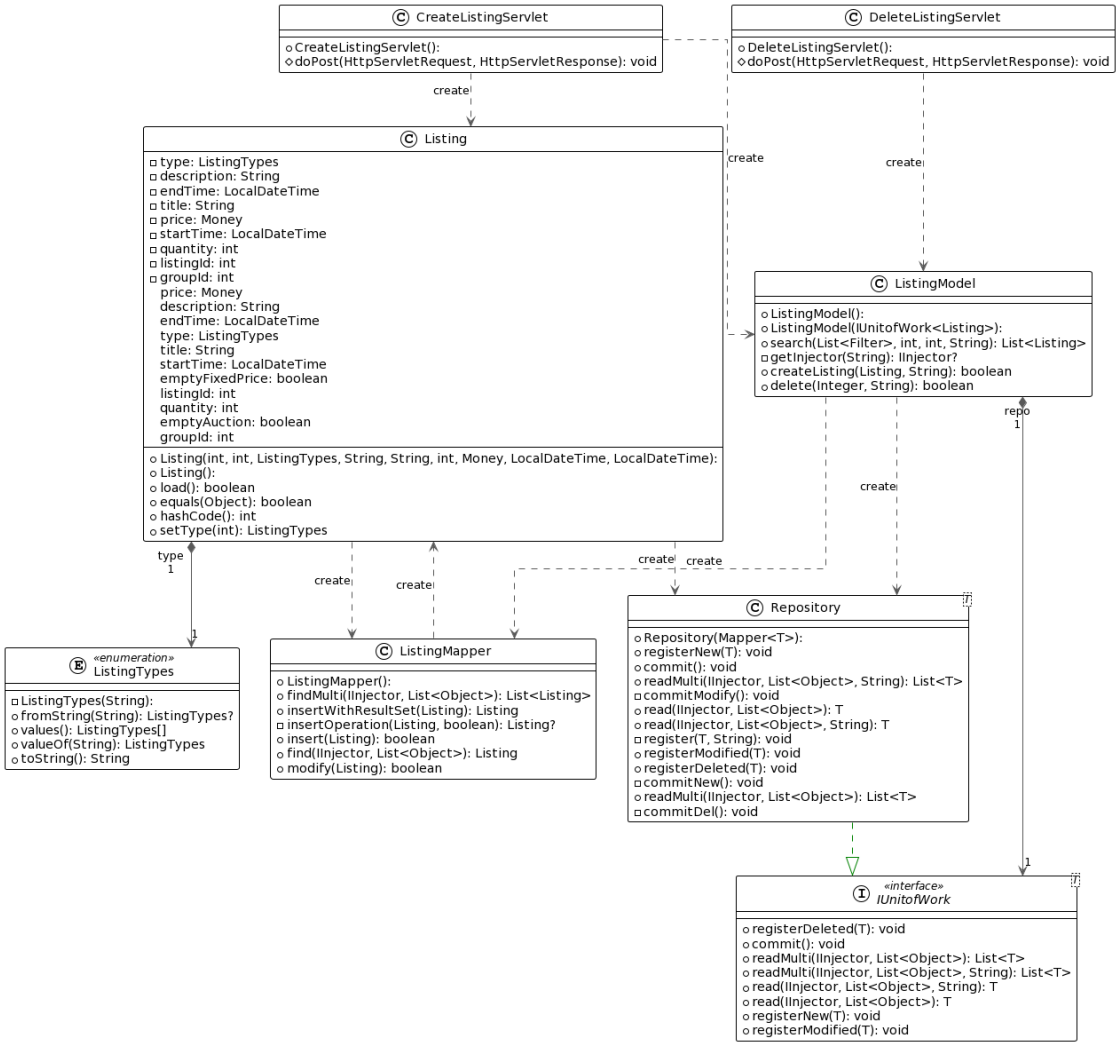
