SWEN90007 SOFTWARE DESIGN ARCHITECHTURE

Team Info

Name	***Email***
Chuang Wang	*chuangw@student.unimelb.edu.au*
Junhan Yang	*junhany@student.unimelb.edu.au*

Table of Contents

Section 1: Project Overview

In this project, we are aiming to build a web-based and enterprise-scale property resource application which connects real estate agents with highly engaged and informed consumers right across the buying and renting cycle. Properties on this platform include residential and commercial properties. The application serves as a way that real estate agents can publish housing estates on it for sale or rent. Meanwhile, it provides a centralized resource platform where consumers can seek for properties to buy or rent and book inspections for ones they like.

This system has sufficient complex business logic for two main reasons. First, there are two main roles involved in this system including real estate agency and customer. Meanwhile, each real estate agency may have multiple individual agents working in that company and each agent may have a different set of property resources he or she is particularly responsible for. Secondly, in addition to publishing resources and seeking for properties, customers and agents can interact and communicate with each other via the platform. For instance, a customer can submit an enquiry regarding a particular apartment to the agent who is in charge of that property.

Feature A: Essentially, all the functionalities implemented for Feature A are supposed to be used by real estate agents. In particular, they are the operations that agents can perform on properties including **C**reate, **R**ead, **U**pdate, **D**elete a particular property (more details in section 2 & 3). Apart from that, use account management has been implemented in Feature A as well including sign up, sign in, user profile update.

Feature B: In this deliverable, all the use cases implemented are intended to be used by property seeker(client). In particular, a client can search properties based some filters, search agents by agent name, and add/remove a particular property from his or her favourited property list (more details in section 2). Apart from that, we also various patterns for session, security, concurrency, input controller and view, and DTO and remote façade.

For the links to project repository and deployed website, please click the following hypertexts or copy and paste the URL below into your internet browser:

Item	**Link**
Project Repository	https://github.com/chuangw46/SWEN90007_Project
Deployed Application website	https://swen90007-2019-realestate.herokuapp.com/

The credentials for existing users for the system:

Username(email)	**Password**	**Role**
chuangw@student.unimelb.edu.au	123456	Agent
junhany@student.unimelb.edu.au	123456	Client(Property seeker)

Notes:

1. Other than using those existing accounts, you are welcome to create your own account by providing required information in the sign-up page. The detailed testing scenarios are provided in Section 3.
2. For the sake of readability, pretty much all the sections or diagram components in **red colour** are for the feature B.

Section 2 Features and Use cases

1. Feature A:

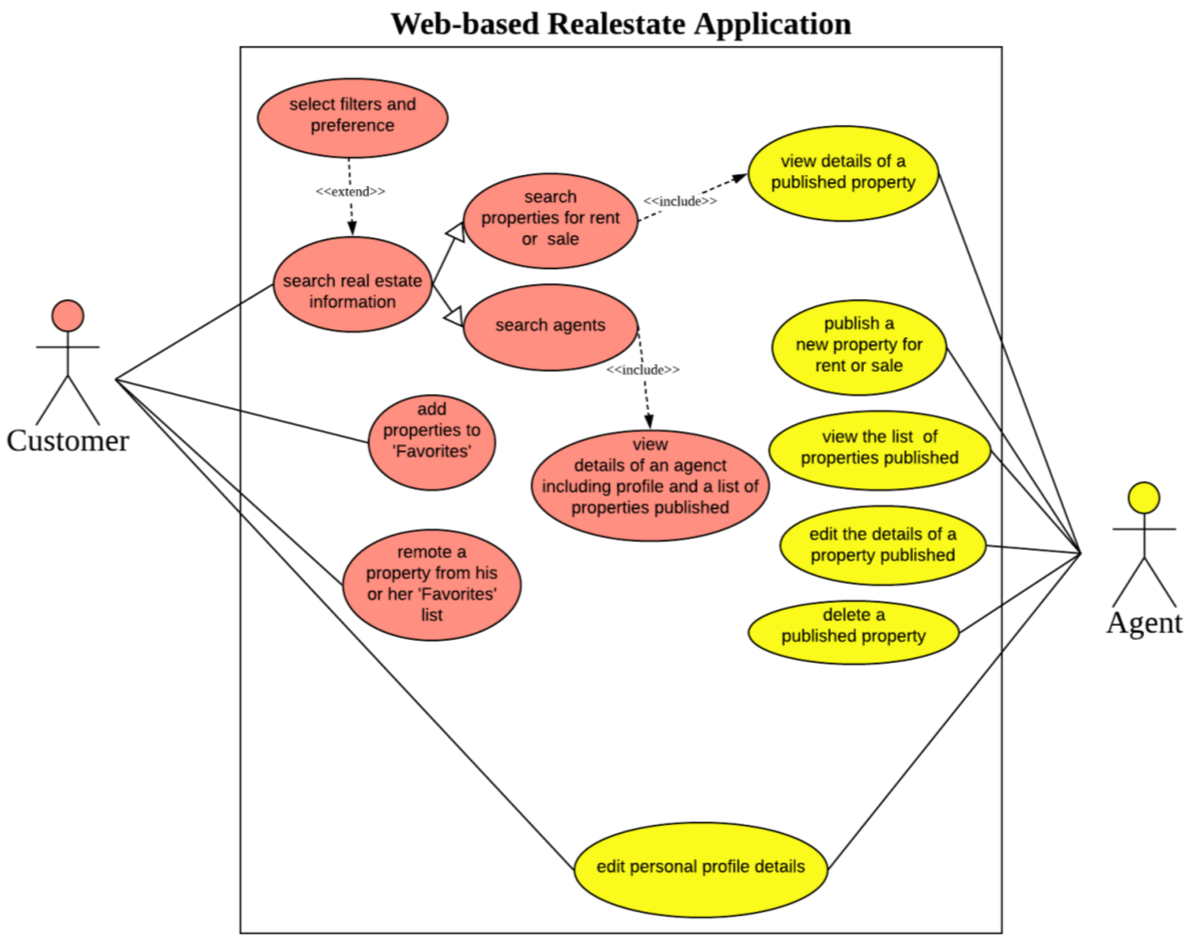
By using the platform, registered real estate agents can publish housing estates on it for sale or rent. Use cases are described as following

1) Real estate agents can register accounts on the platform and create their own personal profiles including name, position, company, bio, skills & expertise, and their contact information. Once they have created accounts, they can login and perform a series of actions on the platform.
2) Registered agents can update their personal profile including contact, and the company he or she is working for.
3) A real estate agent would be able to publish properties for sale or rent. The published information about a particular housing includes the properties address, price, interior photos, prices, available date, facilities description.
4) A real estate agent would be able to check a list of properties he or she published on this platform and see how many people have registered for inspection for each property.
5) If there is a change on a property's price, inspection date and available date, the agent who is responsible for it will have the permission to update that information and make it published and inform observers.
6) If a property is not available any more for some reasons including leased or the factor that the landlord does not want to sell or rent anymore, the agent would be able to delete all the information regarding that particular property.

2. Feature B

Users registered as 'customers' are able to look for properties that are currently available for renting or selling. To be more specific, use cases for a customer are described as following.

1) A customer can sign up for a new account and log in using that account.
2) A customer can edit their own customer profile after logging into their accounts in the profile page.
3) A customer can search properties based on filters under the search bar including price range, room size, location and real estate agencies.
4) A customer can search agents based on names under the agent search bar.
5) A customer can add a preferable property from searching results into their 'Favorites' list.
6) A customer can remove a property from their 'Favorites' property list.
7) A customer can look into detailed information about properties or agents.


Web-based Realestate Application
select filters and preference
<<extend>>
search real estate information
search properties for rent or sale
<<include>>
view details of a published property
search agents
<<include>>
view details of an agenct including profile and a list of properties published
add properties to 'Favorites'
remote a property from his or her 'Favorites' list
publish a new property for rent or sale
view the list of properties published
edit the details of a property published
delete a published property
edit personal profile details
Customer
Agent

Red colour stands for Feature B whereas yellow stands for Feature A.

Section 3 System Test Scenarios

1. Test scenarios for Feature A (Agents)

Step	Action	Description
1	Sign up	1) Check system behavior when providing all required fields in sign up page 2) Check system behavior when not providing all required fields in sign up page
2	Login	1) Check system behavior when valid email id and password is entered. 2) Check system behavior when *invalid* email id and *valid* password is entered. 3) Check system behavior when *valid* email id and *valid* /invalid password is entered. 4) Check system behavior when email id and password are left blank and Sign in entered.
3	View profile	1) Check the profile page by clicking your name. 2) Check whether all the information showed there are what you entered before.
4	Update profile	1) Check system behavior when editing *all/part of your* personal information 2) Check if all information is updated after clicking save button
5	Publish a new Property	1) Check system behaviour by clicking publish a new property 2) Check system behaviour by providing required fields about the property in the form page. 3) Check system behaviour by clicking publish 4) Check if the new property is in your published list
6	View all properties published by the agent	Check if the list contains all properties you published before by clicking view all properties button
7	View detailed information about a specific property	Check if the detailed page contains all the property information you provided before when clicking view button aside the property
8	Edit a specific property	Check if the edit page shows up and allows you to overwrite previous details when clicking edit button aside the property
9	Delete a specific property	Check if the property is deleted from you published list after clicking delete button

2. Test scenarios for Feature B (Property Seekers)

Step	Action	Description
1	Sign up	1) Check system behavior when providing all required fields in sign up page 2) Check system behavior when not providing all required fields in sign up page
2	Login	1) Check system behavior when valid email id and password is entered. 2) Check system behavior when *invalid* email id and *valid* password is entered. 3) Check system behavior when *valid* email id and *valid* /invalid password is entered. 4) Check system behavior when email id and password are left blank and Sign in entered.
3	View profile	1) Check the profile page by clicking your name. 2) Check whether all the information showed there are what you entered before.
4	Update profile	1) Check system behavior when editing *all/part of your* personal information 2) Check if all information is updated after clicking save button
5	Search a property	1) Select all the filters for both buy & rent navigation components and click search, check if the page shows all the properties that meet the criteria you selected in the filter bar 2) Select part of filters or even no filter and click search, check if all available properties show up
6	Check detailed information about a specific property	1) From the search result list, check if the property detailed information page shows up after clicking a particular property box.
7	Search an agent	1) Under the `find agents` navigation, type a character `u` or any character in `Chuang Wang`, check if it shows all the agents whose name contains those character. 2) Under the `find agents` navigation, type an agent's first name, or last name or full name (`Chuang Wang`) and click search, check if it shows all agents whose name matches what you entered. 3) If an agent shows up, try to click the agent name and check if the agent profile page shows up and the page should contains all the detailed information and a list properties published by this agent.
8	Like a property (after login only)	1) From the property detailed page generated from step 6, try to like the property by clicking the blue star 2) Check if the blue star becomes filled with blue colour
9	Unlike a property (after login only)	1) Try to click the filled blue star again to unlike the property from step 8-2) 2) Check if the blue star becomes unfilled, which means the properties is removed from your favourite property list

10	View the list of my favourite (after login only)	1) If you perform the action in step 8 a couple of times, in other words, if you have added at least one property to your favourite list, try to click `my favourite properties`, check if all your favourite properties are saved here.

