Architecture Document

SWEN90007

TomcatBypass
Marketplace System
Team: Team 2

Name	UoM ID	UoM Username	GitHub	Email
James Hollingsworth	915178	jhollingwor	JaeholsUni	jhollingwor@student.unimelb.edu.au
James Vinnicombe	916250	jvinnicombe	jvinn	jvinnicombe@student.unimelb.edu.au
Sable Wang-Wills	832251	lawsonw1	saybb	sable.w@student.unimelb.edu.au
Thomas Woodruff	834481	twoodruff	twoodruff01	twoodruff@student.unimelb.edu.au

SCHOOL OF
COMPUTING &
INFORMATION
SYSTEMS

Revision History

Date	Description	Author
20/8	Created Document Scaffolding	James Vinnicombe
20/9	Rework Report Structure	Sable Wang-Wills
20/9	Add Pattern Discussion and Diagrams	Sable Wang-Wills, James Hollingsworth
21/9	Added Class diagram	James Hollingsworth
21/9	Reviewed & Made Several Changes	Tom Woodruff

Contents

1. Introduction

This document specifies the TomcatBypass Marketplace system's architecture, describing its main standards, modules, 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, emphasising 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 main 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.

1.4. Architectural Representation

The specification of the TomcatBypass Marketplace system's architecture follows a simple design-to-implementation series of views, beginning with class diagrams and high level views and narrowing down to more granular, more implementation-level views, such as directory structure and then environment descriptions. Then, the architectural patterns used are specified, supported by diagrams and justification where appropriate.

2. System Architecture and Implementation Details

This section shows the system's architecture and provides details about its implementation.

2.1. Class Diagram

The Figure provides a class diagram of the system.

[Click to view a higher resolution version of the image below]

