Architectural Pattern Document

Submission Part 2 Specification

[404 Not Found]
SWEN90007 SM2 2023 Project

Members

1. Jun Xu (1074120) juxu1

Github username: JunXu-1

juxu1@student.unimelb.edu.au

2. Shuowen Yu (1174223) shuoweny

Github username: shuoweny

shuoweny@student.unimelb.edu.au

3. Yuchen Cao (1174014) yuccao

Github username: YuchenCAO01

yuccao@student.unimelb.edu.au

4. Zhaolong Meng (1118637) zhaolongm

Github username: kdyxnm

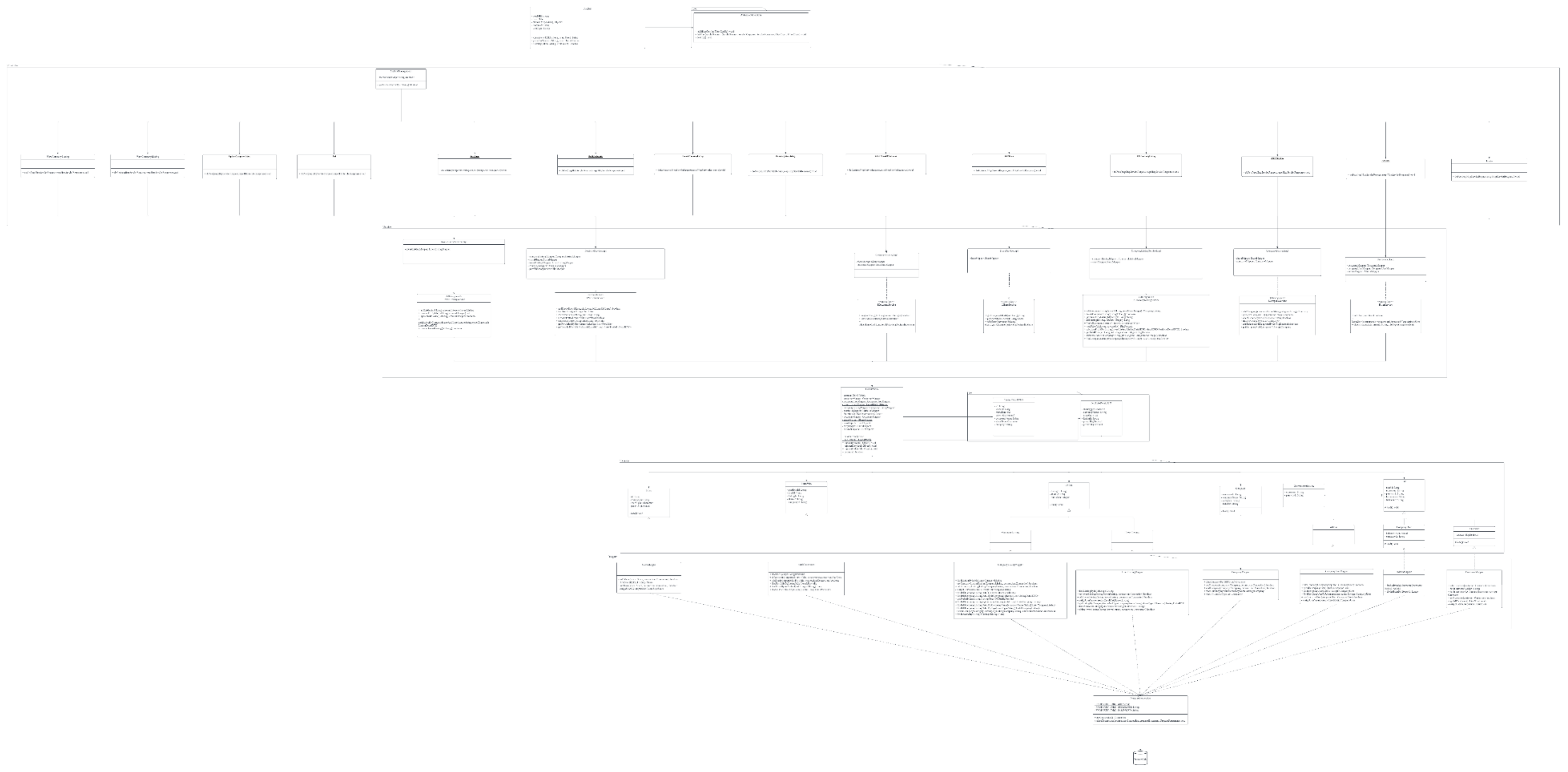
zhaolongm@student.unimelb.edu.au

Revision History

Date	Version	Description	Author
13.9	02.00.D1	setup basic report template and write the description about data mapper pattern	Shuowen Yu
13.9	02.00.D1	write the justification for identity field, foreign key mapping, association table mapping, authentication and authorization	Jun Xu
14.9	02.00.D2	write the description about concrete table inheritance and embedded value pattern	Shuowen Yu
14.9	02.00.D2	write the description of the identity field	Jun Xu
15.9	02.00.D3	write the description about Unit Of Work Pattern	Shuowen Yu
15.9	02.00.D3	write the description for foreign key mapping, association table mapping, authorization and authentication patterns	Jun Xu
16.9	02.00.D4	refactor the document structure based on the tutor's feedback and write the description about lazy load	Shuowen Yu
17.9	02.00.D5	draw class diagram according to code structure	Zhaolong Meng
18.9	02.00.D6	write the description for the domain model	Yuchen Cao
19.9	02.00	Review and Edit the whole document and ready to submit	Shuowen Yu, Jun Xu, Zhaolong Meng, Yuchen Cao

Contents

1. Class Diagram for the whole system (Link: clear diagram check here)



2. Architectural Patterns

2.1 Data Mapper Pattern

Pattern	**Reason**
Data Mapper Pattern	Justification: The data mapper pattern acts as a middleware that is used to communicate with the database and assign the value to the domain object. This approach satisfies the low coupling pattern by ensuring that both domain objects and database systems are separated. This separation allows us to develop independently without considering the complexity of each other. Additionally, its high cohesion ensures that the domain and data mapper focus solely on their respective responsibilities. The domain only has the domain logic that makes our code structure clear and maintainable. The data mapper is highly compatible with the domain model pattern, which we want to use in our project. Comparison: Row data gateway is not a good option for us. Although it provides an easy way to deal with a single row of data in the database, considering we are not familiar with the data design, we might need to constantly change our database structure. This would necessitate regular updates and maintenance of our domain. Therefore, the row data gateway is not a choice for us. For the table data gateway, it is incompatible with the domain model pattern. For the active record pattern, based on its lack of scalability, it cannot handle situations well when our domain model has a lot of inheritance. Thus, it is better for us to use the data mapper instead of other patterns.

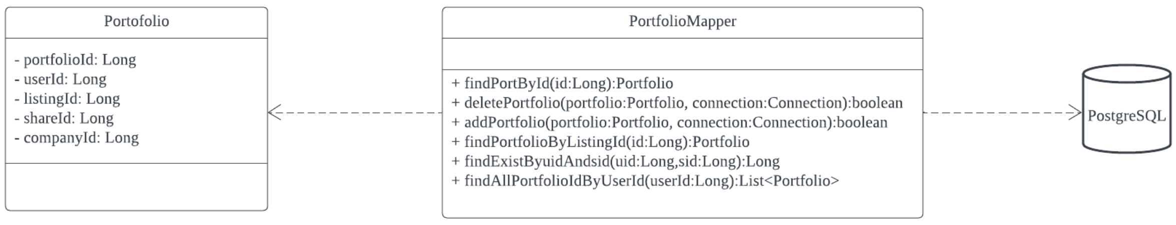
2.1.1 Implementation

To implement the data mapper pattern, we have constructed the following mapper classes.

Mapper Class	**Domain Object Class**
AdminMapper	Admin
CompanyListingMapper	CompanyListing
CompanyMapper	Company
CompanyUserMapper	CompanyUser
CustomerMapper	Customer
InvestListingMapper	InvestListing

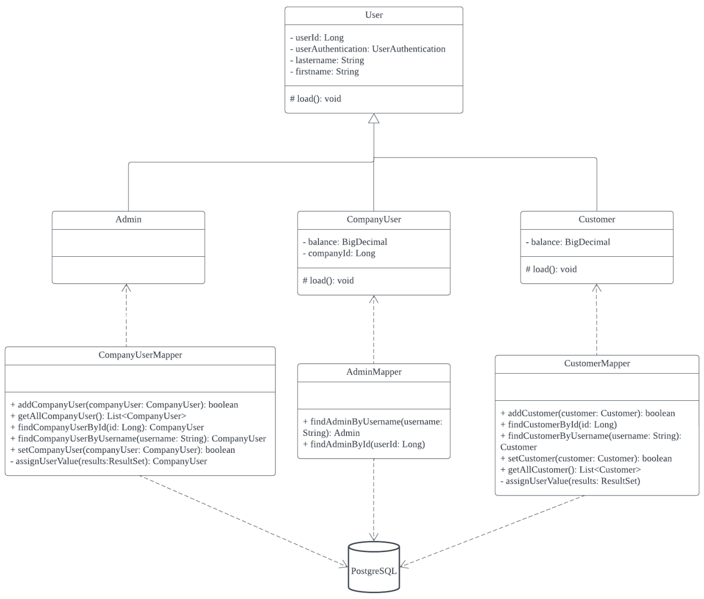
PortfolioMapper	Portfolio
ShareMapper	Share

The diagram below demonstrate the implementation of the data mapper. The data mapper class retrieves data from its corresponding database table and maps it to a specific domain object. Conversely, domain objects can also map their values back to the database. This ensures our mapper and domain are both testable and maintainable. The most common processes of operations are "find", "add" and "update".