Section4: Diagrams and Explanation

1. High level Layered Architecture Diagram

1) Presentation Layer

   This layer contains the user-oriented functionality responsible for managing user interaction with the backend. It also consists of components that provide a communication channel into the core domain(business) logic encapsulated in the domain logic layer.

2) Domain Logic Layer

   This layer decouples the data source layer and the presentation layer. It implements the core functionality of the main features for agents and encapsulates the main business logic.

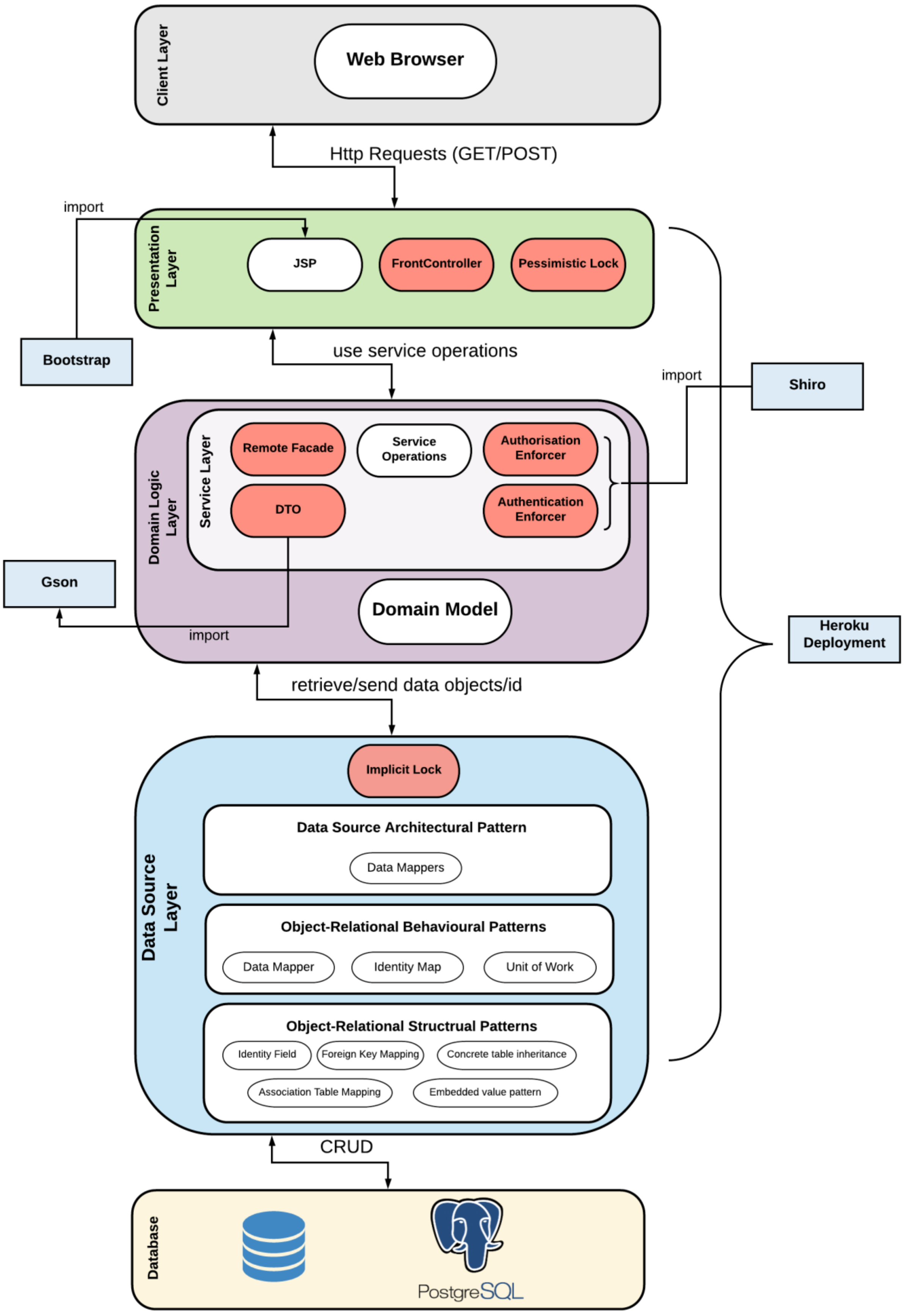
   - Service Layer
     - This layer defines a set of available operations for user account management and property management
     - It also coordinates the requests sent from presentation layer and the response from the data source layer. From the lecture and lecture notes, in order to make domain model classes clearer and for the sake of simplicity of the system, service layer has been designed to communicate with data source layer directly by calling unit of work or specific mapper to retrieve data, rather than delegating the tasks to domain model classes, which increases the complexity of domain model classes and makes our architecture messed.
   - Domain Model
     - Domain model layer basically contains all classes that encapsulates data only corresponding each table in the database.

3) Data Source Layer

   Data layer. This layer provides access to data hosted in PostgreSQL database including create, read, update and delete operations. To facilitate the data access and queries sent from client, unit of work pattern and mapper pattern are used in this layer.

   Diagram is available at next page.

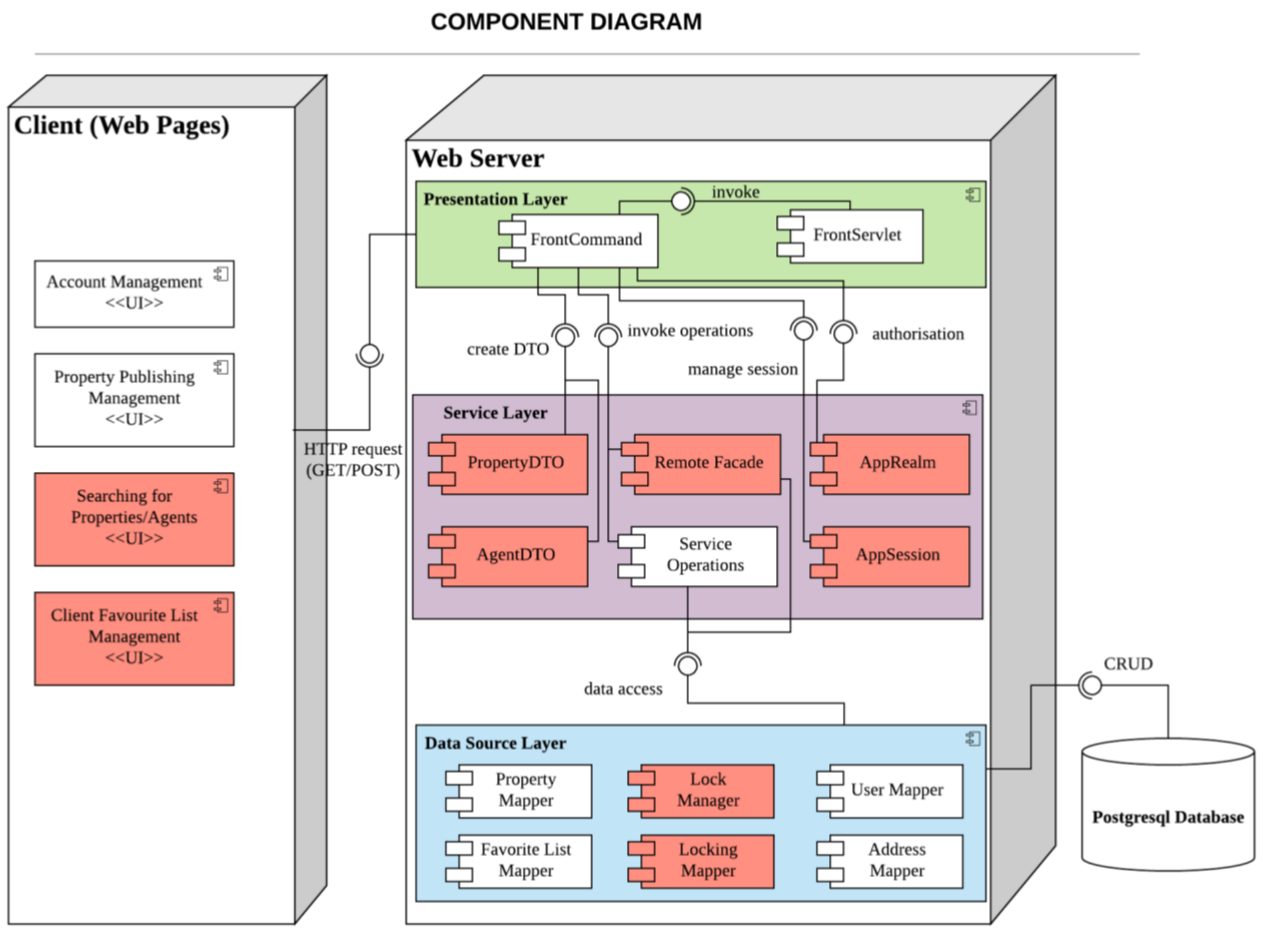
LAYERED ARCHITECHTURE DIAGRAM



2. Component Diagram

In this modified component diagram, components in the web server are classified according to which layers they are at. In particular, the client side communicates with the server side through the FrontServlet component in terms of GET/POST requests, and the data mappers such as Property Mapper performs CRUD actions to manage the data stored in PostgreSQL.