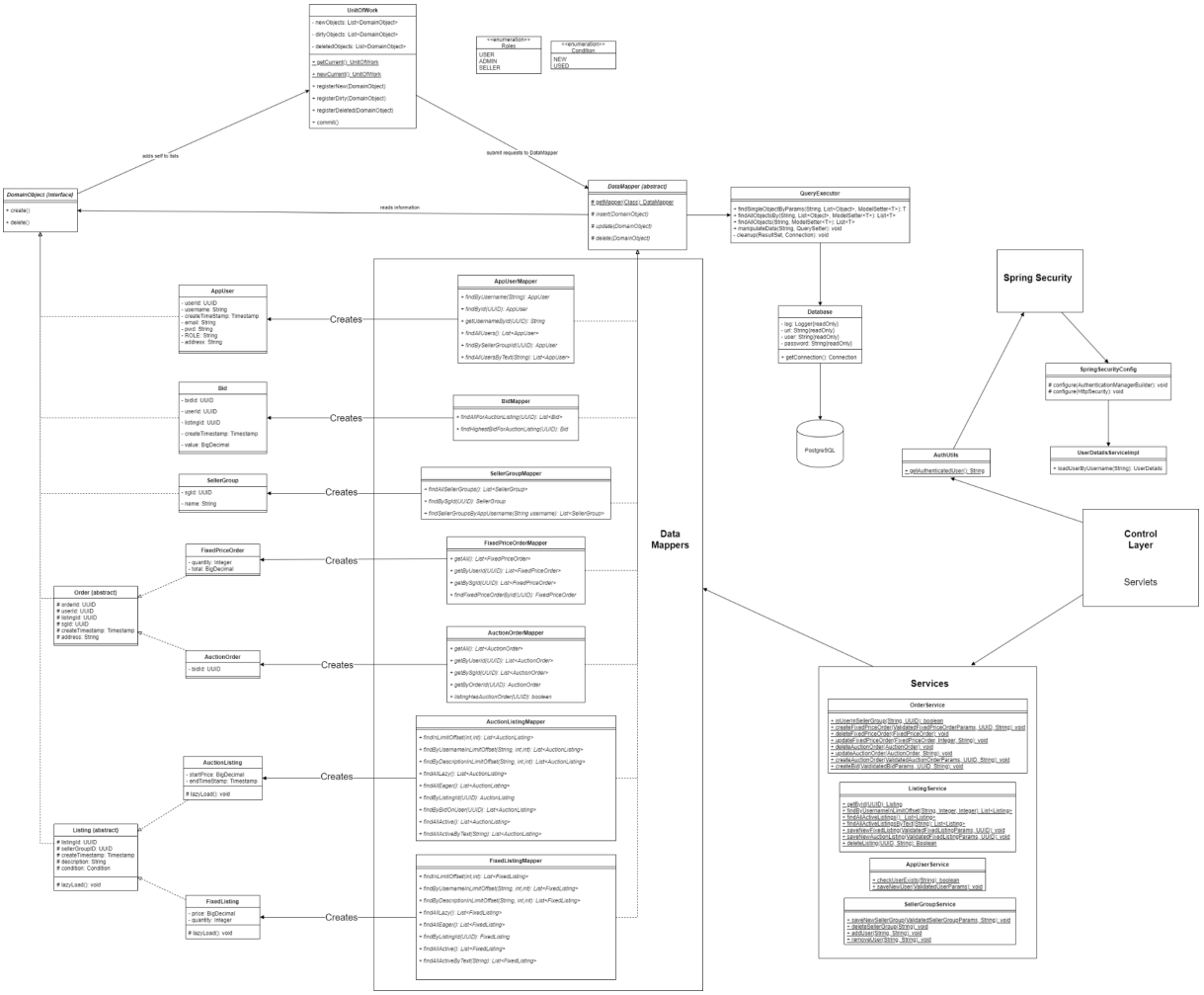


Figure: System Class Diagram

2.2 Database Diagram

The database specification is provided in the Figure.

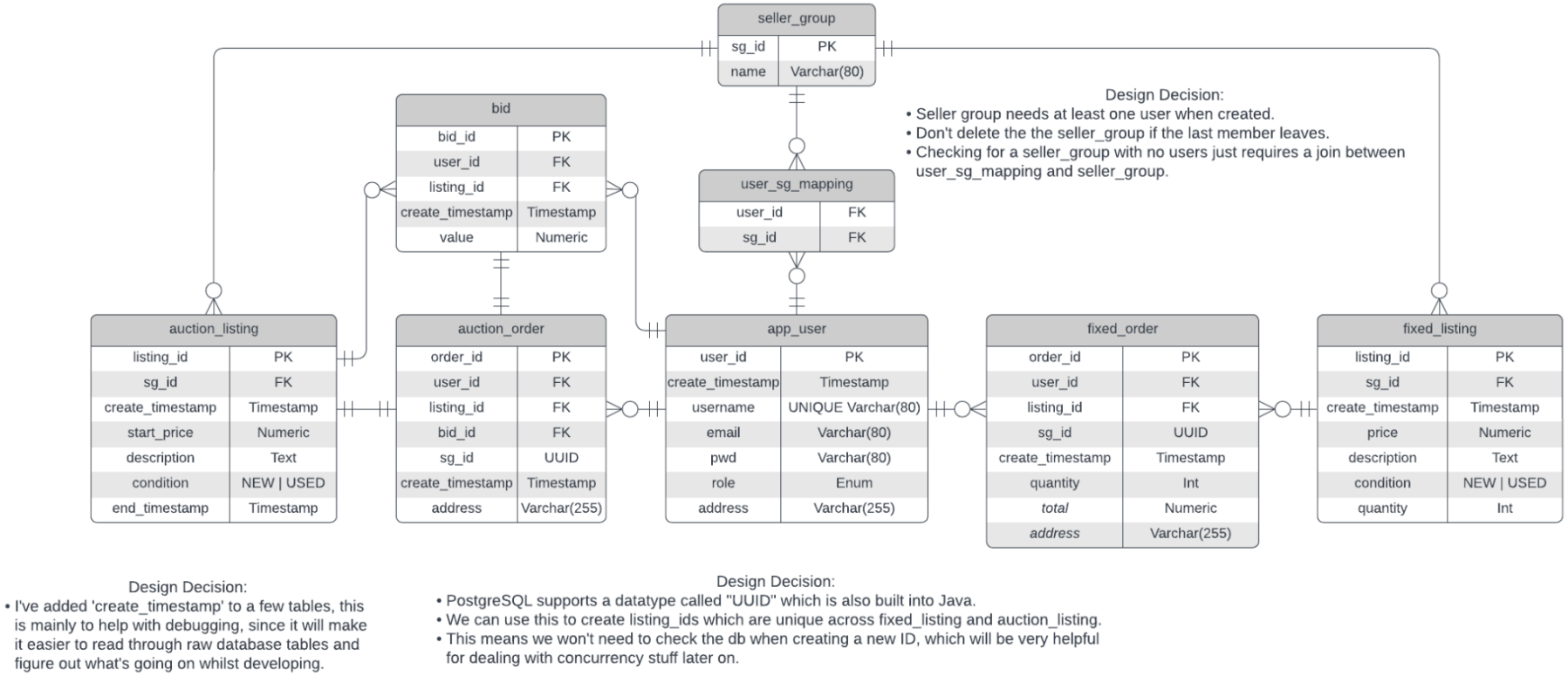


Figure: Database Diagram

2.3. Components

This section describes the main components of the system.

2.3.1. Data Base Layer

The Data Base Layer is responsible for all interactions between the Business Rules Layer and the database. All SQL statements are defined and executed within the Data Base Layer and passed back to the Business Rules Layer.