Portofolio
- portfolioId: Long
- userId: Long
- listingId: Long
- shareId: Long
- companyId: Long
PortfolioMapper
+ findPortById(id:Long):Portfolio
+ deletePortfolio(portfolio:Portfolio, connection:Connection):boolean
+ addPortfolio(portfolio:Portfolio, connection:Connection):boolean
+ findPortfolioByListingId(id:Long):Portfolio
+ findExistByuidAndsid(uid:Long,sid:Long):Long
+ findAllPortfolioIdByUserId(userId:Long):List<Portfolio>
PostgreSQL

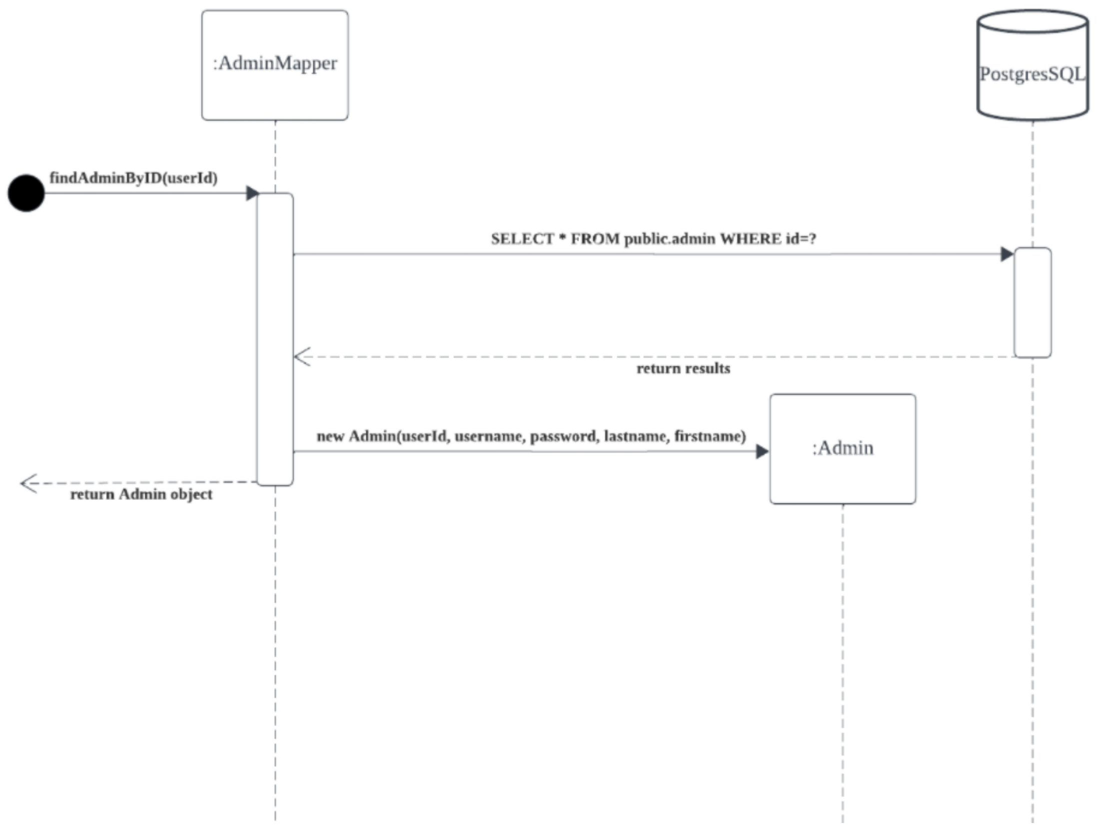
2.1.1.1 class diagram

Since the implementation logic for all of our data mapper classes is the same, we will use the user-related design structure as a representative example to fully illustrate our approach when the inheritance happens.


User
- userId: Long
- userAuthentication: UserAuthentication
- lastername: String
- firstname: String
# load(): void
Admin
CompanyUser
- balance: BigDecimal
- companyId: Long
# load(): void
Customer
- balance: BigDecimal
# load(): void
CompanyUserMapper
+ addCompanyUser(companyUser: CompanyUser): boolean
+ getAllCompanyUser(): List<CompanyUser>
+ findCompanyUserById(id: Long): CompanyUser
+ findCompanyUserByUsername(username: String): CompanyUser
+ setCompanyUser(companyUser: CompanyUser): boolean
- assignUserValue(results:ResultSet): CompanyUser
AdminMapper
+ findAdminByUsername(username: String): Admin
+ findAdminById(userId: Long)
CustomerMapper
+ addCustomer(customer: Customer): boolean
+ findCustomerById(id: Long)
+ findCustomerByUsername(username: String): Customer
+ setCustomer(customer: Customer): boolean
+ getAllCustomer(): List<Customer>
- assignUserValue(results: ResultSet)
PostgreSQL

2.1.1.2 sequence diagram

In the sequence diagram, we will illustrate how the Admin domain class can retrieve information using its ID through the AdminMapper. This would be an example of the data mapper behavior, as they have similar implementations.

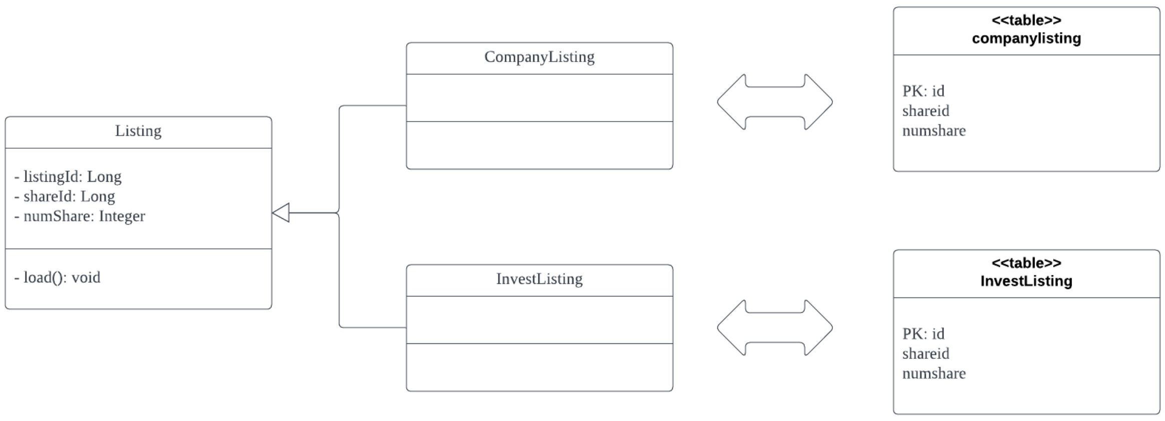

:AdminMapper
PostgresSQL
findAdminByID(userId)
SELECT * FROM public.admin WHERE id=?
return results
new Admin(userId, username, password, lastname, firstname)
:Admin
return Admin object

2.2 Concrete table Inheritance Pattern

Pattern	**Reason**
Concrete Inheritance Pattern	Justification: By using the concrete table inheritance pattern, we can allocate one table for each domain class in our project and store its own data to clearly define the responsibilities of each class. This highest degree of decoupling combined with high cohesion is beneficial for our system. Although many tables have similar columns, It helps us to separate the different purposes. For instance, company listing indicates the shares companies intend to sell, while invest listing records the shares customers have purchased. Processing these shared variables in one table would be complex as we need to identify the role of the data. Similar to user-related tables, they might have the same username and password but the role is different. The concrete table inheritance

	can help us solve this complexity and increase the query speed as we will only search data in a specific role table instead of the whole table. Comparison: Single table is not a suitable option for us. It combines all the data into one table leading to low cohesion, and it is a challenge to process data based on the different roles. The class table inheritance has low cohesion on storing the shared column data in the same table, and the high coupling leads to frequent join to get the extra column will reduce the efficiency.

2.2.1 Implementation



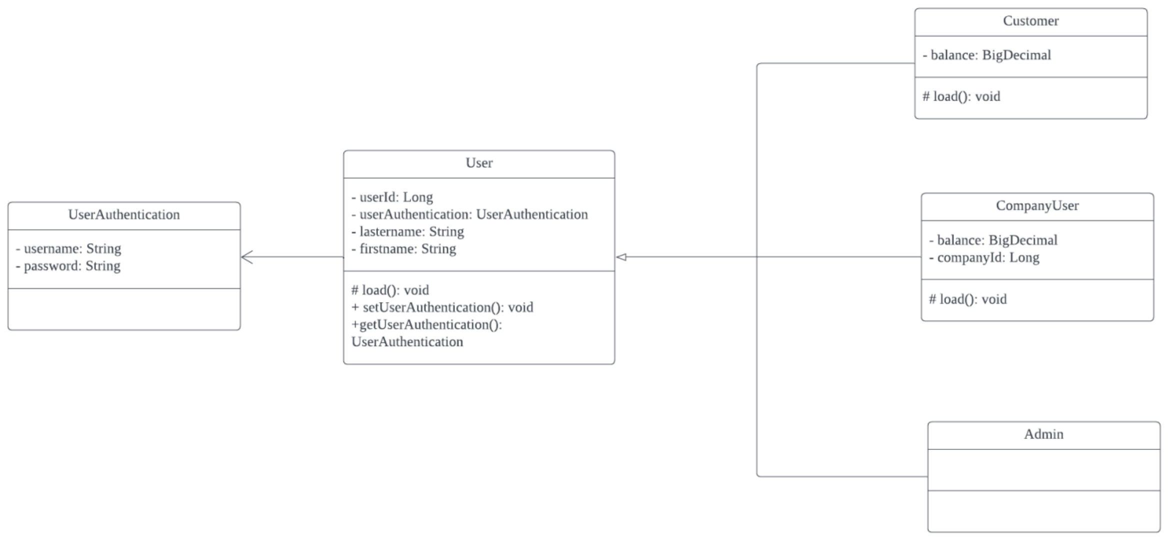
We have parent domain classes named User class and Listing class. The picture above shows the listing case. The domain classes that inherit from these parent classes have their own table to store their data. The following table shows the mapping relationship between domain class and table. We choose not to apply the inheritance in our data mapper class to keep a clear responsibility. Each data mapper will be distinct and independent. If a new function is added in the future, we can modify the mapper independently without affecting the other mapper.