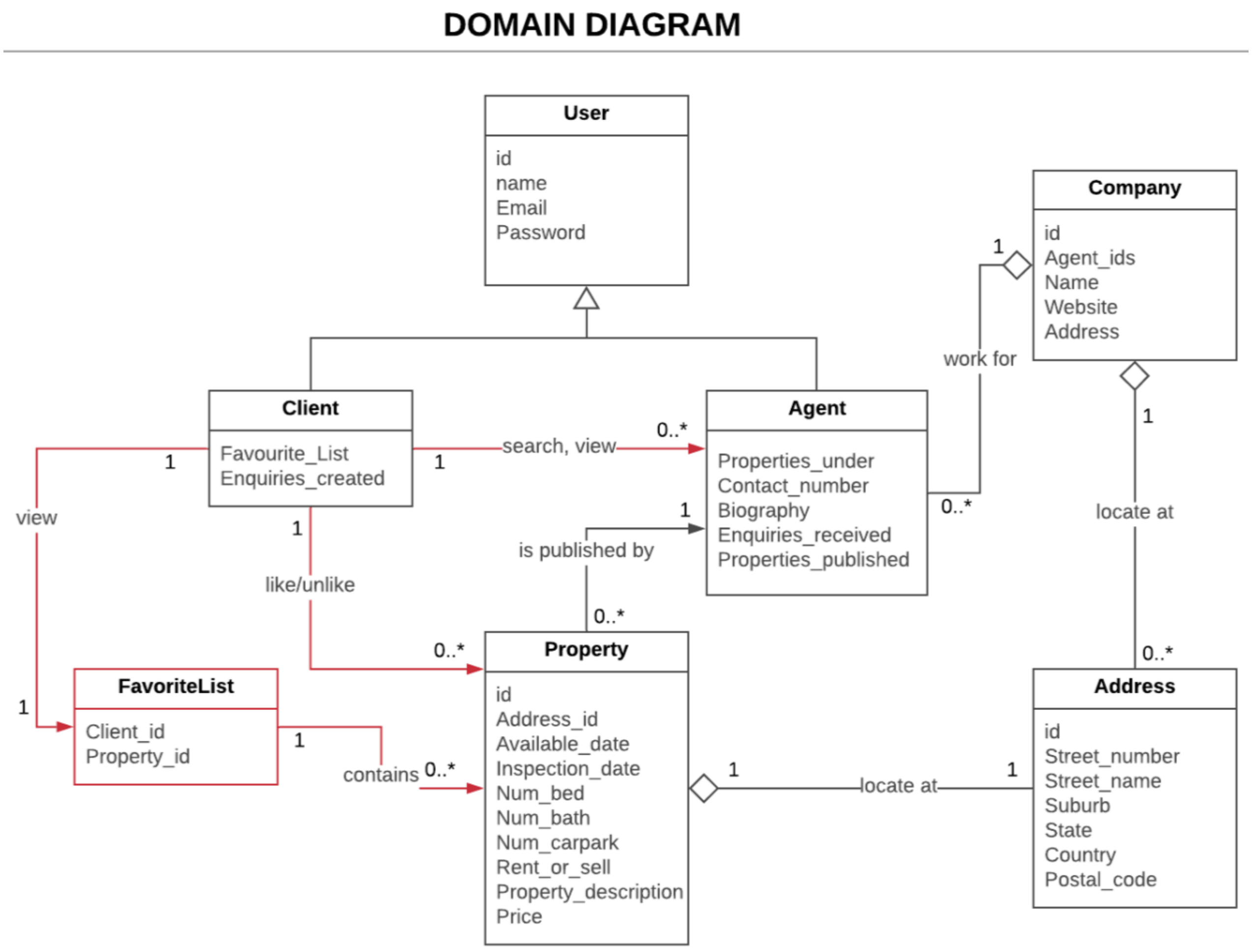
Noted that components with red background colour are implemented for Feature B in this deliverable. Since the system is not distributed, DTO and remote façade patterns in the service layer are only partially used.



3. Domain Diagram

This domain diagram illustrates the system of abstraction and the design of our object model including their data attributes, relationships and interactions. Red lines and borders refer to the latest changes in our domain diagram for Feature B. In particular,

1) A client has one favourite list of properties which can be viewed.
2) A client can like or unlike multiple properties.
3) A client can search multiple agents by name and view their profiles.
4) A favourite list contains multiple properties.