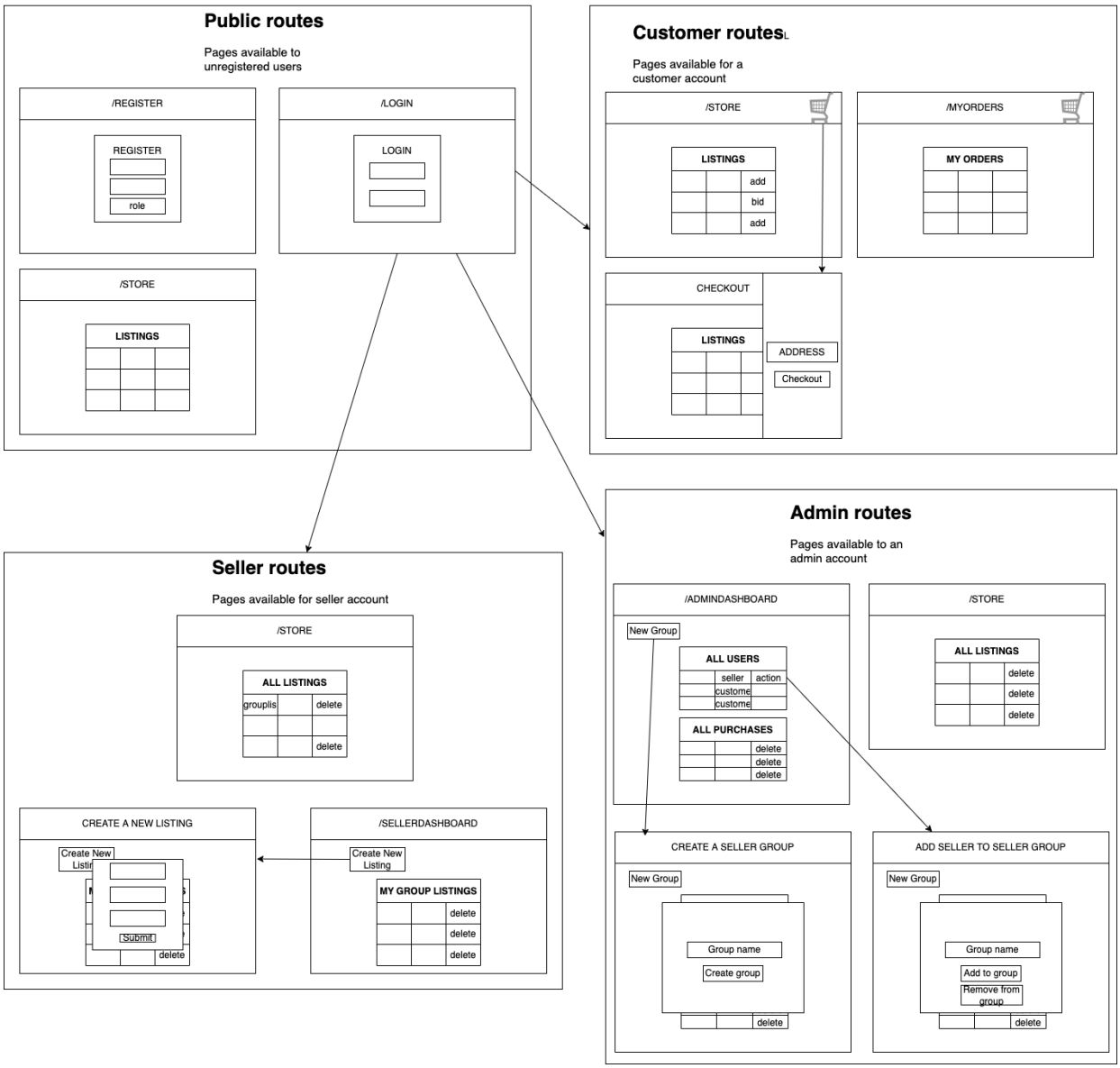