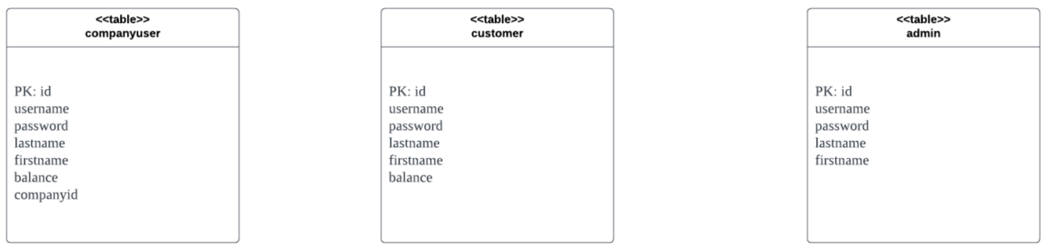
Parent Domain Class	Domain Class	Table name
User	Admin	admin
	CompanyUser	companyuser
	Customer	customer
Listing	InvestListing	investlisting
	CompanyListing	companylisting

2.3 Embedded Value Pattern

Pattern	Reason
Embedded Value Pattern	By using the embedded value pattern, our embedded value class will only interact with the class that embedded it. Other classes can utilize the getters and setters to modify the embedded value within the main class. This pattern ensures the main class remains highly cohesive as it retains its primary responsibility, while distributing distinct responsibilities to achieve low coupling. In our system, it is important for our system to use as there is lots of different information in the user class, such as personal details and authentication data like username and password. Storing them directly within the User class is not a good design as the user authentication might undergo different logical changes in the future, such as password hashing for security purposes. By isolating authentication into its own class, it ensures that future modifications will not impact much in the User class directly, as the interactions primarily involve getter and setter from the Authentication class.

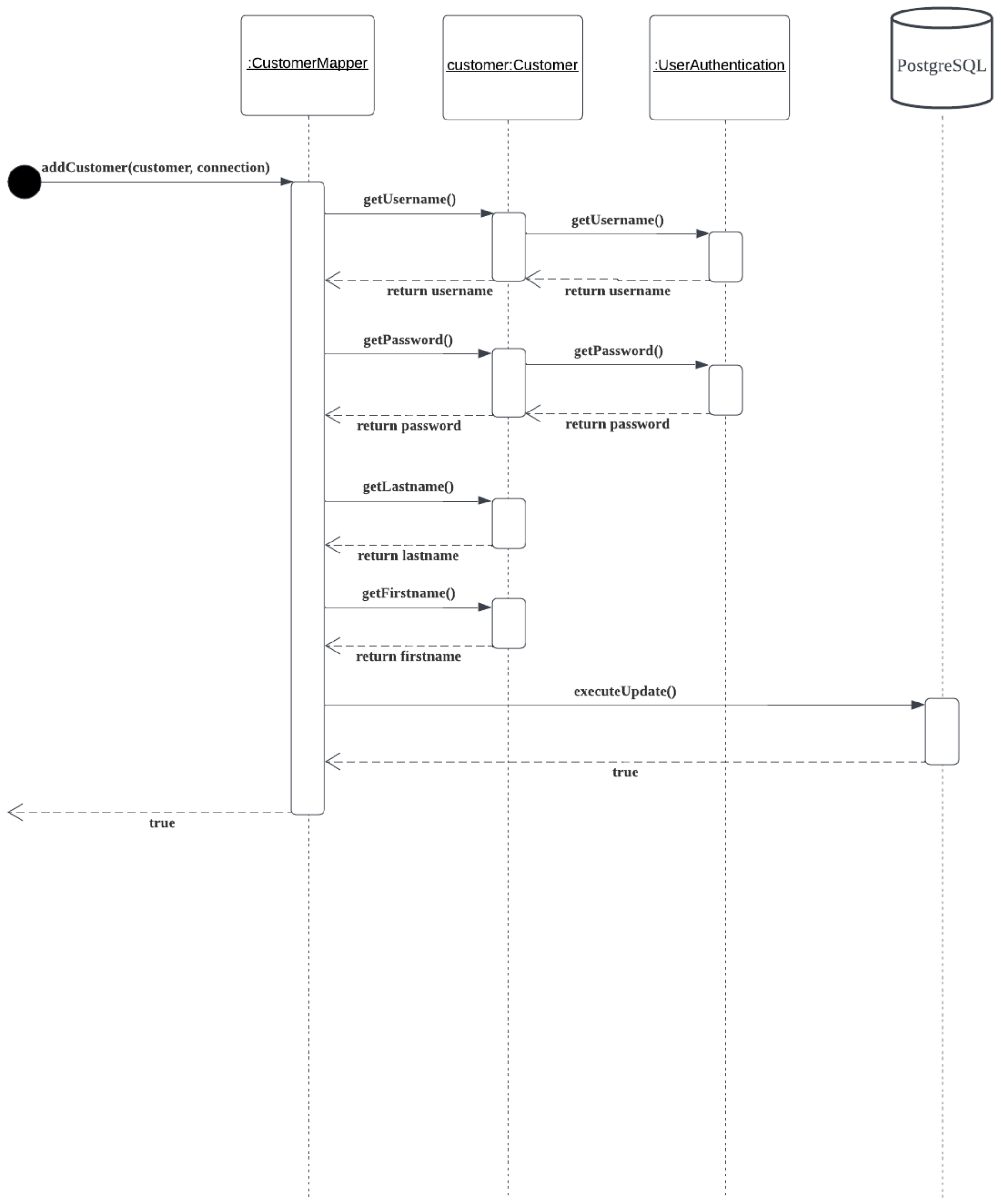
2.3.1 Implementation





To implement the embedded value pattern, we create a class called UserAuthentication. It holds the username and password for a User. We do not create a table to store the UserAuthentication data in our database as it is part of the user data. When we require the username and password, we will instantiate a new UserAuthentication object and use the setter method in User class to update the respective information. Also, when the mapper retrieves data from the dataset, it will pass all the information to User class and the User will transfer the username and password to UserAuthentication to ensure low coupling and high cohesion in our design. The UserAuthentication is able to apply the lazy load in the User class as we do not always use this data.

We will illustrate the behavior of mapper, Customer and UserAuthentication by using the sequence diagram. The sequence is triggered when UnitOfWork invokes the addCustomer(customer, connection) method on CustomerMapper. This method will gather all the information from the customer and update it to the database. In this scenario, we will only consider the success case.



2.4 Unit Of Work

Pattern	Reason
Unit Of Work	By using the unit of work, we are able to commit all the changes in a single request together. This method follows the principles of atomicity and consistency in the ACID (Atomicity, Consistency, Isolation, Durability). If the commit fails, we are able to roll all the changes back, which ensures the data is reliable and consistent. It is an important feature for our system as we are mainly focused on purchasing and selling the shares. Systems related to money must ensure their consistency to protect our customers' rights. The unit of work also has low coupling and high cohesion as it has centralized transaction management, all the changes to the database can be managed in one place. This approach will make it easier for us to maintain and modify. Also, the efficiency of the system will improve as it only requires one connection to handle all of the changes during one request.

2.4.1 Design Rationale

When integrating the unit of work pattern, we aimed to maintain a clear structure and responsibility. Each part should have a distinct role to minimize confusion among team members. The service handles business logic, while the controller interacts with various services and the UnitOfWork. This approach highlights high cohesion that can allow us to easily implement the new feature or make adjustments without disrupting other parts of the system. It can efficiently reduce the potential conflict when team members work on different sections at the same time.

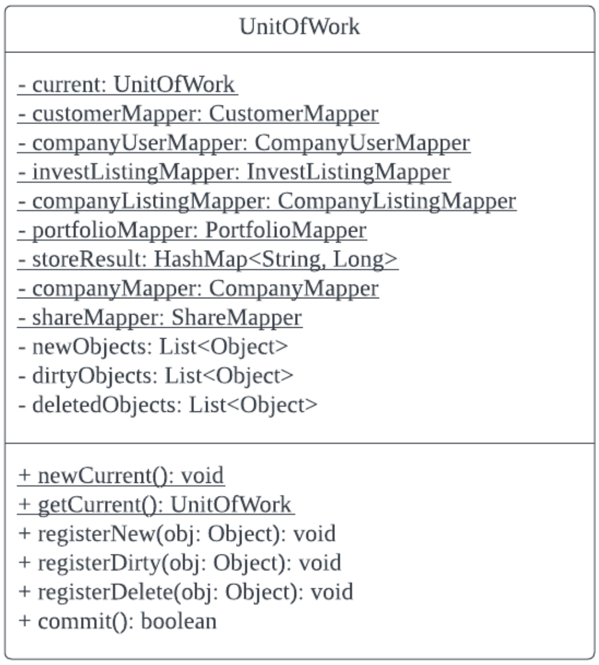
We chose to establish a new connection to the database in the ‘commit’ method rather than directly in the controller to ensure the centralized error handling. We can simply handle exceptions using try-catch within the commit method and no further modification on controller class.

2.4.2 Implementation

The heart of the unit of work pattern in our system is the ‘UnitOfWork’ class. This class is designed to process any changes to the database within the same database connection which allows the consolidated commit or rollback for each request. The ‘UnitOfWork’ class is used in the following classes.