3.2.1.1 Mappers

Mappers function by preparing sql statements and returning model objects back to the Business Rules Layer. Mappers also function in the opposite way allowing classes to pass model objects to the Mapper to be updated or inserted into the database. The Mappers functions act as end points to be accessed by the Business Rules objects.

3.2.1.2 Utils

Utils function to provide database connectivity for the Mappers. The Utils work to open a database connection and execute queries for the Mapper classes.

2.3.2. Business Rules Layer

The Business Rules Layer is responsible for capturing the structure of system information through models and performing operations on these objects.The Business Rules Layer also interfaces with the Data Base Layer.

3.2.2.1 Models

Models act as objects to represent the information that is stored in the system in a logical and useful way.

3.2.2.2 Services

The services exist for 2 functions. Firstly perform any actions on model objects that are repeated and as such are created as methods. Secondly the services interface with the Data Base Layer to allow for other components to insert, update and delete model information in the database safely.

3.2.2.3 Enumeration

Enumerations exist to show representations of strict and limited data fields. These fields are then utilised to show aspects of model objects.

2.3.3. Presentation Layer

The Presentation Layer is responsible for displaying the correct information in the system to the user and ingesting user input. The Presentation Layer also interfaces with the Business Rules Layer.

3.2.3.1 Controllers

Controllers exist to ingest user input and to ensure that the correct information is displayed to the user. These controllers, implemented as servlets, interact with the Business Rules Layer to procure the correct information and direct the user to the intended Server Page ensuring that the Server Page has the procured information.

3.2.3.2 Server Pages

Server Pages exist to display the information they receive to the user. Through a combination of markup and scripts they ensure that the information is displayed correctly and clearly to

the user and allow for data ingestion and navigation to be directed towards the correct Controllers.

2.4. Source Code Directories Structure

The source code is contained within the src folder.

All Java is contained within java/com/unimelb/tomcatbypass and within this there are sub directories for different functional components of the java model.

All JSPs are contained within the webapp directory, these are organised based on their url pathway.

Additionally in webapp we have other web resources such as css, images and web-inf which contains the xml and jspf files.

```
src/main
└── java/com/unimelb/
|     └── tomcatbypass
|           ├── auth
|           ├── control
|           ├── enums
|           ├── mapper
|           ├── model
|           ├── service
└── webapp  └── utils
      ├── auth
      |    ├── admin
      |    |    ├── listing
      |    |    ├── orders
      |    |    ├── sellergroup
      |    |    └── user
      |    ├── listing
      |    ├── orders
      |    ├── sellergroup
      |    └── user
      ├── css
      ├── images
      └── WEB-INF
            └── jspf
```

2.5. Libraries and Frameworks

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

Library / *Framework*	Reason	Version	Environment
Spring Security	Authentication and Authorisation handling throughout the application.	5.7.3	All
Lombok	Getters, setters, builders and other tags to simplify the visuals of Java models	1.18.24	All
Spotless	Auto Format code when git committing to increase legibility and standardisation	3.3.2	Dev
JSTL: Tag Library	Implementing tags within our jsp files to simplify functional logic for display purposes	1.2	All

2.6. Development Environment

The development environment will consist of an Intellij project, with codebase hosted on GitHub, and a local PostgreSQL instance. The system will execute via Tomcat, and is built using Maven.

The Intellij project will require a run configuration executed via Tomcat which has DB information - credentials and a link to it. While executing, the system can be accessed via localhost on a web browser. If on a unix machine, it can also be executed with the linux_dev.sh script.

2.7. Production Environment

The production environment is hosted on Heroku, with a PostgreSQL database. The system is accessible by web browser via URL.

This link is for the deployed application: https://tomcatbypass.herokuapp.com/.

This decision to deploy on Heroku was based on recommendation from the teaching staff. This takes care of HTTPS and other painful configuration issues.

3. Architectural Patterns

This section provides information about how architectural patterns are applied in the project and system implementation. Some patterns are described in more detail in separate sections, as they can be demonstrated with examples and diagrams.