Figure 4: General Class Diagram

Due to the substantial number of classes in this project, a complete class diagram encompassing all the classes that we have developed is unlikely to be helpful due to its complexity. Instead, we opted to present a snapshot of the structure of our class that can easily be replicated for newer features.

The first point of contact from the front end is the servlet, which in the diagram above are the CreatingListingServlet and DeleteListingServlet. The servlet processes the JSON strings into relevant Java objects. It then feeds the data to a relevant model to carry out the relevant business logic. Here, the ListingModel will interface with the Repository class to register, read, and commit data into the database. The Repository will interface with a mapper that is attached to it during initialisation to perform the reading and writing process.

4.4 UI Layout

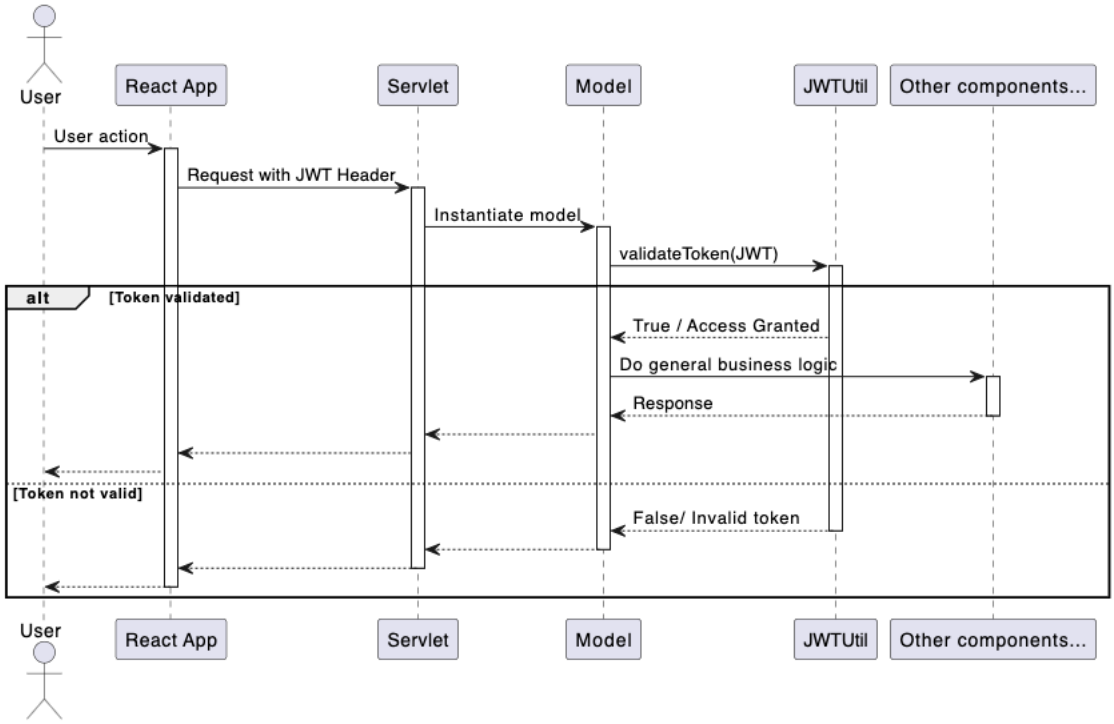


5. Process View

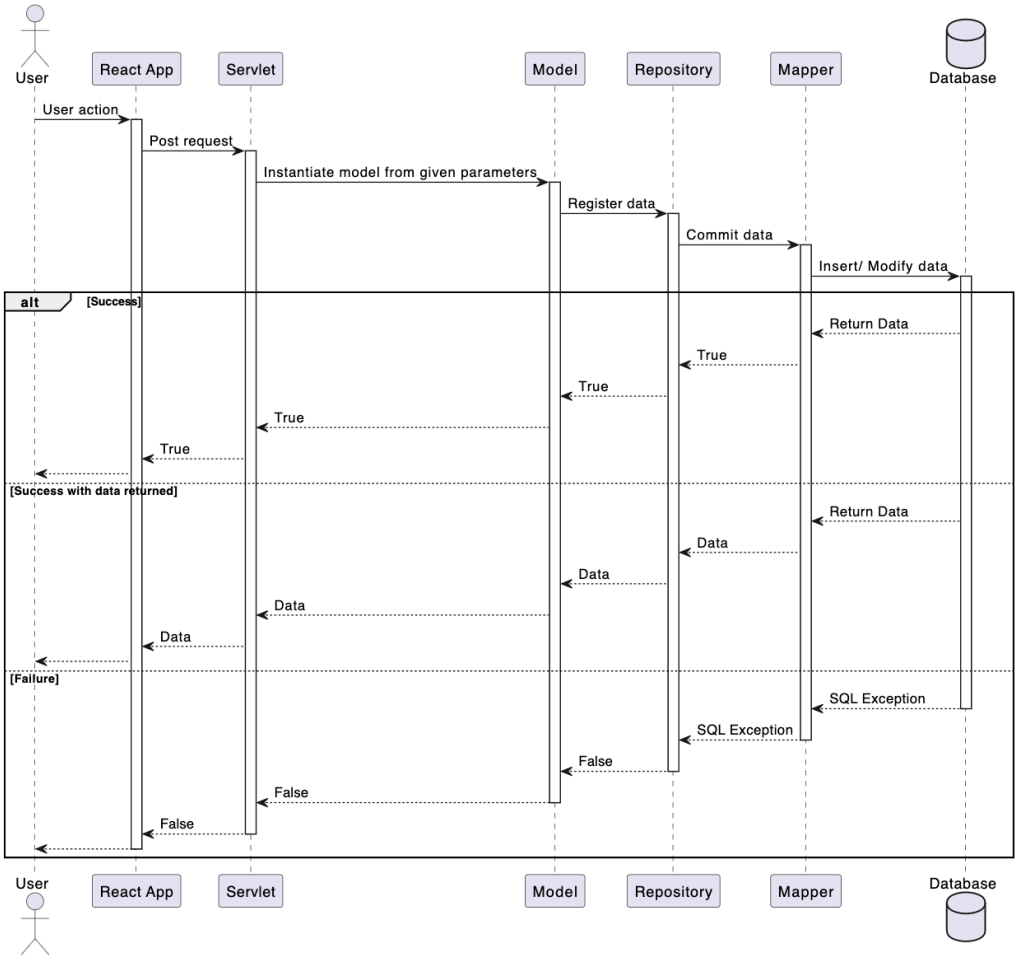
At this stage of project, we are only concerned about the developing the system for the use of a person. Therefore, elevated levels of concurrency were not considered as of the time of reporting. This section will be focused on the runtime behaviour of the major key components of the system, such as the CRUD operations, authentication and the lazy loading process used in the entity objects.

5.1 Sequence Diagram

5.1.1 CRUD Operations



5.1.2 JWT authentication



6. Development View

This section provides orientations to the project and system implementation in accordance with the established architecture. **Error! Reference source not found.** illustrates the implementation view of system architecture.

6.1 Architectural Patterns

Pattern	Reason
Domain Model	A domain model allows us to have a one-to-one relationship with the records in the database tables. This helps give structure when dealing with complicated business logic that might involve multiple entities.
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 is used as a facilitator to other architectural patterns that we have implemented, namely the identity field and dependency injection. It also allows us to coordinate database operations between different repositories. This helps prevent dangling records in the event of failure in business logic at the domain level.
Lazy Loading	It is infeasible to load the data from the entire table in a read operation. Lazy loading allows us to only load data that is relevant for our use cases.
Identity Field	Identity maps allow us to get access to a context-specific, in-memory cache that retrieves data faster than fetching it from the database.
JWT Tokens	JWT Tokens are used to keep track of the user's session, as well as to provide user context specific information, such as user id, group id and role. This helps 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 the relevant SQL query in runtime to the read function. This also helps reduce the coupling between the database and data mapper.