Classes
AddCompanyController
AddCompanyListingController
AddShareController
DeleteCompanyListingController
PurchaseController
SellController
updateCompanyListingController

Key Components for implementation:


UnitOfWork
- current: UnitOfWork
- customerMapper: CustomerMapper
- companyUserMapper: CompanyUserMapper
- investListingMapper: InvestListingMapper
- companyListingMapper: CompanyListingMapper
- portfolioMapper: PortfolioMapper
- storeResult: HashMap<String, Long>
- companyMapper: CompanyMapper
- shareMapper: ShareMapper
- newObjects: List<Object>
- dirtyObjects: List<Object>
- deletedObjects: List<Object>
+ newCurrent(): void
+ getCurrent(): UnitOfWork
+ registerNew(obj: Object): void
+ registerDirty(obj: Object): void
+ registerDelete(obj: Object): void
+ commit(): boolean

1. Static Instance:

When any of these controllers receive a request from the frontend, the new static instance of UnitOfWork will be created by calling the 'newCurrent()' method. By using this static instance, we can make sure the UnitOfWork object can be easily accessed in different classes, almost like a global variable.

2. Storing Listings:

We have three lists in our UnitOfWork named ‘newObjects’ used to store new objects that need to be added during each request, ‘dirtyObjects’ used to store the objects that need to update its value in database, ‘deleteObjects’is to store the object that need to be deleted in the database during each request. These three lists can help us iterate and manage the changing objects by calling the corresponding mapper method.

3. Service:

We use services to execute the business logic for each operation. If our logic determines that the user can proceed with add, update, or delete operations. The services will create the object and use methods like 'registerNew', 'registerDirty', or 'registerDelete' to queue these objects for processing.

4. Commit Changes:

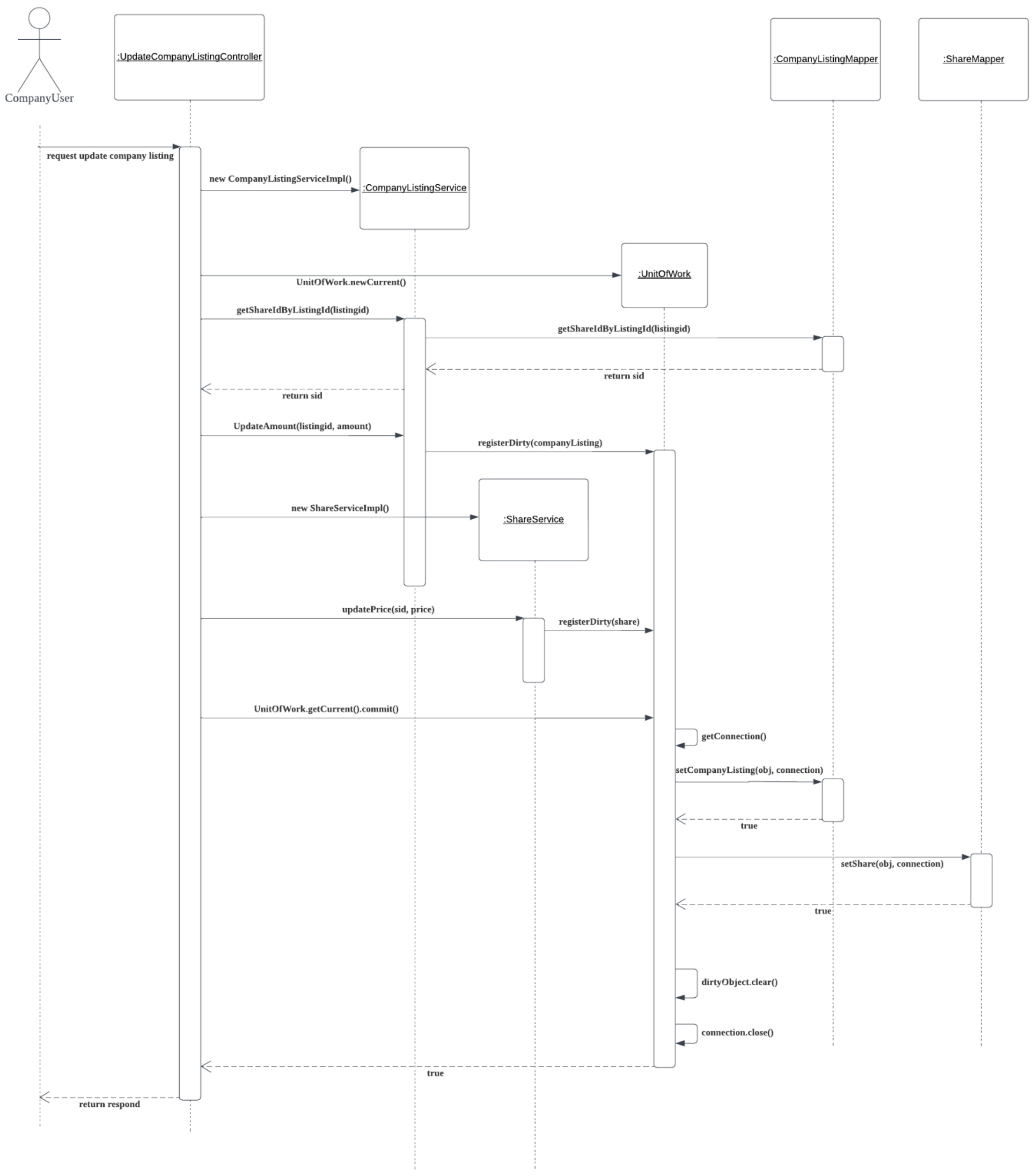
After all changes are registered, the controller calls the ‘UnitOfWork.commit()’ method. This commit method opens a new connection to the database. It then iterates through all three lists, sending each changed object to the appropriate mapper method under the same connection. Once processed, the lists are cleared for the next request. If any operation fails, the exception can be catched in the commit method, and the system will rollback all changes.

The sequence diagram illustrates the interaction behavior between the UnitOfWork, controller, and Service. Since the implementation is consistent across all controllers related to UnitOfWork, this diagram will specifically showcase the behavior of the updateCompanyListingController.

5. Storing Result Hashmap:

We create a hashmap to record the result we want after or during the commit. In some cases, the upcoming operation needs the result from the last operation during commit. The frontend may also need some results after the commit. We will clear this hashmap in the controller for the next request.

The sequence diagram illustrates the behavior of the UpdateCompanyListingController. This is the representative case that every other listed controller follows.


CompanyUser
:UpdateCompanyListingController
:CompanyListingMapper
:ShareMapper
request update company listing
new CompanyListingServiceImpl()
:CompanyListingService
:UnitOfWork
UnitOfWork.newCurrent()
getShareIdByListingId(listingid)
getShareIdByListingId(listingid)
return sid
return sid
UpdateAmount(listingid, amount)
registerDirty(companyListing)
new ShareServiceImpl()
:ShareService
updatePrice(sid, price)
registerDirty(share)
UnitOfWork.getCurrent().commit()
getConnection()
setCompanyListing(obj, connection)
true
setShare(obj, connection)
true
dirtyObject.clear()
connection.close()
true
return respond

2.5 Lazy Load

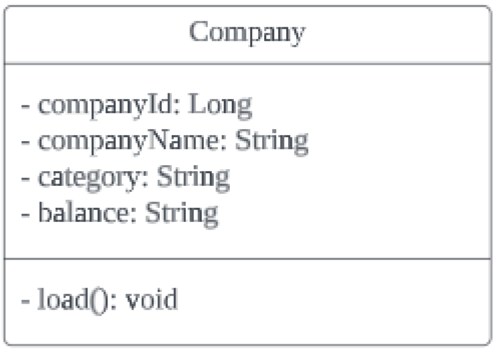
Pattern	Reason
Lazy Load	Justification: The lazy load can optimize the system performance. The system can delay loading the unnecessary data we do not want until we actually need it. This approach will minimize the usage of memory and the user will not wait a long time for loading too

	much unnecessary data. The ghost implementation is a better choice for our system as it will load all the null fields when the getter method is getting the null attribute. This approach will reduce the number of connections with databases as to get the specific value for each connection will slow the process time when the user needs more than one information. Also, Its simplistic and low coupling can make our system clear and maintainable.