Pattern	Reason
Domain Model	The system is to be implemented in an object-oriented language, and can feature some complex business rules. Requirements are fixed at a high level, but the details may change. It is natural to use Domain Model in this situation, in addition to the benefits of extensibility and reusability of code inherent to this model.
Identity Field	Required when using the domain model in order to maintain object identity between in-memory objects and database rows. Present in every domain object in the system.
Foreign Key Mapping	Required when using the domain model in order to maintain object identity between referenced objects and the appropriate database rows. Present in most domain objects in the system.
Association Table Mapping	Foreign Key Mapping does not handle many-to-many relationships. For this reason, association table mapping was used for querying relationships between listings and users, as one example. There are four association tables in our database design; bid, auction_order, user_sg_mapping, and fixed_order. This pattern has been used to map relationships in all but one case. Unfortunately due to the fast pace of development, an explicit class representing a mapping table was created, and the error was discovered too close to the deadline for the team to be able to refactor it out.
Embedded Value	The intent of using embedded fields here was to avoid creating an Address table that would never be queried on its own, but to still have a robust Address domain object that would aid in input validation for addresses. Initially, and for the majority of the project, address was an embedded value in AppUser, FixedPriceOrder and AuctionOrder, but the team forgot it was required to use embedded value in the specification and deleted the class and replaced the DB fields with a single string for simplicity. The error was discovered too close to the deadline to be remediated.
Concrete Table Inheritance	Since the design features inheritance, an inheritance pattern had to be chosen. Concrete Table Inheritance was selected as it simplifies queries made to the database, and minimised bottlenecking that could occur if all inheriting classes used the same table or had to fetch from the same superclass table, especially as AuctionListings will likely be queried more often due to bids being made and checked, so FixedPriceListings will not suffer. Additionally, the standard drawbacks of the pattern are not difficult for the system's

	use case. Using UUIDs solve the key problem, the database schema is stable and will not necessitate bad refactors, and even then changes are relatively easy to implement.
Data Mapper	See detailed discussion below.
Unit of Work	See detailed discussion below.
Lazy Load	See detailed discussion below.
Model-View-Controller	This pattern was implemented throughout the entire project. The main purpose is to control user input and what is seen by the user. We have used Servlets for our Controllers, Java Classes for our Models and JSPs for our Views. Through these 3 components we have been able to handle the input, manipulation and display of information.
Authentication and Authorisation	This has been implemented through the utilisation of Spring Security. Authentication in spring security is implemented through their security context holder. When a user logs in their details, and importantly role, are stored in the context. When a user sends a request it is intercepted by the Authorisation portion of Spring Security, the security filter chain. This is a chain of filters that determines what the logged in user has access to based on their roles stored in the security context. This allows Spring Security to store store and control the security for the session and can quickly and easily clear the security context when a user logs out allowing a user with different authorities to log in without issue

3.1. Data Mapper

The Data Mapper pattern will be used in order to act as a mediator between domain objects in memory and the underlying database. Every domain object modelled in the database will have an associated DataMapper. Since the Concrete Table Inheritance pattern is being used, only concrete classes will have associated DataMappers - Order and Listing do not, whereas FixedPriceOrder, AuctionOrder, FixedPriceListing and AuctionListing do. The DataMapper must provide insert, update and delete functionality for each domain object, and may also provide select functionality as needed. It is worth noting that the use of the Unit of Work pattern in this system results in a slightly modified DataMapper where insert, update and delete are not publicly accessible, in order to enforce the usage of UnitOfWork. See the corresponding section for more information.

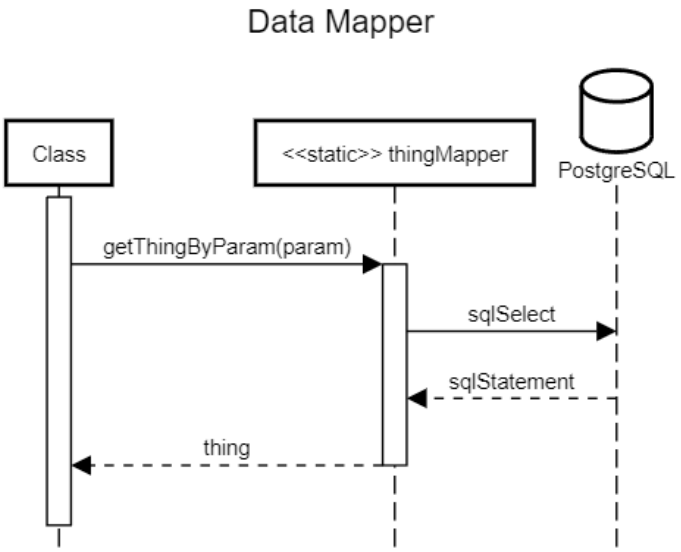


Figure: Sequence Diagram illustrating Data Mapper

3.2. Unit of Work

The Unit of Work pattern will be used to reduce the number of times queries are made to the database by collecting them as they would be made and then executing them all at once. This is done by creating a UnitOfWork instance which tracks changes that need to be made to the database. An object that is created, modified or deleted is added to a new, dirty or delete list respectively and when the Unit of Work is committed, all objects in each list have the relevant query executed.