6.1.1 Domain Model

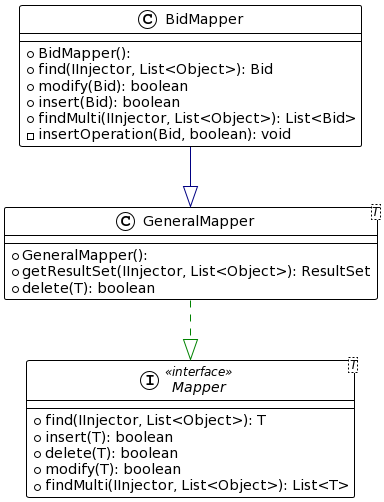
There are several entities that need to be represented in our business logic. A list of the entities is shown in the table below.

Entity	Description
Bid	A Bid represents a customer's price offer for an Auction listing. The highest bidder by the end of the listing duration will automatically have the item added to their order list.
GroupMembership	A Group membership represents a user's/users' belonging to a seller organisation. A seller must be associated with a group to create a product listing and a member can only belong to one seller group. This association is represented by a Seller's User ID and an existing Group ID. Only the Administrator can manage the addition and removal of members within a seller group.
Listing	A listing represents a product being sold by a seller organisation. It can either be an Auction listing or a Fixed price listing. There can be multiple items in a single listing. A Fixed price listing can directly be added to Customer user's cart and be checked out. whereas an Auction listing will have a limited duration where customers can place bids. Listings can be deleted by members of the listing's seller group or by an administrator.
Order	An order represents a customer's purchase after checking out. An order can consist of multiple listings of different quantities for each item and is associated with a single order ID. An order also requires a customer's address and will be linked to the customer's User ID
OrderItem	An OrderItem represents a single listing within an order. It can have more than one item and is associated with an Order ID.
SellerGroup	A SellerGroup represents a real-world seller organization. To be able to post a listing in the online marketplace, a seller must be a member of the group. (GroupMembership). Every listing is represented by a seller group and any members of the seller group can delete and modify the listing
User	A User represents each user in the system and can have the roles of Customer, Seller and Administrator.

6.1.2 Data Mapper

Data mappers has been used to abstract the database logic away from the domain models where the business logic lies. Every table in the database is matched 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.

6.1.2.1 Examples of Use



The image above is the class diagram of our data mapper for bids. The Bid Mapper implements the functions defined in the Mapper Interface. It also extends the General Mapper class which contains two common functions that is shared across all data mappers, getResultSet(injector, param) and delete(T). Both the common functions utilise dependency injection to inject relevant SQL queries for the find and delete operations.