2.5.1 Design Rationale

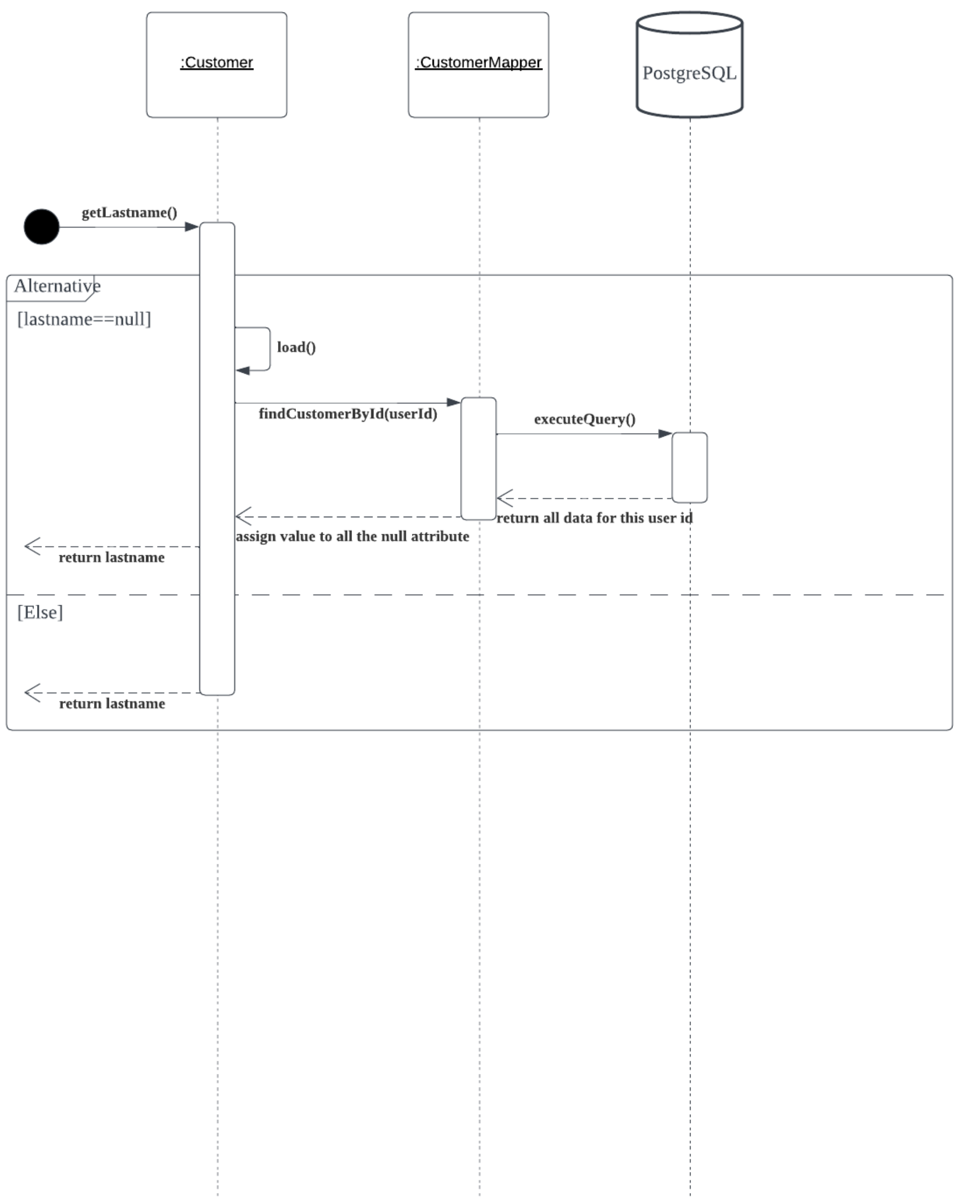
In our system, we are using the ghost implementation. It allows the system to automatically get the full value of the object when it is needed by invoking the getter method directly. The lazy load will be maintained in the domain object class which leads to a low coupling as each domain model is independent. If we want to modify the lazy load for a specific domain object we can directly modify it in the corresponding class.

2.5.2 Implementation



We add the load method for each domain class and verify if it is null in each getter method. Then we will load all the attributes again in the load() method based on its id.

We will illustrate the behavior of lazy loading using a sequence diagram. Since all implementations are similar, we will showcase only one scenario: the process when calling the getLastname() method for customers and userid is exists.



2.6 Identity Field

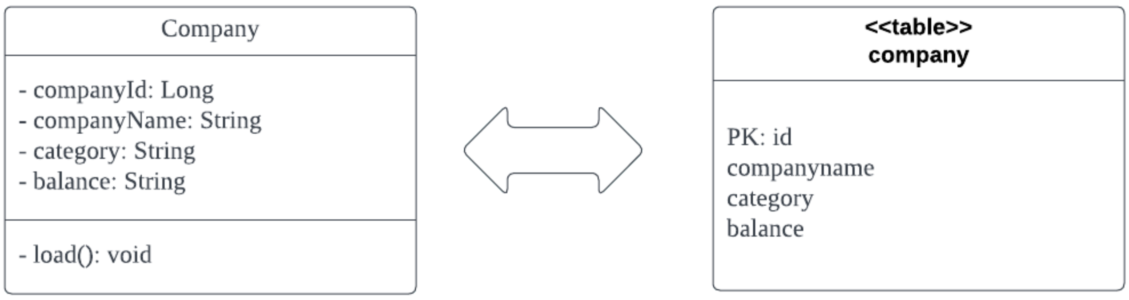
Pattern	Reason
Identity Field	Justification: Identity field pattern is used to identify the specific row by a unique field. Our system will use meaningless, simple and table-unique keys as we would like to use ids generated

	automatically by the database when a new record is inserted into the database for efficient purposes. For each table, the id will be unique for us to select each row. For example, the method 'findAdminById' uses id as the identity field to find the expected row about the user details. We can easily link the domain object with the corresponding row in the database by the identity field. The identity field pattern is really simple. And we can use the identity field to get the specific row we want which increases the efficiency of the database.

2.6.1 Implementation

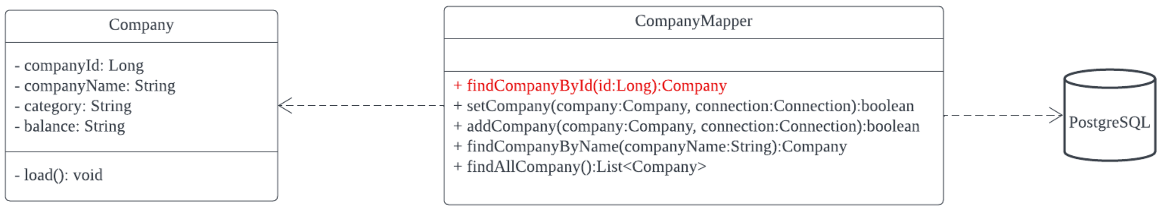
We use the id generated automatically by the database as each identity field to distinguish the different rows in one table. Whenever a new row is created, the identity field is generated as well.

Tables in the database	Identity Field
customer	id
company	id
companyuser	id
admin	id
share	id
companylisting	id
investlisting	id
portfolio	id



According to the domain class of the company, we will generate the company table in our database and the primary key is the id field. The picture above shows the corresponding implementation and we can find the specific row by id.

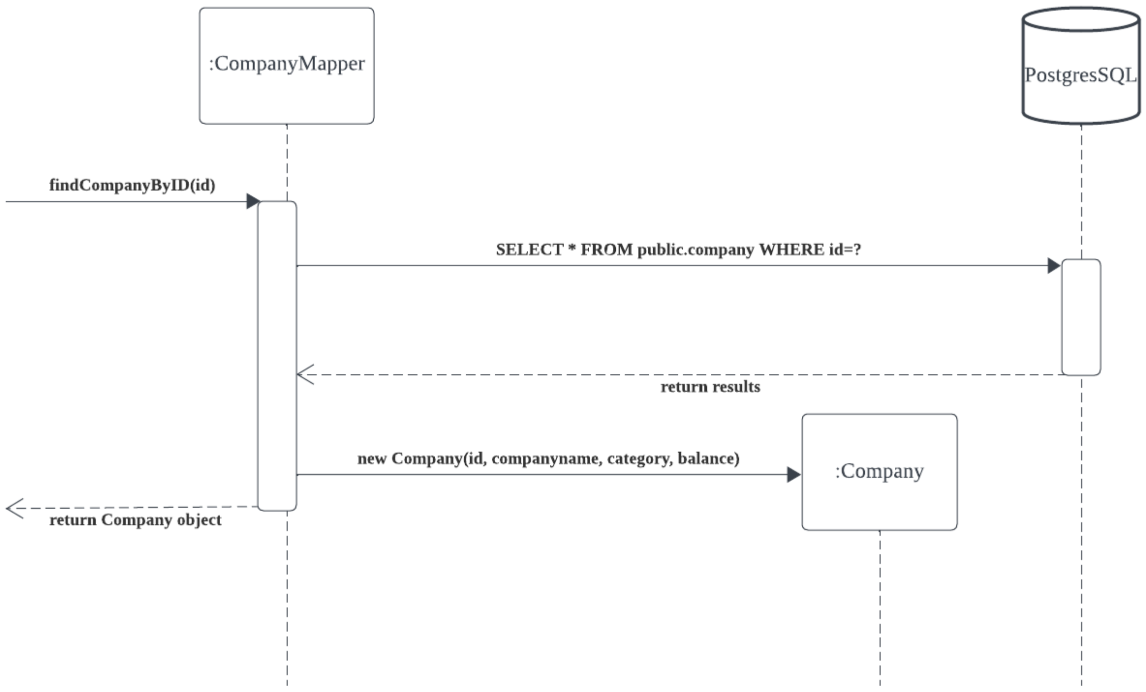
2.6.1.1 class diagram


Company
- companyId: Long
- companyName: String
- category: String
- balance: String
- load(): void
CompanyMapper
+ findCompanyById(id:Long):Company
+ setCompany(company:Company, connection:Connection):boolean
+ addCompany(company:Company, connection:Connection):boolean
+ findCompanyByName(companyName:String):Company
+ findAllCompany():List<Company>
PostgreSQL

For almost every domain class, we have a method like findCompanyById. This method basically uses the identity field ‘id’ to select the specific row in the database.

2.6.1.2 sequence diagram

In the sequence diagram, we will illustrate how we get a row of the company data from the database by the identity field id. This would be a great example of the identity field pattern.