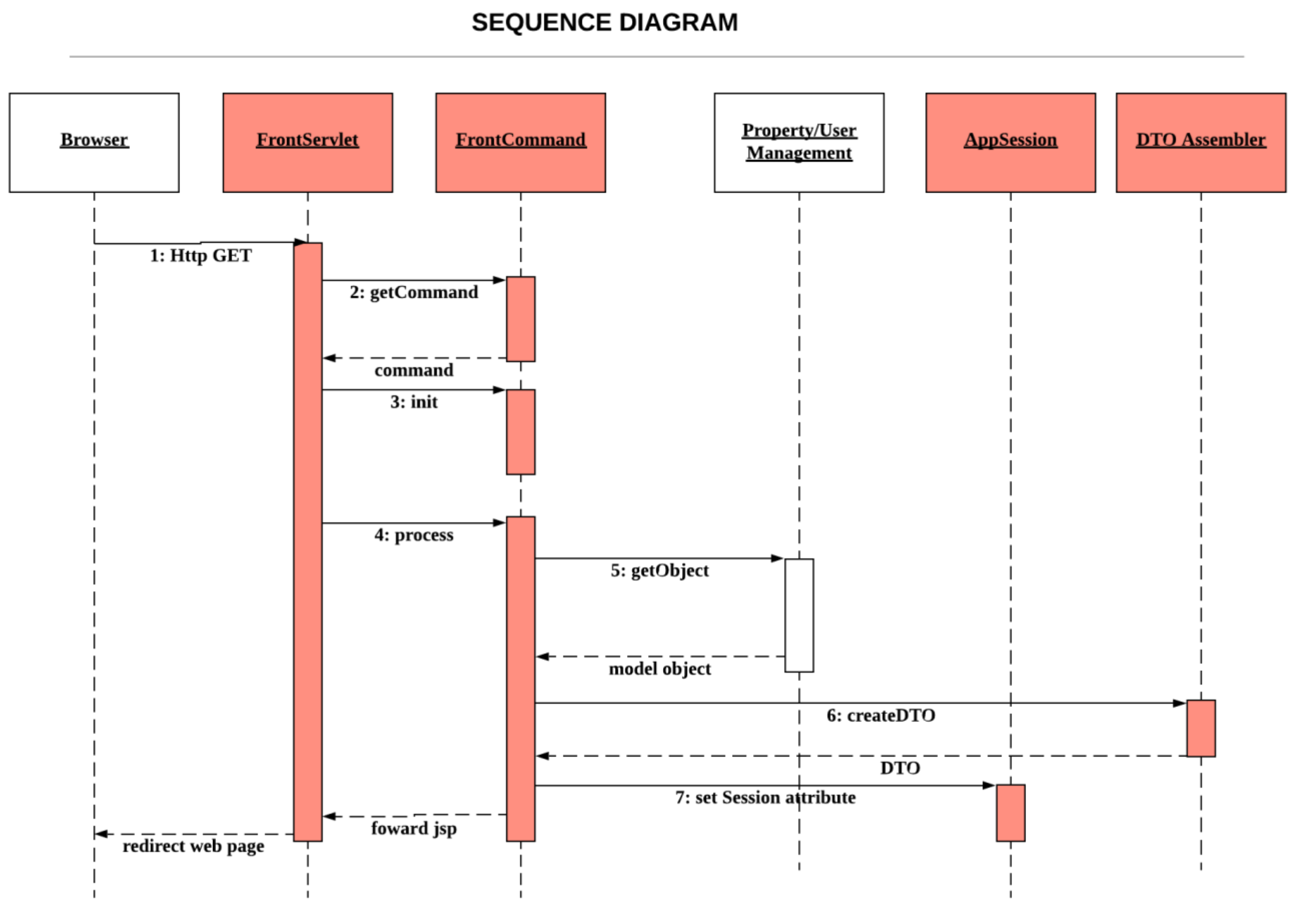


4. Sequence Diagram

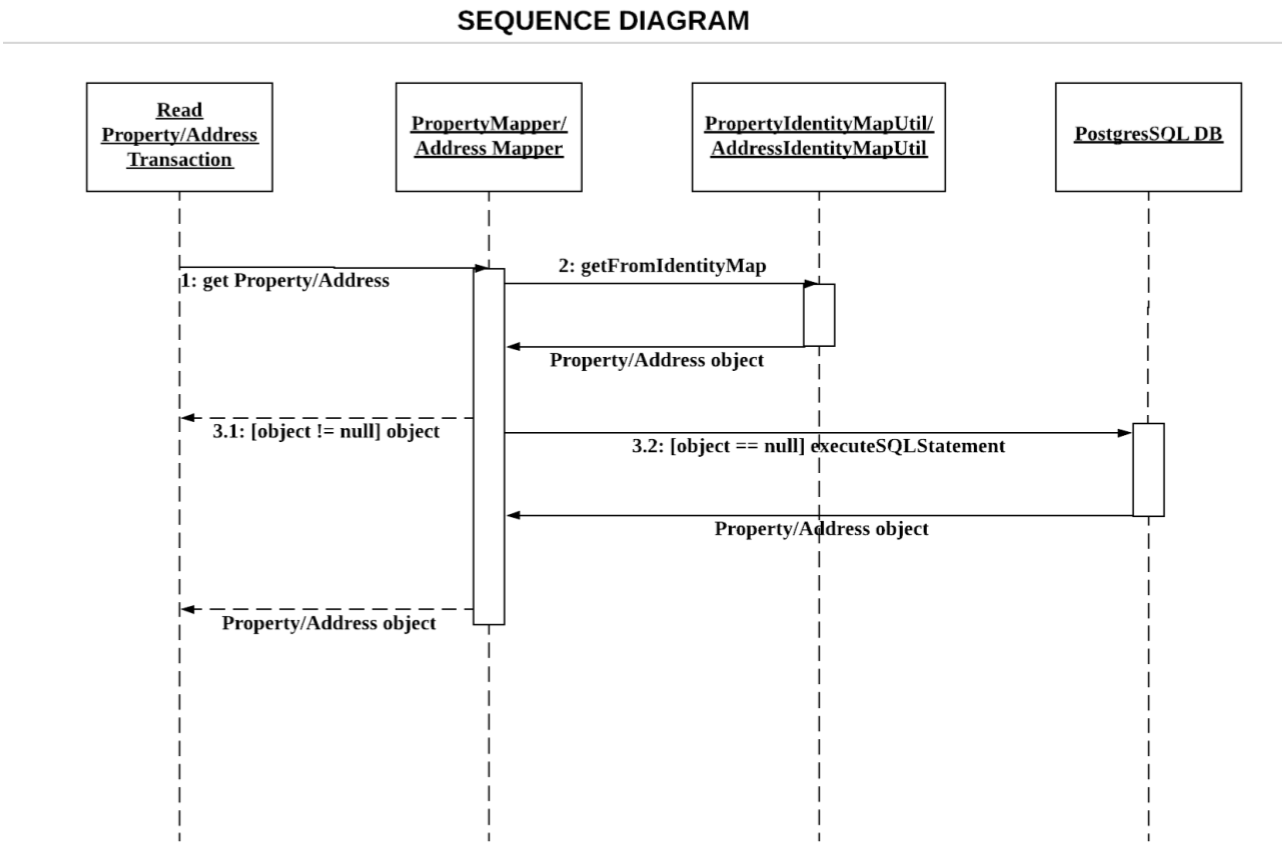
A. GET Request

1) Send from client to server

This sequence diagram illustrates the sequence of actions taken for GET request sent from the client side until the service layer. Red parts in the sequence diagram refer to the latest implementation for Feature B. Note that there are two types of DTO, namely propertyDTO and agentDTO, so not all models are encapsulated into DTO for communications between server side and client side.



2) **Send from server to database**
This sequence diagram illustrates the sequence of actions taken for GET request sent from the service layer to database for the purpose of retrieving data objects. There are no changes in reading property or address from the PostgresSQL database for the submission 3.

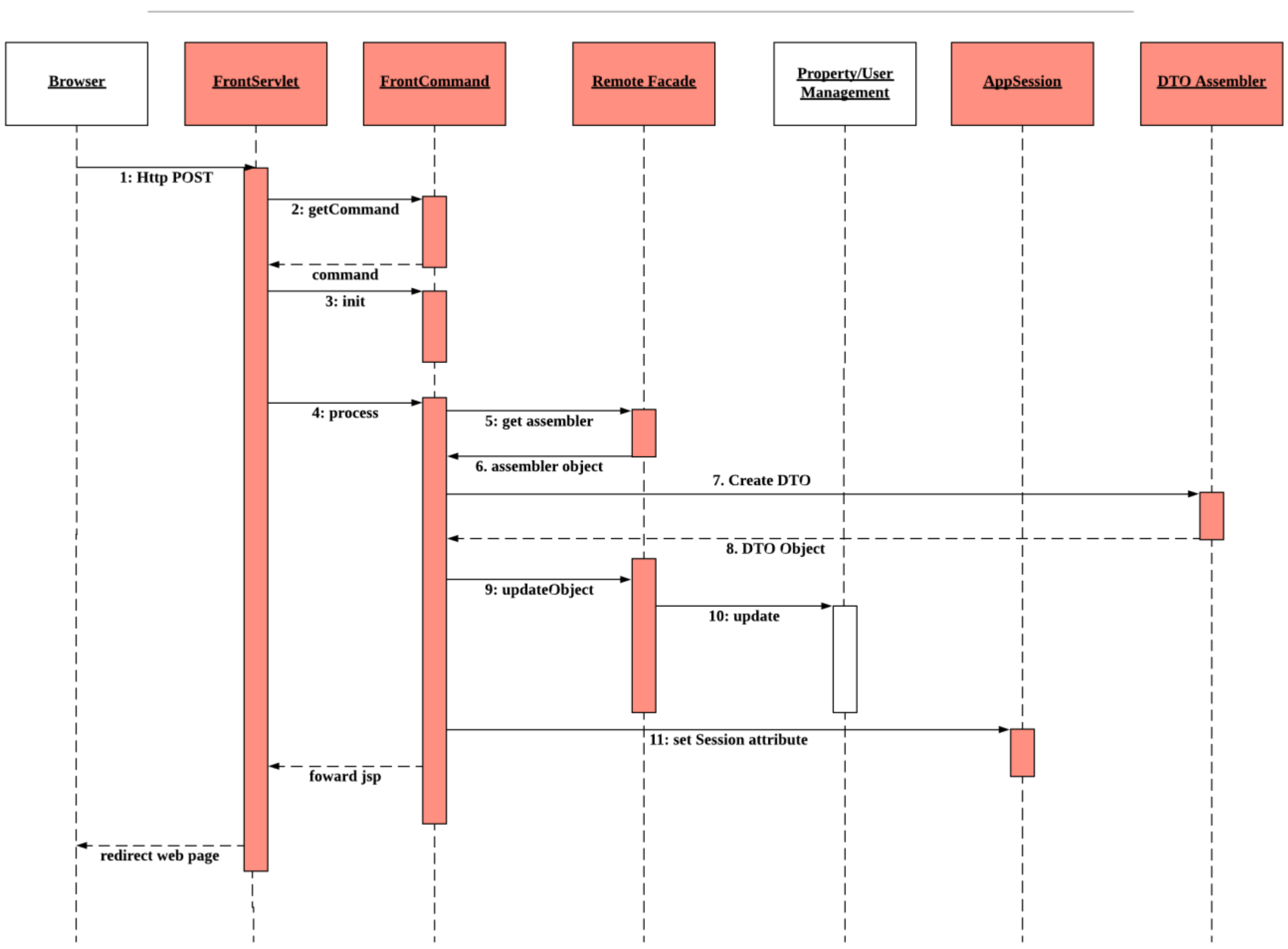

SEQUENCE DIAGRAM
Read Property/Address Transaction
PropertyMapper/ Address Mapper
PropertyIdentityMapUtil/ AddressIdentityMapUtil
PostgresSQL DB
1: get Property/Address
2: getFromIdentityMap
Property/Address object
3.1: [object != null] object
3.2: [object == null] executeSQLStatement
Property/Address object
Property/Address object

B. POST Request

1) Send from client to server

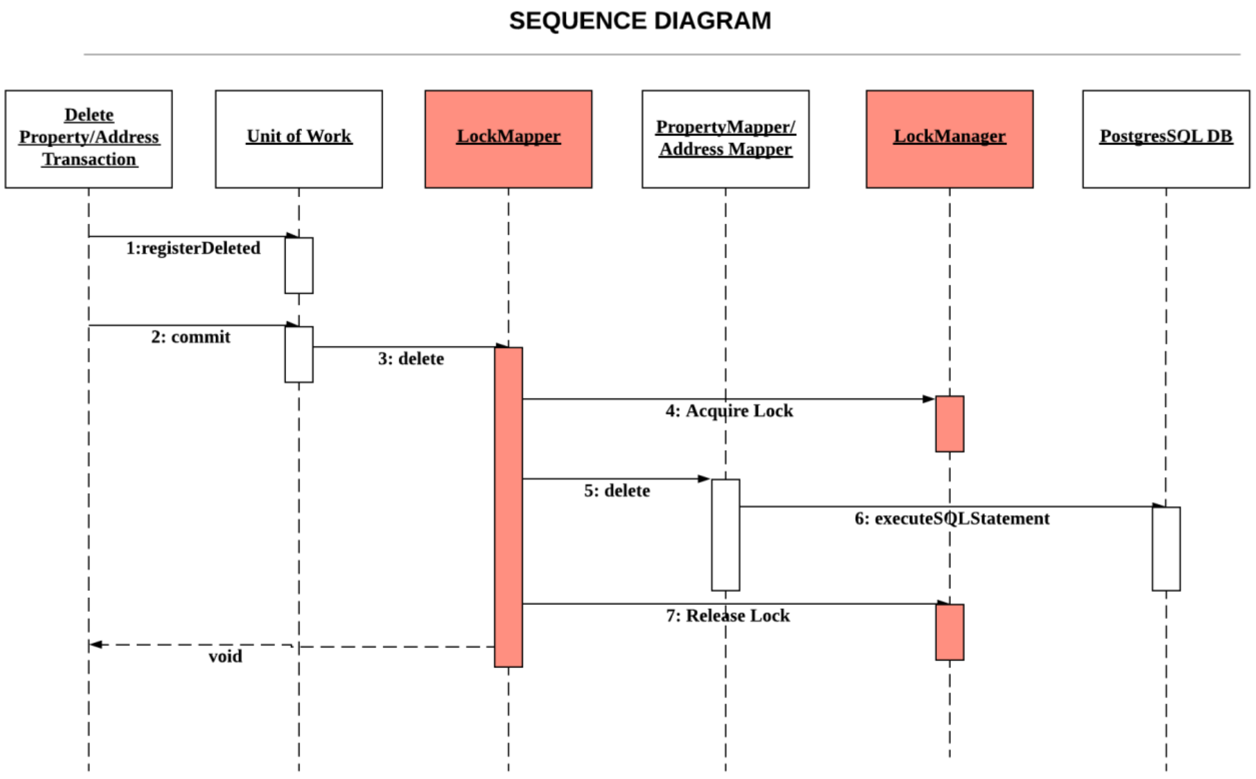
This sequence diagram illustrates the sequence of actions taken for POST request sent from the client side until the service layer (Property/User management service). Red parts in the sequence diagram refer to the latest implementation for Feature B. Noted that the return type for updateObject() method in remote facade class is void. If a SQLException is caught in process() method in the FrontCommand child class, the update action is considered as failure.