This in particular reduces the number of queries made when multiple fields are modified for an object, as the object only needs to be updated on the database once. In the system, as orders and auctions can both be modified, with many of their fields changing in succession, it makes sense to collect multiple changes into one update as a single Unit of Work. This will also help to facilitate transactions further down the line, when concurrency becomes a necessary concern, as a UnitOfWork can act as a delimiter for the bounds of a transaction.

The implementation of the UnitOfWork pattern in this project will integrate the DataMapper pattern and subsume its insert, update and delete interfaces, so that any changes made to the database have to be made through committing a Unit of Work. This is illustrated in the Figure below. DomainObjects, when modified, register themselves with the UnitOfWork instance, and then when an external entity commits the UnitOfWork, it will fetch the appropriate mappers and execute changes to the database through them.

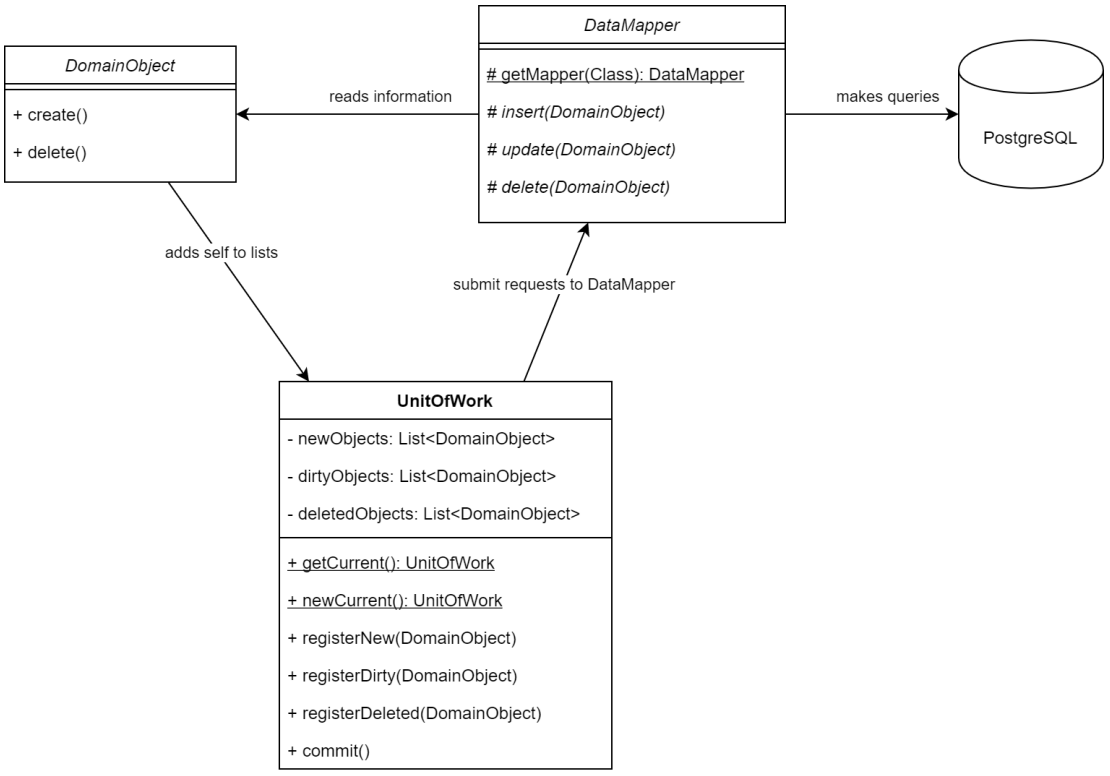
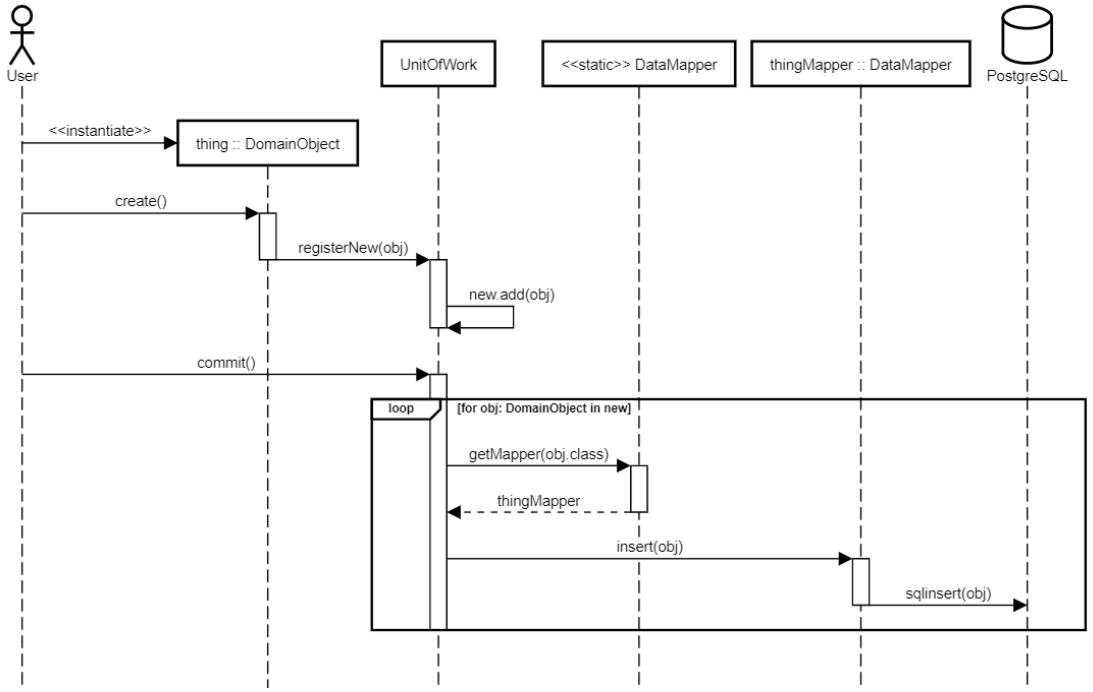