After we get a specific id of the company, we can use findCompanyById method in the companyMapper to select the details of the company which is a row in the database. For example, if one row is:

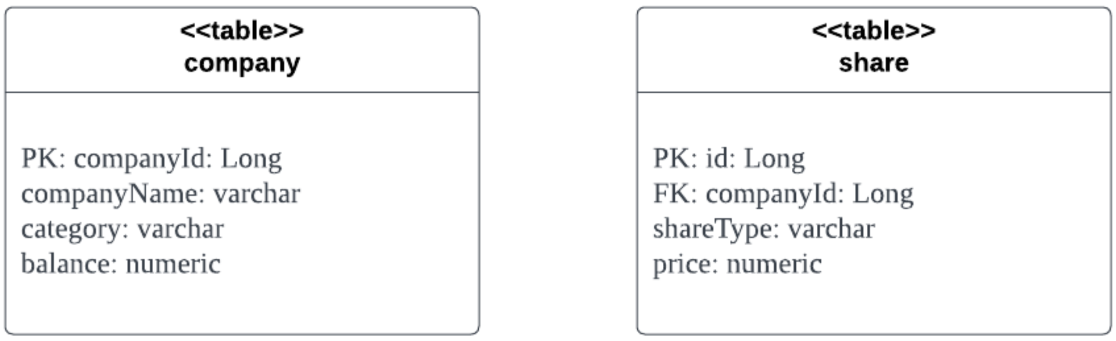
id	companyname	category	balance
1	Google	Internet	xxxxxxxx

We can use the id of ‘1’ to get this row as expected.

2.7 Foreign Key Mapping

Pattern	Reason
Foreign Key Mapping	Justification: While the identity field pattern is used to map domain objects with their corresponding rows in the database, the foreign key mapping is used to link between the tables in the database. For example, the company listings in our system store the share object. The share object stores the type and the price of the share. The company listing will show this information to the customers so that customers can decide to purchase the share or not. We can use the foreign key shareId to link the companyListing table with the share table in the database, and find out the specific row in the share table. The share id is the identity field for the share table in the database. Each company listing will store a share id as the foreign key to show the share details. Foreign key mapping pattern joins two tables from the database easily. We can get information like the name of the company for the share by the joined tables.

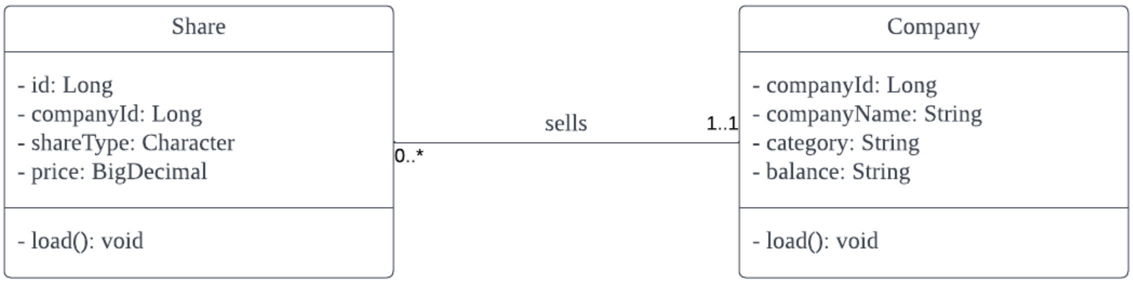
2.7.1 Implementation



We decided to store the companyId in the share table because we want to know which company the share belongs to. The companyId in the company table is the primary key and the companyId in the share table is the foreign key. It is easy for us to use queries to get what I want by using the foreign key mapping pattern.

2.7.1.1 class diagram

As the foreign key mapping pattern is mainly used to combine 2 tables of the database, we decided to use a class diagram to show this pattern. 'CompanyId' is the foreign key stored in the Share class. By using the companyId, we can join the share table and the company table together to get useful information.



For example:
Share Table

id	**companyid**	**shareType**	**price**
1	1	A	50

Company Table

id	**companyname**	**category**	**balance**
1	Google	Internet	xxxxxxxx

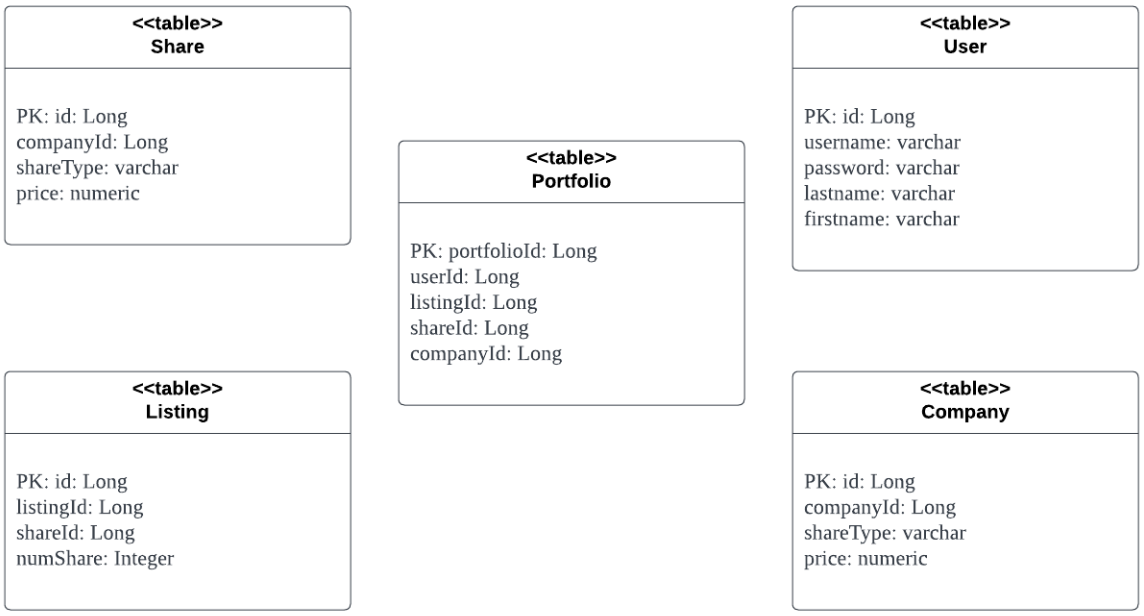
Combined Table by foreign key mapping

shareid	**shareType**	**price**	**companyid**	**companyname**	**category**	**balance**
1	A	50	1	Google	Internet	xxxxxxxx

2.8 Association Table Mapping

Pattern	**Reason**
Association Table Mapping	Justification: Association table mapping is used to create a table with foreign keys to store the association between 2 related tables in the database. In our system, the portfolio table in the database is created to link the customer table with the invest listing table. Each customer should have multiple invest listings because one customer can buy different types of shares for multiple companies. The portfolio table stores one user id and one invest listing id as foreign keys in the database. Corresponding rows can be selected correctly by the association table mapping. As a result, users can view their own invest listings by their portfolios. Association table mapping pattern makes our work easier to find associated information from the database. We can easily find out which company is the share from in the invest listing by the portfolio table.

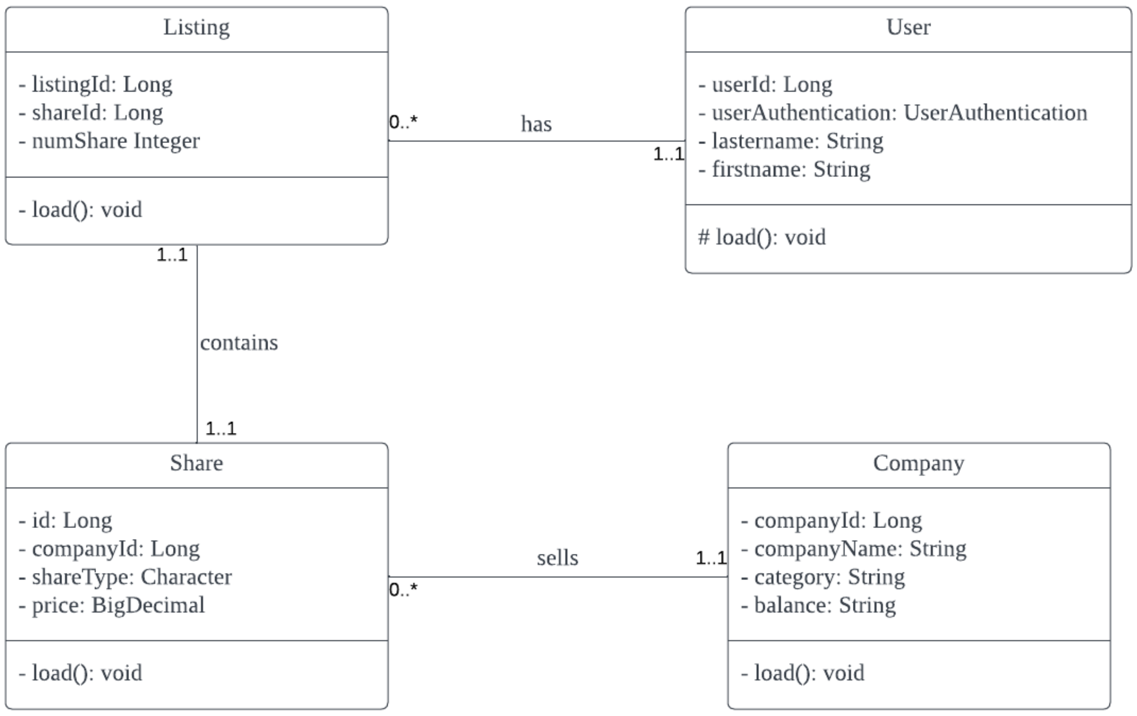
2.8.1 Implementation



We have the portfolio class which shows the mapping between the tables. Portfolios are used to store the information of customers' invest listings. Each row of the portfolio table in the database represents the association between a customer, the invest listings of the customer, the share stored in the listings and the company which sells the share. Foreign keys stored in the portfolio are the primary keys of other tables. This pattern provides the flexibility for us to get information from the database.

2.8.1.1 class diagram

The class diagram shown below shows the relationship between the user class, the listing class, the share class and the company class. And the portfolio class is representative for the association mapping pattern.