SEQUENCE DIAGRAM



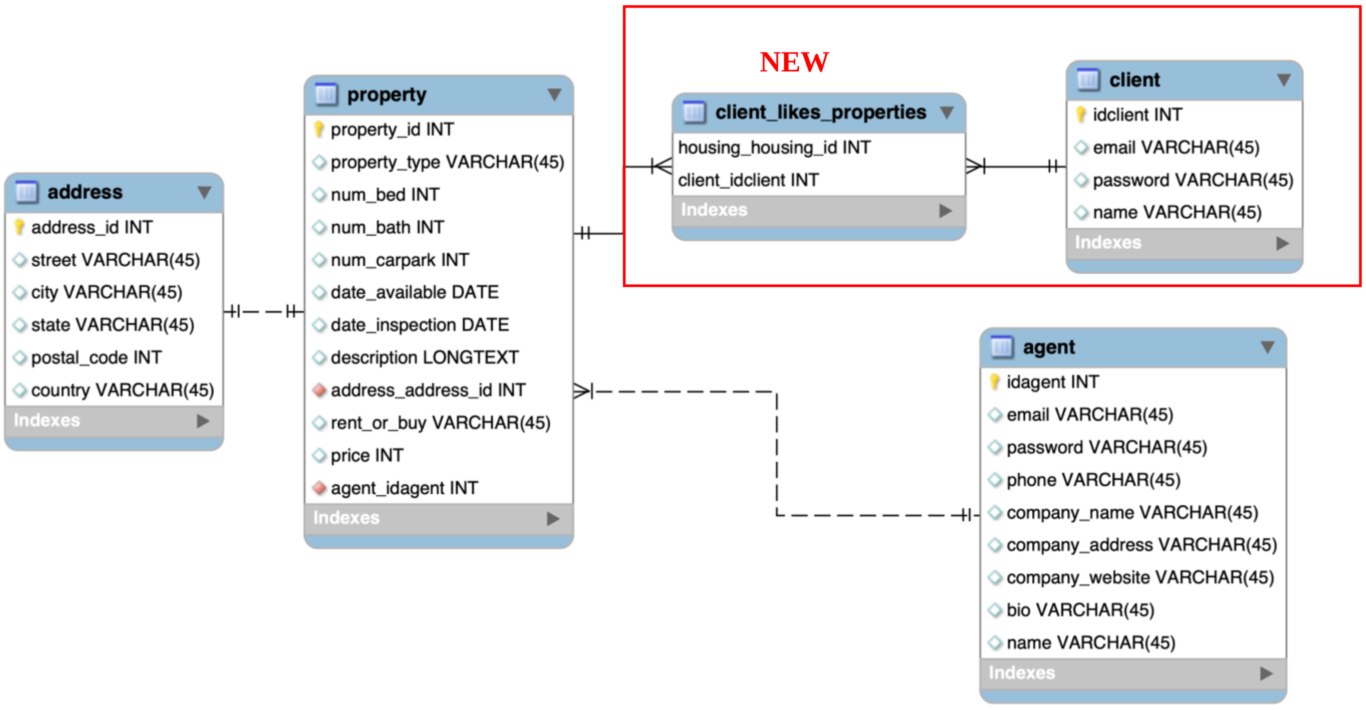
2) Send from server to database

This sequence diagram illustrates the sequence of actions taken for POST request sent from the service layer for retrieving data objects. Red parts in the sequence diagram refer to the latest implementation for Feature B. In particular, delete actions adopt implicit lock pattern by acquiring and releasing the lock before and after executing the delete action in the data mapper layer.



5. Database Scheme EER Diagram

This diagram shows how we built the database tables in PostgreSQL. Each property has one address only. The reason why address has a separate table instead of storing all the address in property table is to avoid storing too many fields in property table. Although client and agent are the intended end users of the system, we design a separate table for each of them because they perform different actions while using our system. For instance, an agent is the main person who manages a list of properties including publishing a new property, updating or deleting an existing property he or she published before. Therefore, `agent` and `property` have one-to-many relationship. On the other hand, a client (property seeker) will mainly search properties based on some filters and save a particular property into his or her favourite property list. In that sense, `property` and `client` have many-to-many relationship. As such, an association table called `client_likes_properties` is created to solve many-to-many relationship.



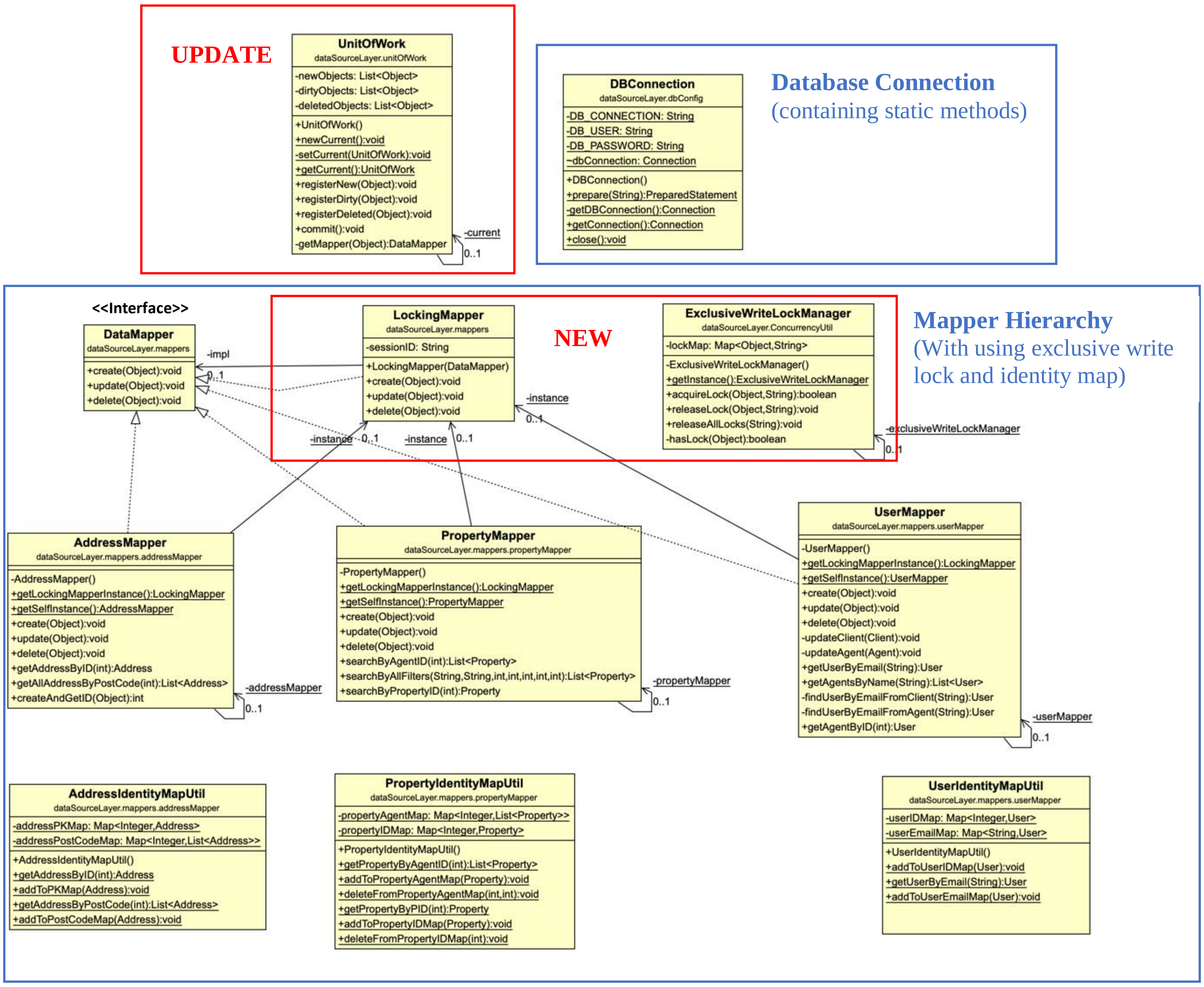
6. Class Diagram

1) Data Source Layer

This diagram shows all the necessary classes located at data source layer. Data mapper pattern is used to serve as a postman which transfer data around between the underlying database and domain layer.

It connects to database by calling static methods in `DBConnection` class. On the other hand, domain layer talks to mappers by calling unit of work methods for post operations including create, update and delete operations. However, domain layer talks to mappers by calling the get methods from concrete mapper classes for the purpose of retrieving data from database. The reason why retrieves operations and post operations are design in different way is the fact that domain layer might retrieve data based on different parameters i.e. email, name, price, ID etc., whereas post operations can be processed based on an object. And `Object` class is the super superclass for all classes in java so that create, update and delete methods which take `Object` as parameter can be extracted into an interface called `DataMapper`, which will facilitate the implementation for unit of work pattern.