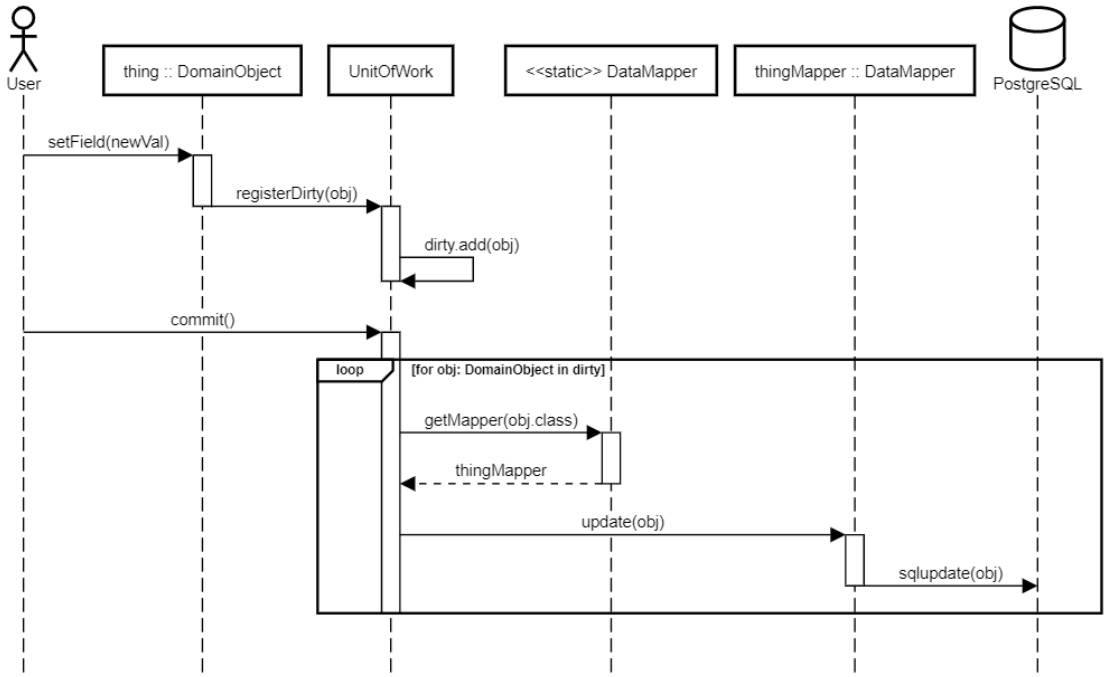
Figure: Class Diagram illustrating UnitOfWork