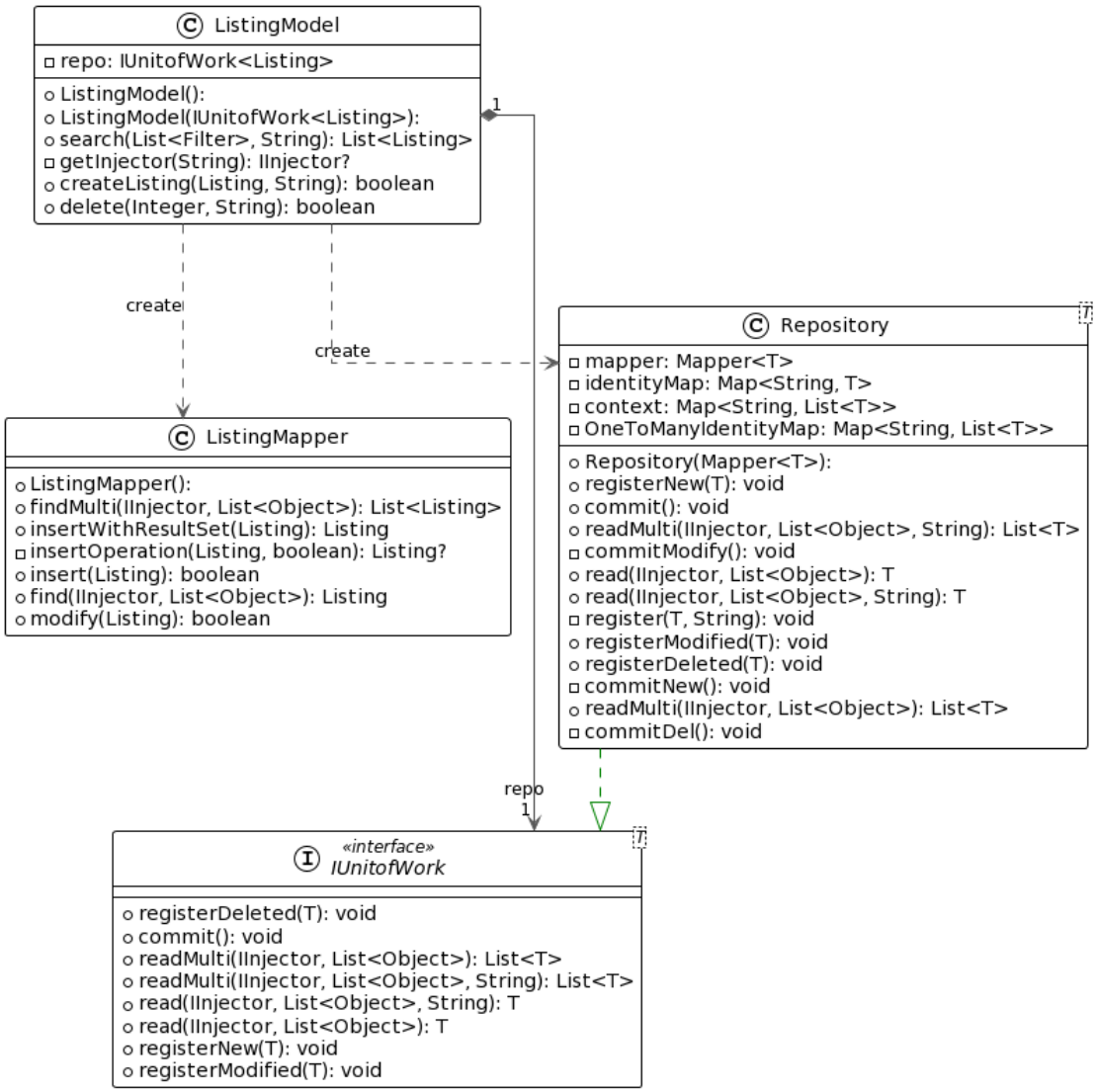
6.1.3 Unit of Work, Identity Map & Dependency Injection

We are utilising the unit of work pattern between the domain model, where the business logic lies and the data mappers. The unit of work in our solution is used as a facilitator for other architectural patterns that we have implemented, namely the identity field and dependency injection. It also allows us to coordinate database operations between different repositories, which can be crucial in preventing dangling records in the event of failure during a transaction.

Since all the data used in our system will pass through the repository, it is a good place to place the identity map. The identity map will behave as a context-specific, in-memory cache. When using the identity map, the unit of work will first check if there is an existing instance of the data currently stored in the identity map. If such an instance is not found, then the unit of work will initiate a database call to read the required data, which is a lot more expensive.

The dependency injection pattern is used in the find and delete operations, where there is a larger variety of find and delete conditions. The dependency injection pattern would allow us to inject the relevant SQL query in run time into the Repository class, which will then pass it on to the mapper for the relevant operations. This allowed us to implement a general find and delete function, which helps reduce code dependency to specific mapper implementations.