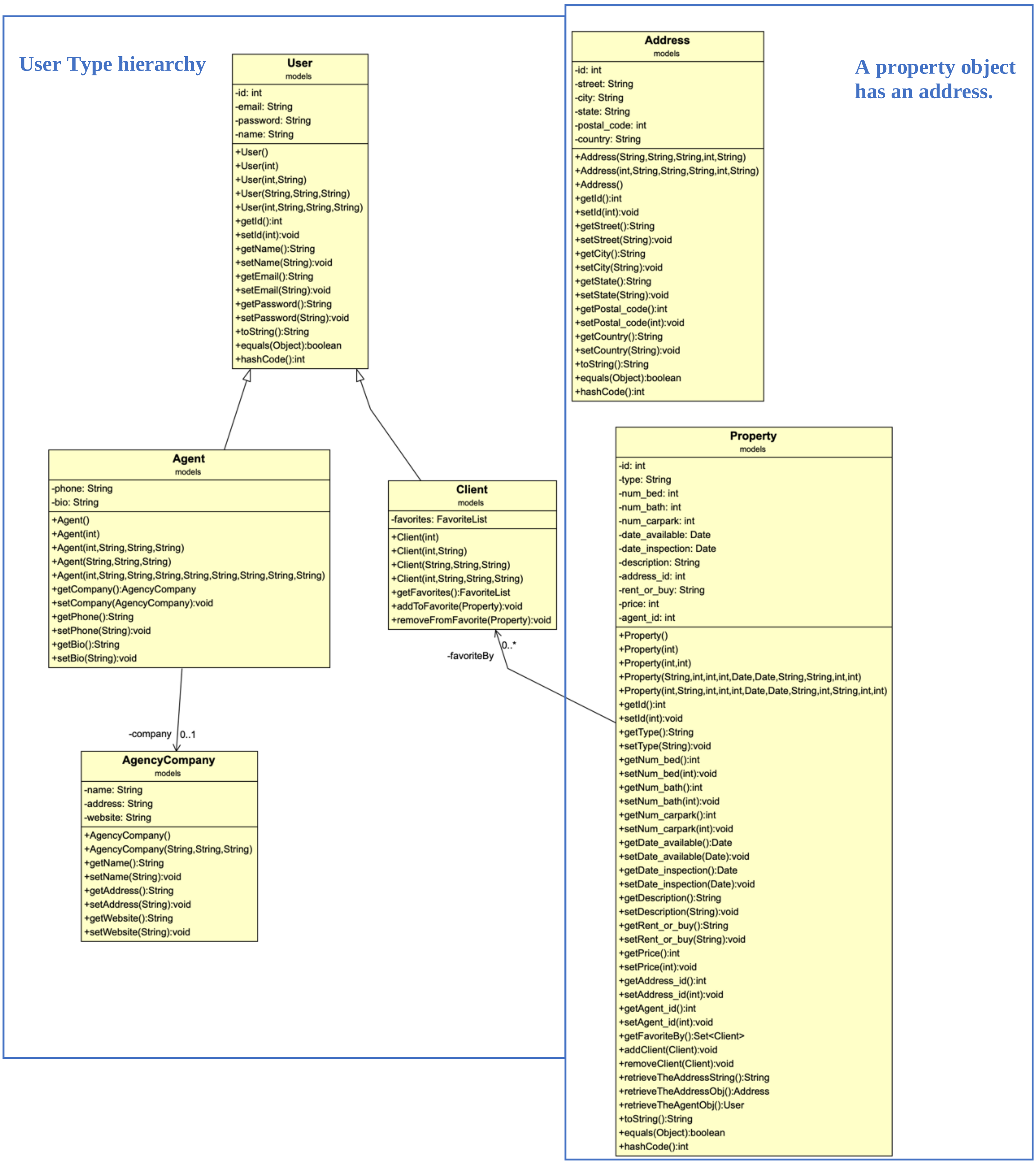
Because methods are designed static in some classes, an association line between two classes you might expected to see does not exist, such as `DBConnection` class and identity map classes.



UPDATE
UnitOfWork
dataSourceLayer.unitOfWork
-newObjects: List<Object>
-dirtyObjects: List<Object>
-deletedObjects: List<Object>
+UnitOfWork()
+newCurrent():void
-setCurrent(UnitOfWork):void
+getCurrent():UnitOfWork
+registerNew(Object):void
+registerDirty(Object):void
+registerDeleted(Object):void
+commit():void
-getMapper(Object):DataMapper
-current
0..1
DBConnection
dataSourceLayer.dbConfig
-DB_CONNECTION: String
-DB_USER: String
-DB_PASSWORD: String
~dbConnection: Connection
+DBConnection()
+prepare(String):PreparedStatement
-getDBConnection():Connection
+getConnection():Connection
+close():void
Database Connection
(containing static methods)
<<Interface>>
DataMapper
dataSourceLayer.mappers
+create(Object):void
+update(Object):void
+delete(Object):void
-impl
0..1
LockingMapper
dataSourceLayer.mappers
-sessionID: String
+LockingMapper(DataMapper)
+create(Object):void
+update(Object):void
+delete(Object):void
-instance
0..1
-instance
0..1
NEW
ExclusiveWriteLockManager
dataSourceLayer.ConcurrencyUtil
-lockMap: Map<Object,String>
-ExclusiveWriteLockManager()
+getInstance():ExclusiveWriteLockManager
+acquireLock(Object,String):boolean
+releaseLock(Object,String):void
+releaseAllLocks(String):void
-hasLock(Object):boolean
-instance
0..1
-exclusiveWriteLockManager
0..1
Mapper Hierarchy
(With using exclusive write lock and identity map)
AddressMapper
dataSourceLayer.mappers.addressMapper
-AddressMapper()
+getLockingMapperInstance():LockingMapper
+getSelfInstance():AddressMapper
+create(Object):void
+update(Object):void
+delete(Object):void
+getAddressByID(int):Address
+getAllAddressByPostCode(int):List<Address>
+createAndGetID(Object):int
-addressMapper
0..1
PropertyMapper
dataSourceLayer.mappers.propertyMapper
-PropertyMapper()
+getLockingMapperInstance():LockingMapper
+getSelfInstance():PropertyMapper
+create(Object):void
+update(Object):void
+delete(Object):void
+searchByAgentID(int):List<Property>
+searchByAllFilters(String,String,int,int,int,int,int):List<Property>
+searchByPropertyID(int):Property
-propertyMapper
0..1
UserMapper
dataSourceLayer.mappers.userMapper
-UserMapper()
+getLockingMapperInstance():LockingMapper
+getSelfInstance():UserMapper
+create(Object):void
+update(Object):void
+delete(Object):void
-updateClient(Client):void
-updateAgent(Agent):void
+getUserByEmail(String):User
+getAgentsByName(String):List<User>
-findUserByEmailFromClient(String):User
-findUserByEmailFromAgent(String):User
+getAgentByID(int):User
-userMapper
0..1
AddressIdentityMapUtil
dataSourceLayer.mappers.addressMapper
-addressPKMap: Map<Integer,Address>
-addressPostCodeMap: Map<Integer,List<Address>>
+AddressIdentityMapUtil()
+getAddressByID(int):Address
+addToPKMap(Address):void
+getAddressByPostCode(int):List<Address>
+addToPostCodeMap(Address):void
PropertyIdentityMapUtil
dataSourceLayer.mappers.propertyMapper
-propertyAgentMap: Map<Integer,List<Property>>
-propertyIDMap: Map<Integer,Property>
+PropertyIdentityMapUtil()
+getPropertyByAgentID(int):List<Property>
+addToPropertyAgentMap(Property):void
+deleteFromPropertyAgentMap(int,int):void
+getPropertyByPID(int):Property
+addToPropertyIDMap(Property):void
+deleteFromPropertyIDMap(int):void
UserIdentityMapUtil
dataSourceLayer.mappers.userMapper
-userIDMap: Map<Integer,User>
-userEmailMap: Map<String,User>
+UserIdentityMapUtil()
+addToUserIDMap(User):void
+getUserByEmail(String):User
+addToUserEmailMap(User):void

2) Domain Model

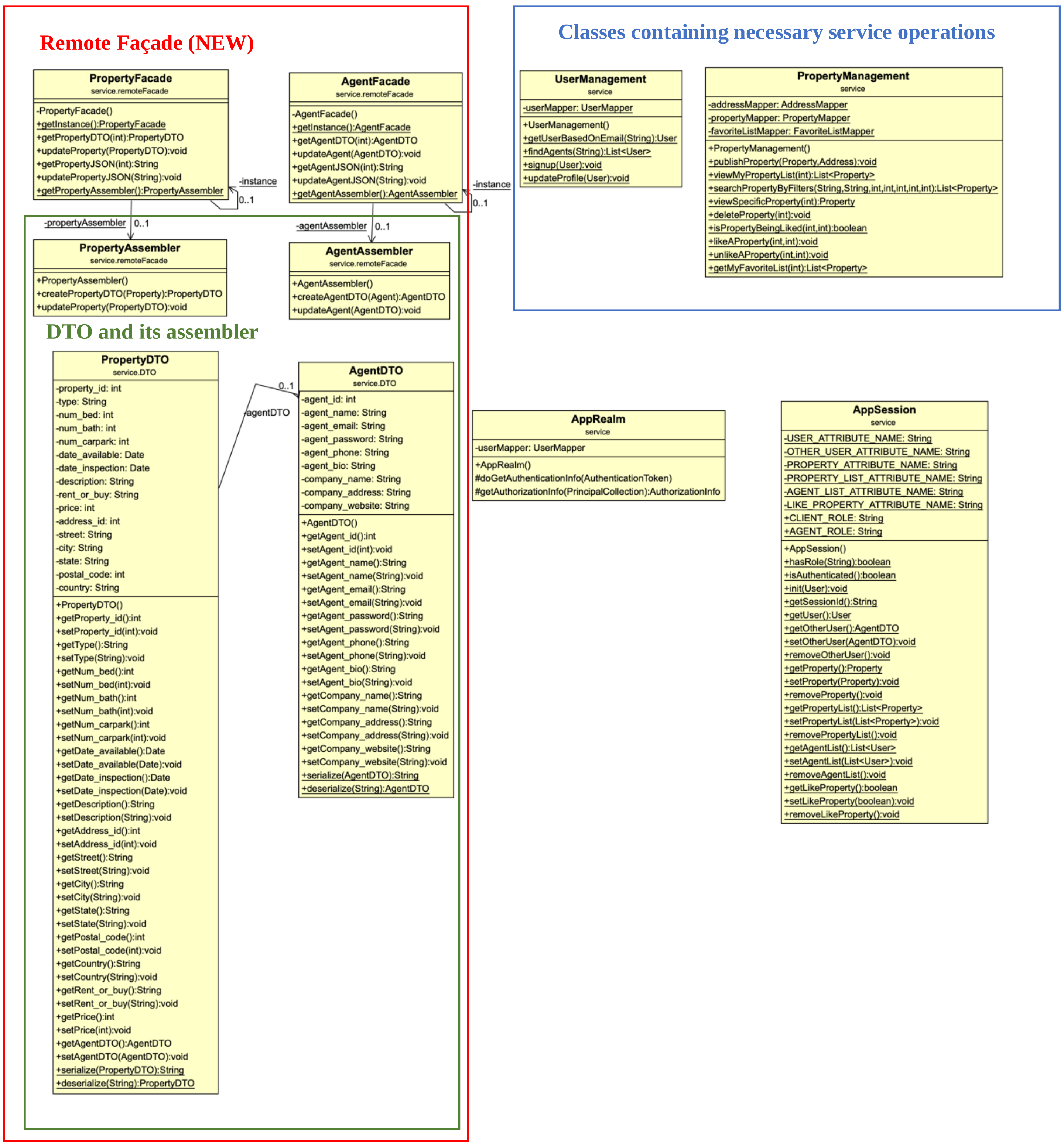
The design for each model class is based on database schema (see section 4.5). The reason why a super class called `User` is created is to capture common data and methods for agent and client class, which will help mapper implementation a lot.