Portfolio Table Example

id	companyid	shareid	userid	listingid
1	1	1	1	1

2.9 Authentication and Authorization

Pattern	Reason
Authentication and Authorization	Justification: By using authentication and authorization patterns, I can make sure that the users are the exact same people as who they say they are and also only allow users to do what they can do. Annotations will be applied on the controllers to check if the user has logged into the system and whether the type of the user is allowed to do specific actions. Comparison: Intercepting Validator is used to ensure the requests are well-formed and non-malicious which requires more time and resources. However, our team has limited resources and our system has already used authentication and authorization to handle most of the secure problems. As a result, we do not consider the intercepting validator at this stage. As our team has decided to use Json Web Token to encrypt and decrypt the transition data, a secure pipe pattern will increase the

	cost of maintainability when it is not necessary. Therefore, our team chooses not to implement the secure pipe pattern to make it more efficient.

2.9.1 Implementation

Authentication

We use the json web token to generate the token which is used to verify the authentication of the user by the jwtUtil class which is shown below. When the user does not log into the system, the user should have an annotation called “loginToken”. After the user logs in, the user will be provided with an annotation called “normalToken”. Any action except login needs to contain the ‘normalToken’.

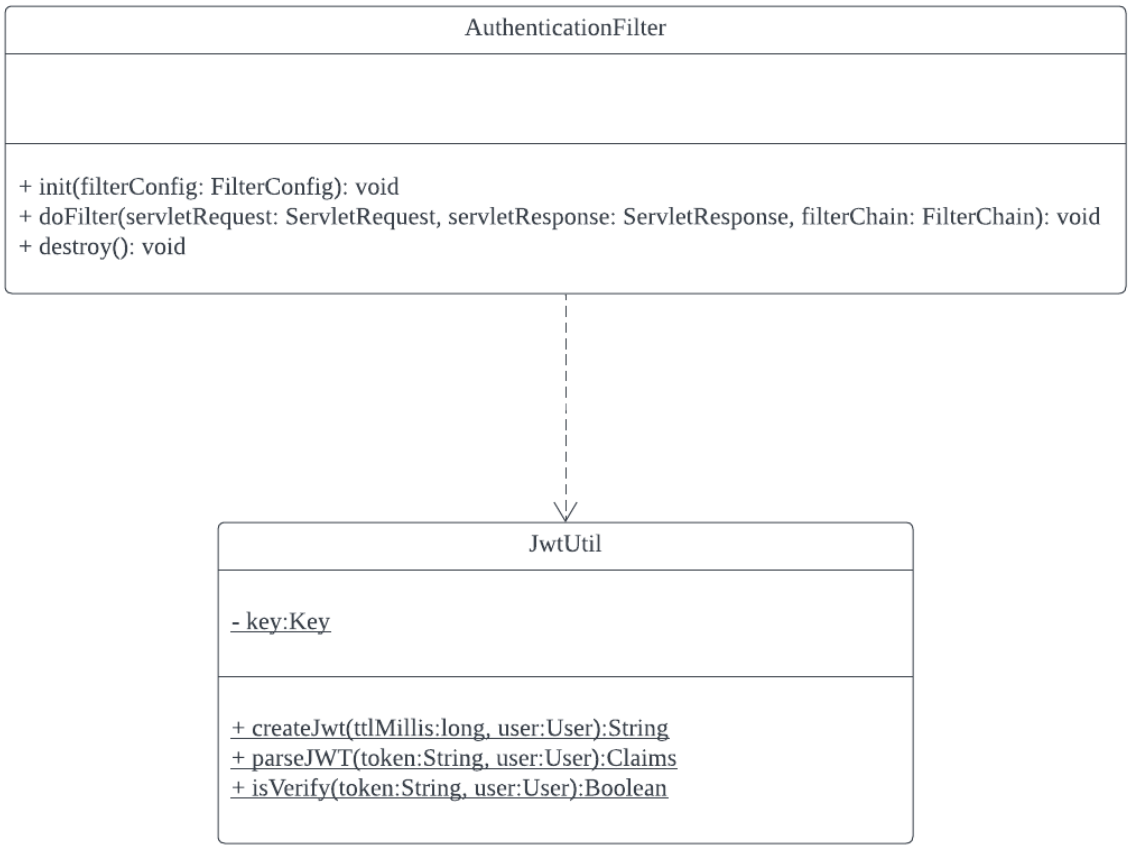
Authorization

In addition, we have a token called “rolesAllowed” to do authorization. It stores a list of valid user types. If the user’s type is in the list, then the user is allowed to do the action.

If the user has the annotations but does not meet the conditions, the user will not be able to do the action by the authenticationFilter shown in the class diagram.

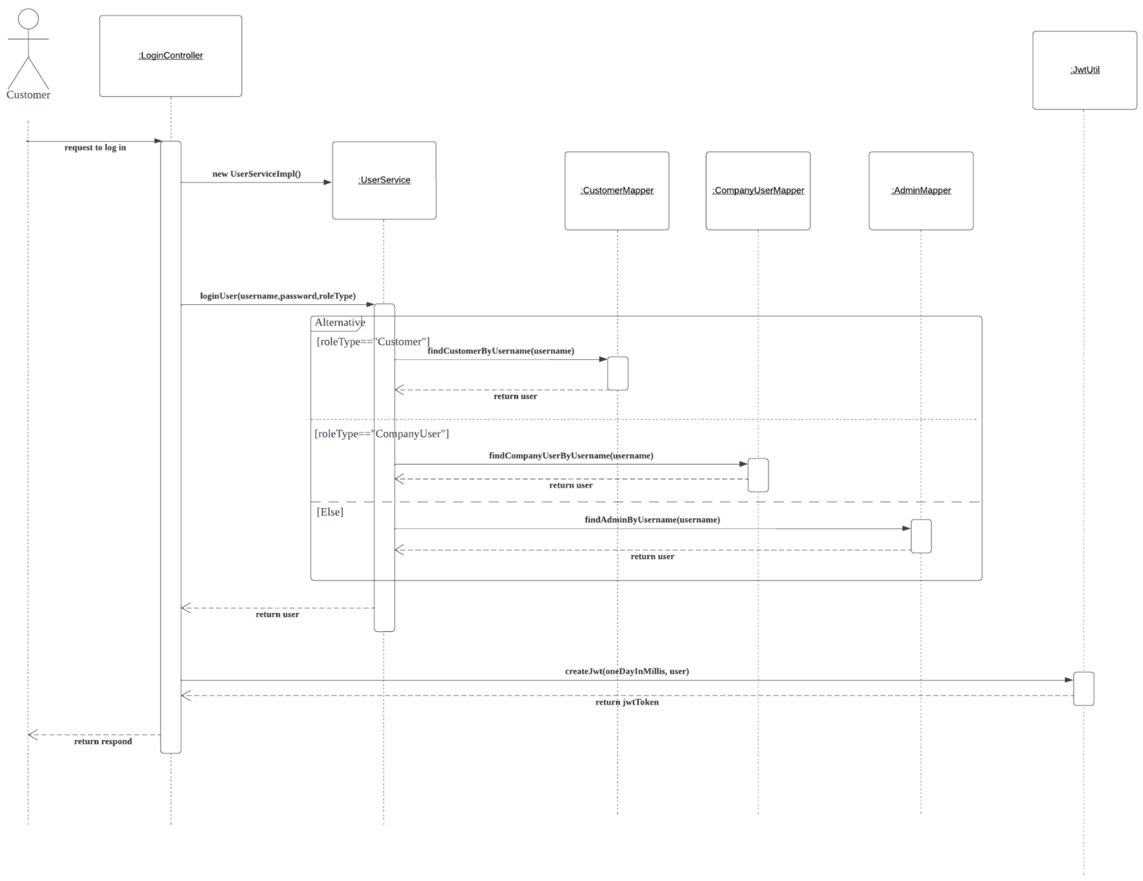
2.9.1.1 class diagram

The class diagram shows the process of the authentication and authorization. After the user logs into the system, the jwt token will be generated and sent back in the response. The user needs to use the token to take actions that they are allowed to. Then the authentication and authorization are ensured to make users only do what they can do.


AuthenticationFilter
+ init(filterConfig: FilterConfig): void
+ doFilter(servletRequest: ServletRequest, servletResponse: ServletResponse, filterChain: FilterChain): void
+ destroy(): void
JwtUtil
- key:Key
+ createJwt(ttlMillis:long, user:User):String
+ parseJWT(token:String, user:User):Claims
+ isVerify(token:String, user:User):Boolean