6.1.3.1 Examples of Use



The image above is the class diagram of the ListingModel, the ListingMapper and the Repository class. Here, we aim to illustrate the relationship between the unit of work and the domain model, along with the mapper. Take a read operation as an example, when the listing model searches for listings based on a set of filtering criterion, the domain model selects and injects the suitable SQL query into readMulti() in the Repository class based on the filter conditions provided. Since there are no read operations, there are no changes for us to register and commit. After the repository queries the data mapper for the results, it keeps a copy of the data in its identity map, in anticipation of future use.

6.1.4 Lazy Loading

There are two instances where lazy loading is used in our application, which are after JSON deserialization from the user inputs and when data is loaded from the database, particularly in the admin dashboard where site-wide data is collected.

In the first instance, we use the Lazy Initialisation pattern. When the system accesses a null field, we either calculate the relevant values or the load the data from the database. In the second instance, we place a LIMIT and OFFSET clause in every SQL query, so that we limit the number of results returned and only return data that is the most relevant to the user.

6.1.4.1 Examples of Use

```
if(bidAmount == null && bidAmountInCents != 0) {
    setBidAmount(
            Money
                    .of(getBidAmountInCents(), Monetary.getCurrency( currencyCode: "AUD"))
                    .divide( divisor: 100)
    );
```

The image above is an instance of Lazy Initialisation in the Bid object. Since the bid amount is passed to the model in the cents, some calculation is needed to convert it into the Big Decimal format. However, since the bidAmount is not always needed, we can leave it as null.

```
public class FindOrderWIthGroupId implements IInjector {
    @Override
    public String getSQLQuery() {
        return "SELECT oi.* FROM orderitems oi " +
                "JOIN listings l on oi.listingid=l.listingid " +
                "where orderid=? LIMIT ? OFFSET ?;";
    }
}
```

The image above is an instance of LIMIT and OFFSET clauses used in SQL query to limit the number of order items in display. The LIMIT clause defines the number of order items that are listed, while the OFFSET clause represents the page number that the user is currently looking at. Doing so allows the user interface to only fetch the most important data.

6.1.5 Foreign Key Mapping, Associated Key Mapping & Embedded Value

Foreign key mapping is used throughout the system to allow us to form associations between objects with reference to a foreign key. In our application, we use foreign key mapping to form and persist strict relationships between users and the different entities in our system. An example would be the relationship between a user and orders where an order holds the user's id as a foreign key. This allows for the one-to-many relationship as users can consist of many orders.

The association table mapping is used to represent a relation between two objects in a single table using a pair of foreign keys. As an example, the groupmembership table represents a seller's membership to a seller group. This was also done as both user and seller group should be able to exist independent of each other.

Embedded value is another database pattern that was used heavily in our project. We do so by matching the entries in the tables with the fields in our domain object. This allows a seamless connection when

loading and writing data to and from the database. Patterns of this can be observed in all the objects in our Entity folder.

6.1.6 JWT Token

JWT Tokens are implemented in this system, instead of the typical session cookies. This is done due to two reasons. One, it reduces the number of reads from the database. Since we no longer need to reach into the database to validate a user supplied session token, we can reduce the number of duplicate reads. JWT Token can achieve this through encryption via a 256-bit key supplied by the user.

The second benefit that JWT Tokens bring is the ability to include claims within the token. This allowed us to include commonly read data, such as user id, role, and group id in the token. If the secret key used to sign the JWT is secured, we will be able to detect any changes to the token and invalidate it.

6.1.6.1 Examples of Use

JWT is used in two scenarios in our system, during login where the JWT token is generated and during normal operations where the token is used to validate the user's session.

```
90          Map<String, String> claims = new HashMap<>();
91          claims.put("role", user1.getRole());
92
93          // If the user is a seller, we include the seller group in the tokem
94          if(user1.getRole() == UserRoles.SELLER.toString()) {
95              param = new ArrayList<>();
96              param.add(user1.getUserId());
97              GroupMembership gm = gmRepo.read(new FindIdInjector("groupmembership"), param);
98              claims.put("groupId", Integer.toString(gm.getGroupId()));
99          }
100
101         // User exists here
102         // Generate jwt token for upcoming sessions
103         return JWTUtil.generateToken(String.valueOf(user1.getUserId()), claims);
```

The image above is taken from the user login function. When generating the JWT token, we included the role, group id and user id as part of the generated token.