3) Service Layer

This partial class diagram only shows the classes in the service layer. For the DTO design, agent model and company model are encapsulated by its assembler to become AgentDTO; property model, address model and AgentDTO are encapsulated by its assembler to become PropertyDTO.

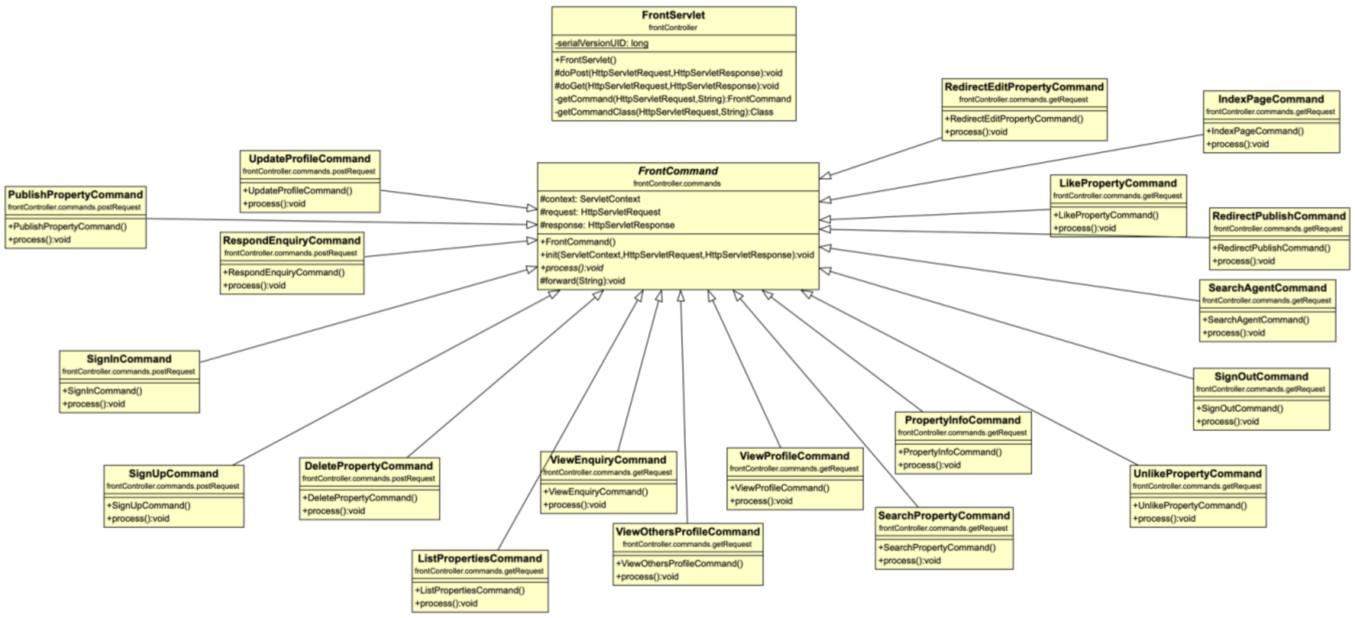
Classes are responsible for communicating with the data source layer including “PropertyAssembler”, “AgentAssembler”, “UserManagement” and “PropertyManagement”



Remote Façade (NEW)
PropertyFacade
service.remoteFacade
-PropertyFacade()
+getInstance():PropertyFacade
+getPropertyDTO(int):PropertyDTO
+updateProperty(PropertyDTO):void
+getPropertyJSON(int):String
+updatePropertyJSON(String):void
+getPropertyAssembler():PropertyAssembler
-instance
0..1
AgentFacade
service.remoteFacade
-AgentFacade()
+getInstance():AgentFacade
+getAgentDTO(int):AgentDTO
+updateAgent(AgentDTO):void
+getAgentJSON(int):String
+updateAgentJSON(String):void
+getAgentAssembler():AgentAssembler
-instance
0..1
-propertyAssembler
0..1
-agentAssembler
0..1
PropertyAssembler
service.remoteFacade
+PropertyAssembler()
+createPropertyDTO(Property):PropertyDTO
+updateProperty(PropertyDTO):void
AgentAssembler
service.remoteFacade
+AgentAssembler()
+createAgentDTO(Agent):AgentDTO
+updateAgent(AgentDTO):void
DTO and its assembler
PropertyDTO
service.DTO
-property_id: int
-type: String
-num_bed: int
-num_bath: int
-num_carpark: int
-date_available: Date
-date_inspection: Date
-description: String
-rent_or_buy: String
-price: int
-address_id: int
-street: String
-city: String
-state: String
-postal_code: int
-country: String
+PropertyDTO()
+getProperty_id():int
+setProperty_id(int):void
+getType():String
+setType(String):void
+getNum_bed():int
+setNum_bed(int):void
+getNum_bath():int
+setNum_bath(int):void
+getNum_carpark():int
+setNum_carpark(int):void
+getDate_available():Date
+setDate_available(Date):void
+getDate_inspection():Date
+setDate_inspection(Date):void
+getDescription():String
+setDescription(String):void
+getAddress_id():int
+setAddress_id(int):void
+getStreet():String
+setStreet(String):void
+getCity():String
+setCity(String):void
+getState():String
+setState(String):void
+getPostal_code():int
+setPostal_code(int):void
+getCountry():String
+setCountry(String):void
+getRent_or_buy():String
+setRent_or_buy(String):void
+getPrice():int
+setPrice(int):void
+getAgentDTO():AgentDTO
+setAgentDTO(AgentDTO):void
+serialize(PropertyDTO):String
+deserialize(String):PropertyDTO
-agentDTO
0..1
AgentDTO
service.DTO
-agent_id: int
-agent_name: String
-agent_email: String
-agent_password: String
-agent_phone: String
-agent_bio: String
-company_name: String
-company_address: String
-company_website: String
+AgentDTO()
+getAgent_id():int
+setAgent_id(int):void
+getAgent_name():String
+setAgent_name(String):void
+getAgent_email():String
+setAgent_email(String):void
+getAgent_password():String
+setAgent_password(String):void
+getAgent_phone():String
+setAgent_phone(String):void
+getAgent_bio():String
+setAgent_bio(String):void
+getCompany_name():String
+setCompany_name(String):void
+getCompany_address():String
+setCompany_address(String):void
+getCompany_website():String
+setCompany_website(String):void
+serialize(AgentDTO):String
+deserialize(String):AgentDTO
Classes containing necessary service operations
UserManagement
service
-userMapper: UserMapper
+UserManagement()
+getUserBasedOnEmail(String):User
+findAgents(String):List<User>
+signup(User):void
+updateProfile(User):void
PropertyManagement
service
-addressMapper: AddressMapper
-propertyMapper: PropertyMapper
-favoriteListMapper: FavoriteListMapper
+PropertyManagement()
+publishProperty(Property,Address):void
+viewMyPropertyList(int):List<Property>
+searchPropertyByFilters(String,String,int,int,int,int,int):List<Property>
+viewSpecificProperty(int):Property
+deleteProperty(int):void
+isPropertyBeingLiked(int,int):boolean
+likeAProperty(int,int):void
+unlikeAProperty(int,int):void
+getMyFavoriteList(int):List<Property>
AppRealm
service
-userMapper: UserMapper
+AppRealm()
#doGetAuthenticationInfo(AuthenticationToken)
#getAuthorizationInfo(PrincipalCollection):AuthorizationInfo
AppSession
service
-USER_ATTRIBUTE_NAME: String
-OTHER_USER_ATTRIBUTE_NAME: String
-PROPERTY_ATTRIBUTE_NAME: String
-PROPERTY_LIST_ATTRIBUTE_NAME: String
-AGENT_LIST_ATTRIBUTE_NAME: String
-LIKE_PROPERTY_ATTRIBUTE_NAME: String
+CLIENT_ROLE: String
+AGENT_ROLE: String
+AppSession()
+hasRole(String):boolean
+isAuthenticated():boolean
+init(User):void
+getSessionId():String
+getUser():User
+getOtherUser():AgentDTO
+setOtherUser(AgentDTO):void
+removeOtherUser():void
+getProperty():Property
+setProperty(Property):void
+removeProperty():void
+getPropertyList():List<Property>
+setPropertyList(List<Property>):void
+removePropertyList():void
+getAgentList():List<User>
+setAgentList(List<User>):void
+removeAgentList():void
+getLikeProperty():boolean
+setLikeProperty(boolean):void
+removeLikeProperty():void

4) Presentation Layer

The presentation layer adopts the front controller pattern. In particular, “FrontCommand” class is the abstract class from all other children “Command” classes to be extended. The “FrontServlet” class dispatches to command object based on the “command” parameter sent along with the HTTP request from the client side.
All children classes extending the “FrontCommand” class are responsible for communicating with the service layer.