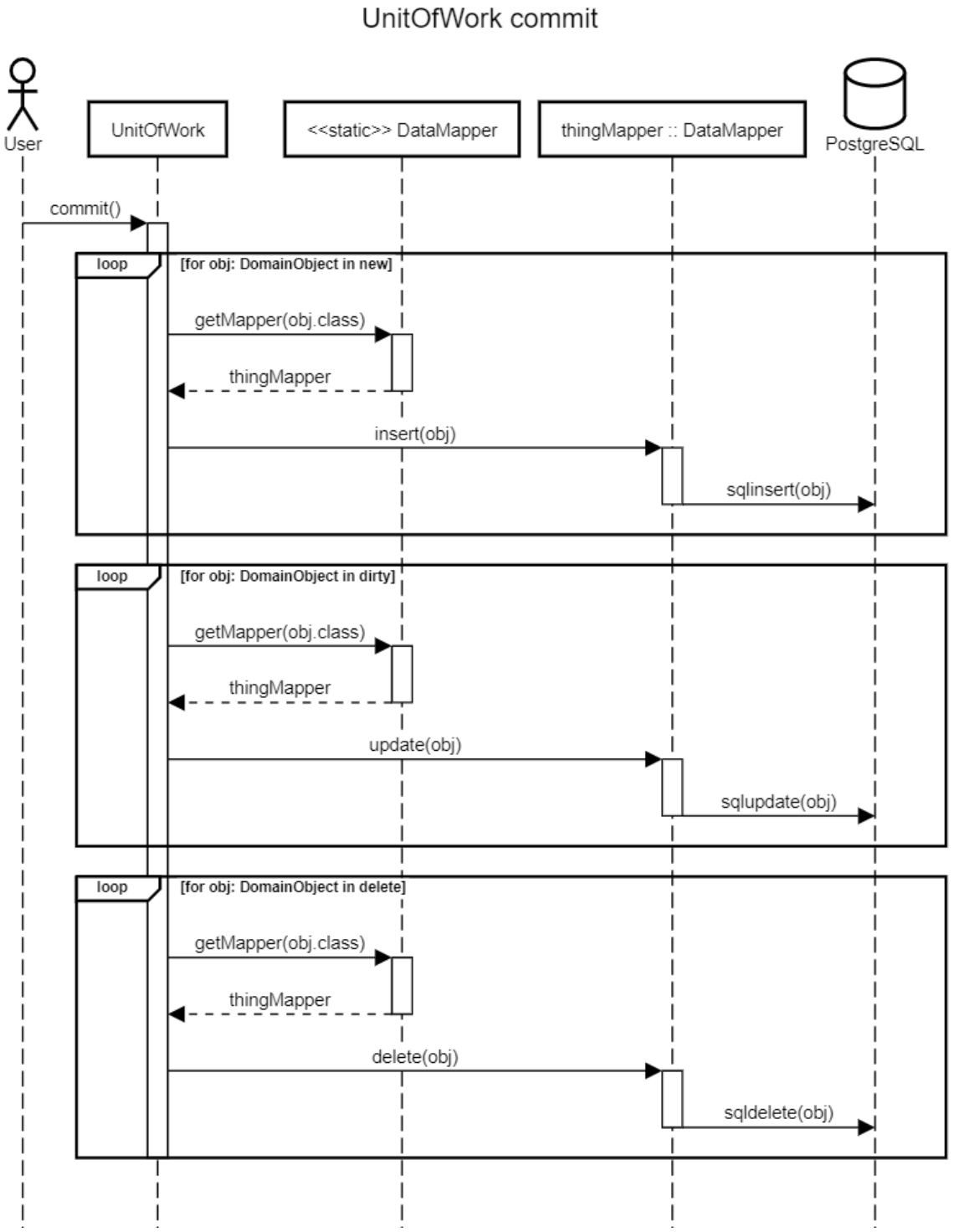
The following sequence diagrams illustrate how UnitOfWork is used to coordinate changes made to the database.

Insert Domain Object into DB



Modify Domain Object in DB




UnitOfWork commit
User
UnitOfWork
<<static>> DataMapper
thingMapper :: DataMapper
PostgreSQL
commit()
loop
[for obj: DomainObject in new]
getMapper(obj.class)
thingMapper
insert(obj)
sqlinsert(obj)
loop
[for obj: DomainObject in dirty]
getMapper(obj.class)
thingMapper
update(obj)
sqlupdate(obj)
loop
[for obj: DomainObject in delete]
getMapper(obj.class)
thingMapper
delete(obj)
sqldelete(obj)

3.3. Lazy Load

Lazy loading allows us to save space in memory by creating objects that do not have all of their variables initialised. These variables are only read from memory when they are required for the object. In larger systems it can also prevent subsystems from being queried for an objects' parameters unless those queries are required, and can delay those queries until the time of need, meaning data is more likely to be up-to-date.

In this case we have chosen to implement the "Ghost" pattern of lazy load. The ghost pattern instantiates an object and only stores 1 identifying attribute of it in memory, all other attributes are null. This ghost object saves on memory by only storing its own key. When a class requests information from the ghost that it does not have, it will request all of its information from the database and ensure that all of its attributes are loaded using the key attribute it stored. This means that we don't have to store information about objects in memory until they are accessed by another class. This pattern is particularly useful for retrieving large sets of objects where we may not need their data unless we specify that we are interested in them.