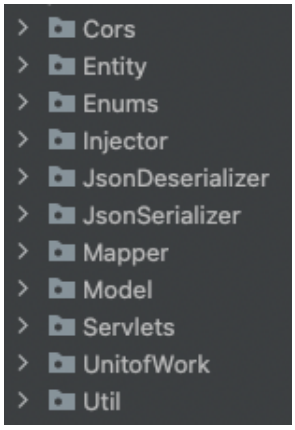
```
public boolean createSellerGroup(SellerGroup sg, String jwt) {
    // Get the role from the jwt
    String role;

    try{
        if(!JWTUtil.validateToken(jwt)) {
            // if not valid, return false
            return false;
        }
        role = JWTUtil.getClaim("role", jwt);
```

```
// Check to see if the role is empty or null or not admin
if(role == null || role.equals("") || !role.equals(UserRoles.ADMIN.toString())) {
    return false;
}
```

The second image shows an instance where the JWT is used beyond the use case of validating a user's session. Here, we extract the role from the JWT Token to ensure that the user has sufficient privilege to access the resource. This has helped reduce the number of reads onto the database for role verification.

6.2 Source Code Directories Structure

The source code in this project is arranged based on its functionality, as shown in the diagram below. The table following will describe the purposes of the folders.



Folder	Description
Cors	The definition of the CORS filter used by Tomcat
Entity	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
Model	Models used to define the business logic
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

This 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.

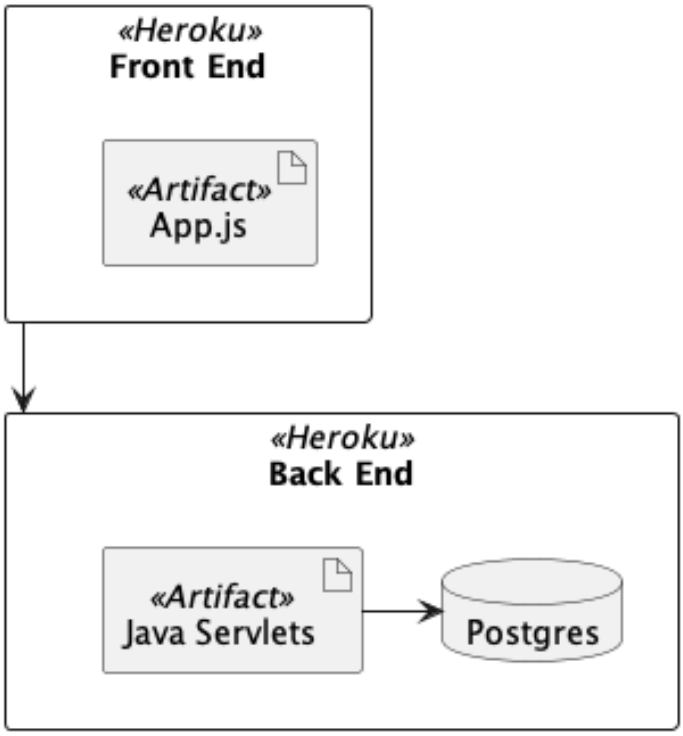


Figure 2. Production environment of the system

8. Scenarios

The use cases below outline the list of requirements that we hope to achieve by the end of the project.

Seller use cases

1. Create listing
2. View an order tied to seller group
3. Modify an order tied to seller group
4. Decrease quantity
5. Cancel an order

Buyer use cases

1. Search for a good
2. Purchase a good
3. View an order
4. Modify an order
5. Increase/decrease quantity
6. Cancel an order

Admin use cases

1. Onboard sellers
2. View all purchases
3. Create seller group
4. Add/remove seller to/from group
5. Remove a listing