FrontServlet
frontController
-serialVersionUID: long
+FrontServlet()
#doPost(HttpServletRequest,HttpServletResponse):void
#doGet(HttpServletRequest,HttpServletResponse):void
-getCommand(HttpServletRequest,String):FrontCommand
-getCommandClass(HttpServletRequest,String):Class
FrontCommand
frontController.commands
#context: ServletContext
#request: HttpServletRequest
#response: HttpServletResponse
+FrontCommand()
+init(ServletContext,HttpServletRequest,HttpServletResponse):void
+process():void
#forward(String):void
UpdateProfileCommand
frontController.commands.postRequest
+UpdateProfileCommand()
+process():void
PublishPropertyCommand
frontController.commands.postRequest
+PublishPropertyCommand()
+process():void
RespondEnquiryCommand
frontController.commands.postRequest
+RespondEnquiryCommand()
+process():void
SignInCommand
frontController.commands.postRequest
+SignInCommand()
+process():void
SignUpCommand
frontController.commands.postRequest
+SignUpCommand()
+process():void
DeletePropertyCommand
frontController.commands.postRequest
+DeletePropertyCommand()
+process():void
ViewEnquiryCommand
frontController.commands.getRequest
+ViewEnquiryCommand()
+process():void
ListPropertiesCommand
frontController.commands.getRequest
+ListPropertiesCommand()
+process():void
ViewOthersProfileCommand
frontController.commands.getRequest
+ViewOthersProfileCommand()
+process():void
ViewProfileCommand
frontController.commands.getRequest
+ViewProfileCommand()
+process():void
SearchPropertyCommand
frontController.commands.getRequest
+SearchPropertyCommand()
+process():void
PropertyInfoCommand
frontController.commands.getRequest
+PropertyInfoCommand()
+process():void
UnlikePropertyCommand
frontController.commands.getRequest
+UnlikePropertyCommand()
+process():void
SignOutCommand
frontController.commands.getRequest
+SignOutCommand()
+process():void
SearchAgentCommand
frontController.commands.getRequest
+SearchAgentCommand()
+process():void
RedirectPublishCommand
frontController.commands.getRequest
+RedirectPublishCommand()
+process():void
LikePropertyCommand
frontController.commands.getRequest
+LikePropertyCommand()
+process():void
IndexPageCommand
frontController.commands.getRequest
+IndexPageCommand()
+process():void
RedirectEditPropertyCommand
frontController.commands.getRequest
+RedirectEditPropertyCommand()
+process():void

Section5: Pattern Discussion

1. Session Patterns – Client Session State

1) Justification

For session patterns, we choose to use client session state pattern. As compared to server session state and database session state patterns, client session state pattern is the easiest to implement. Moreover, it allows low latency since there is no state to be managed on the server side or in database. As we are using a free version of PostgreSQL database in Heroku, the latency issue becomes even critical. Therefore, we decide to reduce latency as much as we can by adopting this pattern.

2) Implementation

All session management is centralized and being done in the AppSession class. The class acts as a manager for the Shiro's subject in the SecurityUtils class. The session gets to be initialized and used right after the user passes the authentication process, and current user object is added into the session. From that point onwards, objects such as "currentProperty" are managed through this class's methods. In the relevant JSP file, client side is able to get these data which are stored in session by calling the "get" methods in AppSession.

2. Input controller Patterns – Front Controller

1) Justification

We decide to use front controller for implementing our web presentation pattern. There are a lot of similar things that need to be done when handling a HTTP request including security, authorization for users and so on. As compared to page controller pattern, front controller pattern consolidates all request handling methods by channeling requests through a single handler object. As a result, only one servlet needs to be created for the whole complex system.

2) Implementation

Within our presentation layer, only one servlet called "frontServlet" is created to consolidate all handling methods for GET and POST request from the client side. There are diverse command objects extending one command abstract class "frontCommand". This servlet then dispatches to command object based on the "command" parameter sent along with the request.

3. View Patterns – Template View

1) Justification

Among template view, transform view and two-step view, we choose to use template view because the other two patterns have high programming and implementation complexity while there is restricting time for this deliverable. Moreover, our system size is not as large as a real enterprise level system, the workload on system maintenance is not that significant. Implementing template view can sufficiently satisfy our need of dynamic content on the web pages.

2) Implementation

Template view supports us to insert Java code in between the HTML code. In some of our JSP, necessary data in the user session is accessed and displayed by embedding markers including “<% %>” and “<%= %>” in the HTML pages. For example, when a propertyDTO is added in the session, the JSP can access the attributes in the propertyDTO through <%= propertyDTO.getPrice() %>.

4. Concurrency Patterns – Pessimistic offline Lock (Exclusive write) & Implicit Lock

1) Justification

There are two reasons why pessimistic lock pattern was selected rather than optimistic offline lock.

a. Provided that the system is not intended to be a high concurrent system, pessimistic offline lock is sufficient and efficient enough to handle concurrent operations and to ensure correctness and data integrity.

b. Unlike the implementation for optimistic offline lock which will cause the expensive change in the database (add extra column for each table to record data version), pessimistic lock is relative cheap and easy to implement for our system. Otherwise, each database table will be changed, which will lead us to change both data mappers and domain layer eventually.

The benefit of using implicit lock is to help us prevent misplacing acquire and release locks by implicitly manage locking tasks in a mapper layer.

Exclusive write lock is chosen to implement because the nature of the real-estate system implies that update operations for a property will not happen very often. As such, it will not degrade the performance, neither cause too much inconsistent reads which this type of lock does not prevent.

2) Implementation

As the class diagram for data source layer shows, an `ExclusiveWriteLockManager` class is created to manage lock acquire and release by maintaining a concurrent HashMap in the database. The key of this map is the data object that should be locked at, and the value is the session ID. For both patterns, we mainly applied those patterns on operations for properties because it is the central data for the whole system.
For implicit lock pattern, we defined a `LockingMapper` class which is to decouple lock acquire&release from mapper. As such, what `LockingMapper` class does is acquire lock, use mapper object to call operations available to that mapper, and release the lock finally. This is only for **delete** method. In order to utilise pessimistic lock at the same time, we manage **update** process locking process at servlet.
For the pessimistic lock pattern, instead of acquiring lock in the mapper layer, we acquire the lock before the servlet retrieves the property from database. (this acquire lock code line is located inside process method of `RedirectEditPropertyCommand` class). Obviously, the release lock happens after the property object has been updated and stored into database. (this release lock line is located inside process method of ` PublishPropertyCommand ` class)

5. Distributed – Data transfer object (DTO) & Remote façade

1) Justification

As it is not required to relocate web server and application server at distributed systems, we just utilise those patterns in order to gain a deeper understanding of them. When we use them in the real-life distributed system, data transfer objects are used to carry data between processes in order to reduce the number of methods calls because it encapsulates multiple primitive data from different classes or other DTO. And remote façade is used together with DTO to translate coarse-grained method calls into fine-grained method calls and collate the results.

As DTOs are designed to be sent over network, they must be serialized into JSON or another format. However, in our case, we just pass object around because all components are located in a same server (not really distributed), even though we have implemented serialize and deserialize methods by using GSON library. Please check façade class code for more details.

2) Implementation

In order to reduce remote calls if our system is applied to distributed systems, we encapsulate address data and property data into \`PropertyDTO\`. However, as a normal property object contains agent information which contains agency company information, we create another \`AgentDTO\` to encapsulate both agent information and agency company data. As such, the core DTO is \`PropertyDTO\` which contains both address data and property data, and the \`AgentDTO\` object. Each corresponding assembler is responsible for assembling a DTO based on given normal property object, and update data based on the information in the DTO.

Finally, each façade provides coarse-grained interface methods over a web of fine-grained objects for remote server.

To utilise DTO to get data, we pass an agent ID to agent remote façade in order to generate an agent DTO which contains both agent data and the company data that she or he works for, and then stores the DTO(big chuck of data) into app session for future use. Please check \` ViewOthersProfileCommand \` class code for more information.

To utilise DTO to update data, we pass a property DTO to a property façade in order to update all relative information about the property. To generate a DTO, we encapsulate all property data (including address ID and agent ID) into a normal property object and pass the property object to property assembler to generate a property DTO. Please check \` PublishPropertyCommand \` class code for more information.

6. Security Patterns - authentication enforcer and authorisation enforcer

1) Justification

Apache Shiro framework provides us a simple and secure way to implement our authentication and authorization process. Authentication enforcer security pattern helps to protect user's and system's information by performing the authentication process in a centralized manner, improving the security by reducing security holes. Authorisation enforcer security pattern avoid duplicate authorization logic code. The responsibility of authorization and authentication can be separately clearly.

2) Implementation

After importing the necessary Shiro JAR libraries, it is able to get a subject object through *SecurityUtils.getSubject()* method and then perform Login authentication on this subject object in a secure way. Any failure in authentication will throw a UnknownAccountException or IncorrectCredentialsException. After passing the Login authentication, the current user object can then be stored into the session.

Authorisation enforcer security pattern is implemented through by assigning current user's role in the AppRealm class. The *getAuthorizationInfo()* method will check whether the current user’s class type is either “Agent” or “Client” and then add corresponding role to it. After this step, we can validate current user’s role through *AppSession.hasRole()* method.

3) **Why was Intercepting Validator Pattern not implemented in this project?**
   First of all, HTML 5 supports to define the input’s type, minimum value and maximum value (for number type only). All these features allow us to restrict user inputs on the client side. This largely decreases the possibility for attackers to attack our system through sending data.
   Moreover, in our data mappers, all SQL executions are done through “*PreparedStatements*” which avoids/prevents SQL Injection.
4) **Why was Secure Pipe patterns not implemented in this project?**
   We do not implement secure pipe security pattern because our system has been deployed on Heroku which uses HTTPS requests which is a more secure version than HTTP as the connections are encrypted, minimizing the possibility for malicious attacks. Therefore, secure pipe pattern seems redundant to be implemented for our system.