In our implementation we decided that listings was the correct place to implement this pattern. Listings have the most information out of any of the models and should be extensible to include pictures and/or videos which are very memory intensive. To do this we implemented a lazy listings method into the mapper class for the object. However we decided in our final implementation not to utilise this method for multiple reasons. For the time being there are no images or videos associated with listings, and they are displayed with the majority of their entries in the system. Through pagination we have restricted the amount of memory that is consumed by listings as well. We have ensured that the lazy loading functionality is in our source code and can be used for extensions and alternative methods of display in the future. But for this release we have left the methods unutilised as they did not save any memory and only resulted in additional redundant database queries to be performed. We believe that lazy loading is not a relevant pattern for the use cases which we have implemented. If we prove to be wrong on this matter, lazy loading can be switched back on for Listings by using the appropriate already implemented methods in Mappers implementing the interface "ListingMapper". The methods for lazy loading are suffixed with "Lazy" and already exist.

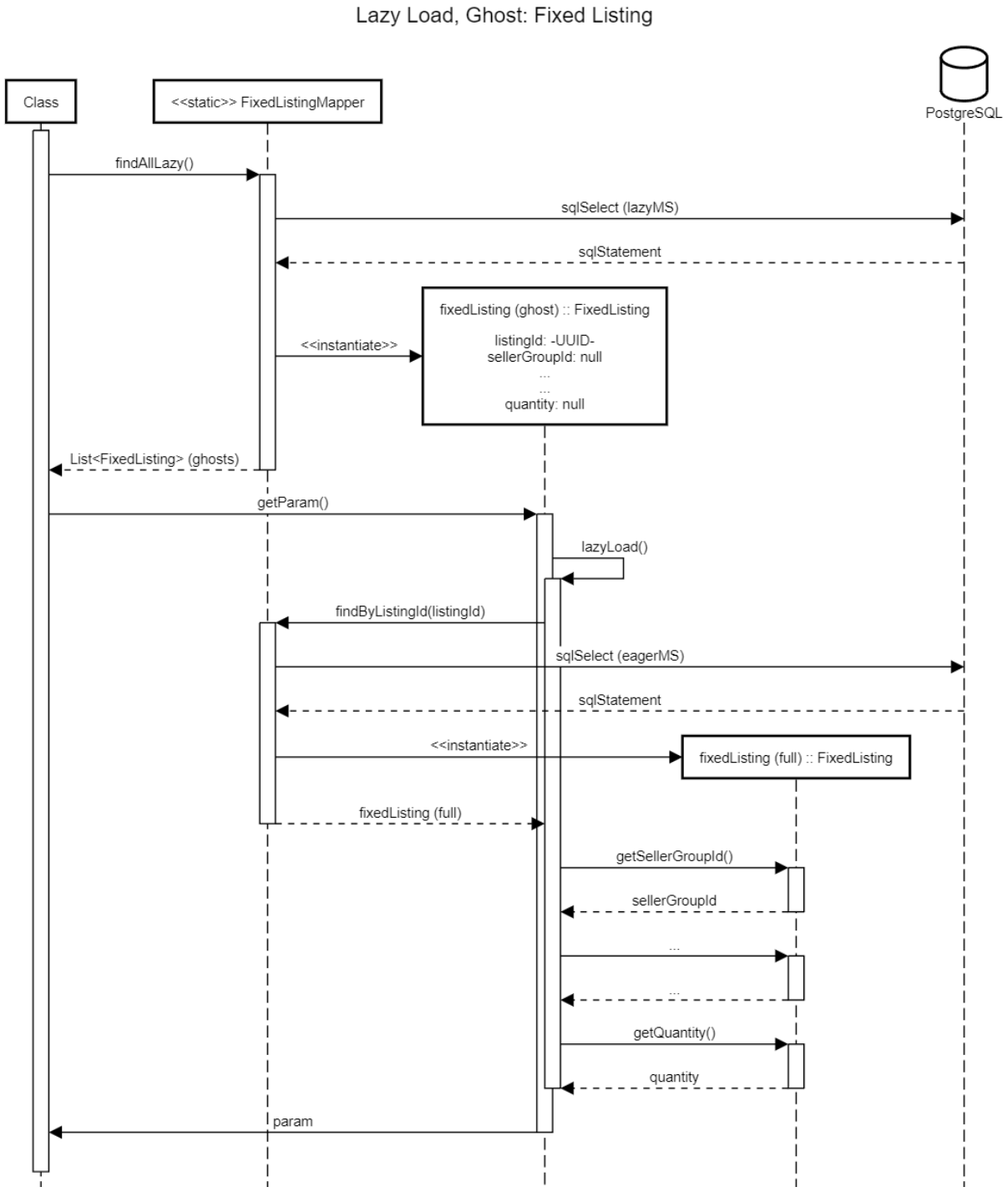


Figure: Sequence Diagram illustrating Lazy Load, Ghost for Fixed Listings

Appendices

Appendix 1: Site Map