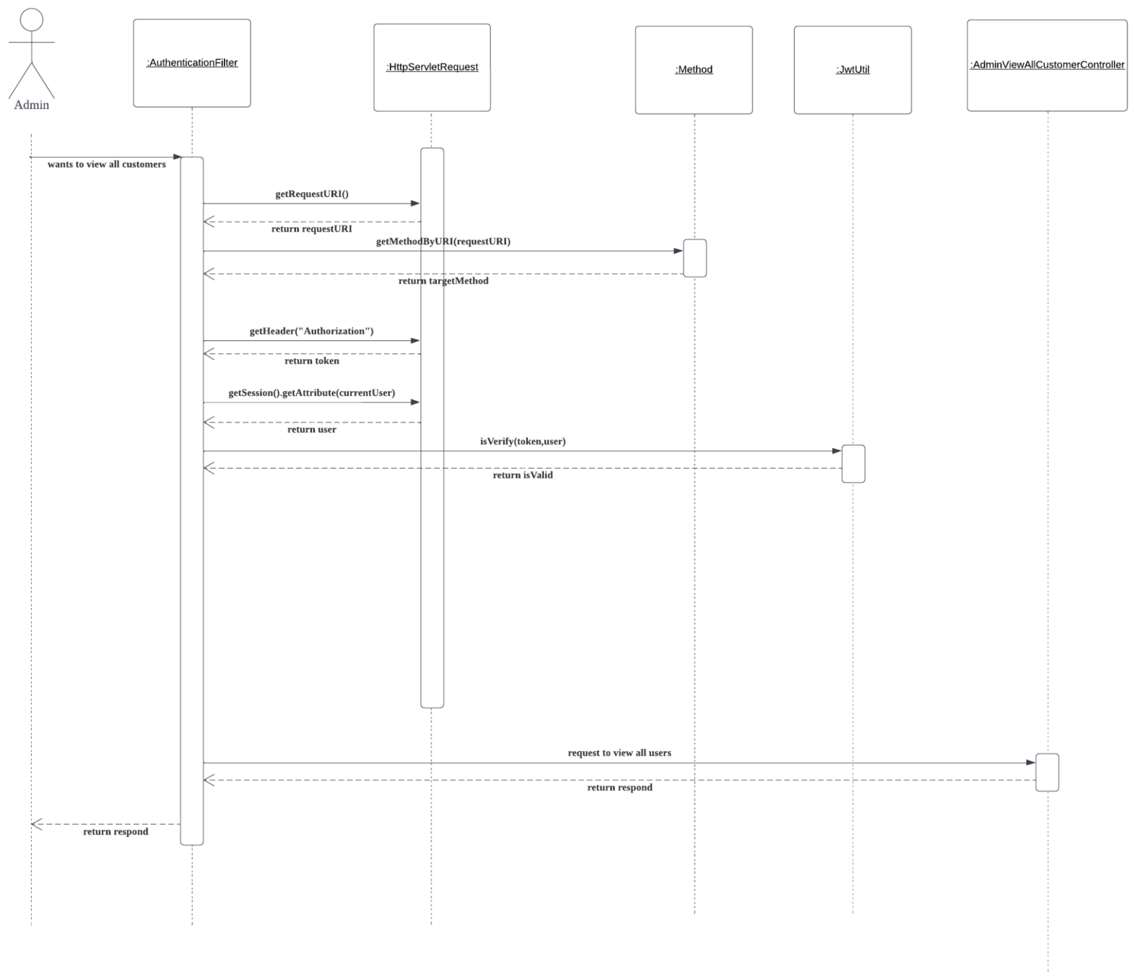
2.9.1.2 sequence diagram

Here is an example of the user logging into the system. Then the user gets the token to take actions.

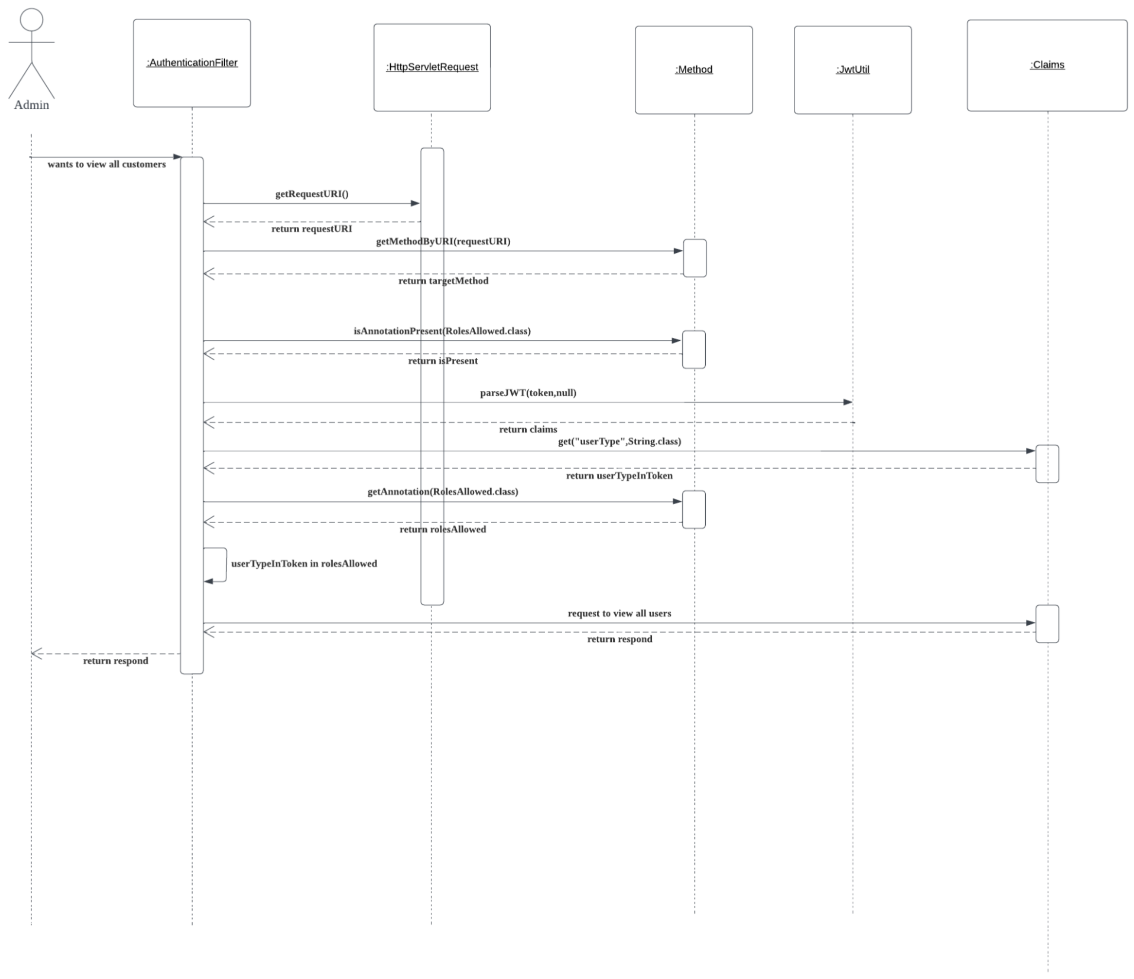


Here is an example of the system verifies the token of the admin and lets the admin view all the customers:

Authentication: The admin must authenticate with the token so that he can request to view all the customers.



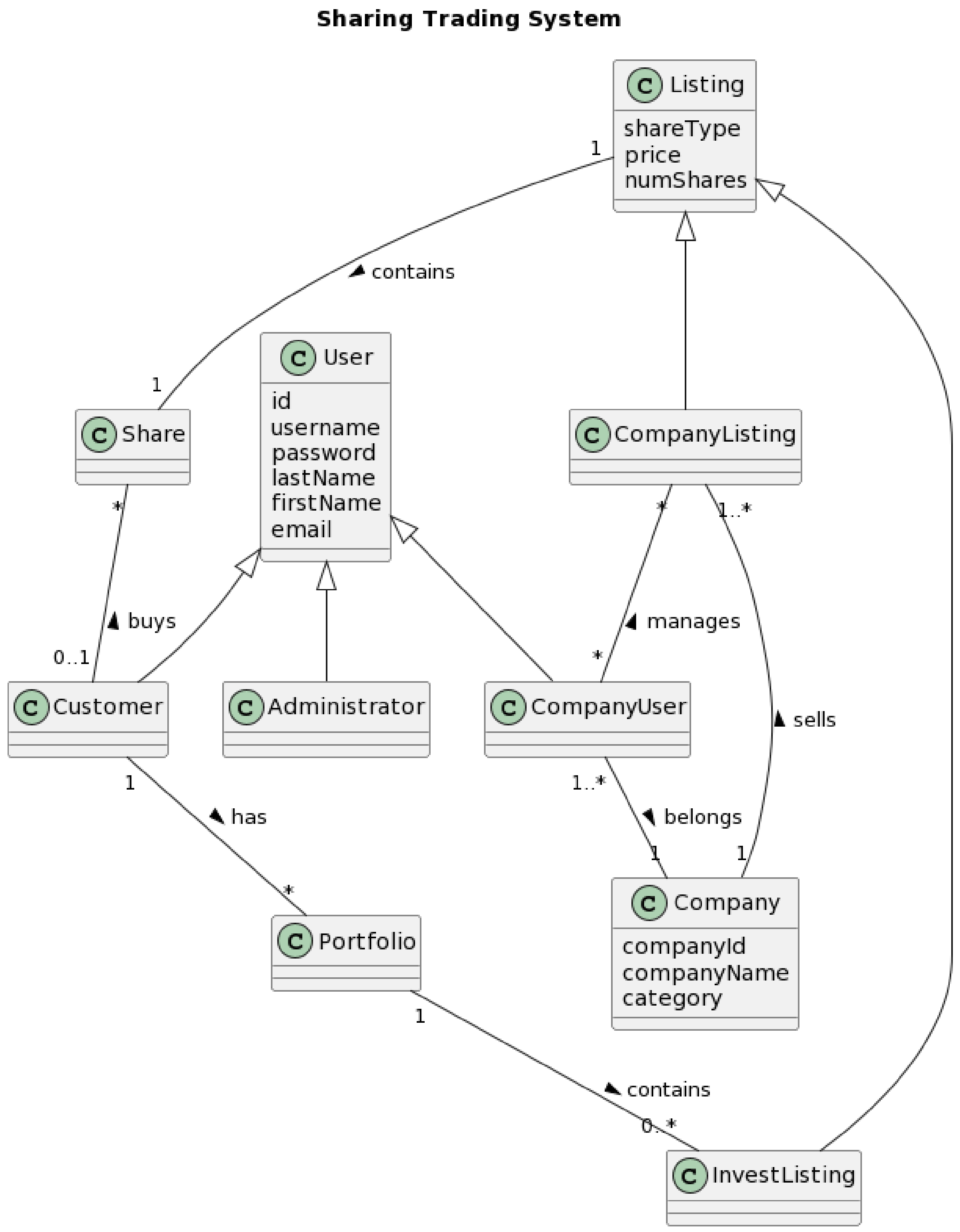
Authorization: The admin must be the role in the ‘rolesAllowed’ list. Then the request can be sent successfully.



2.10 Domain Model

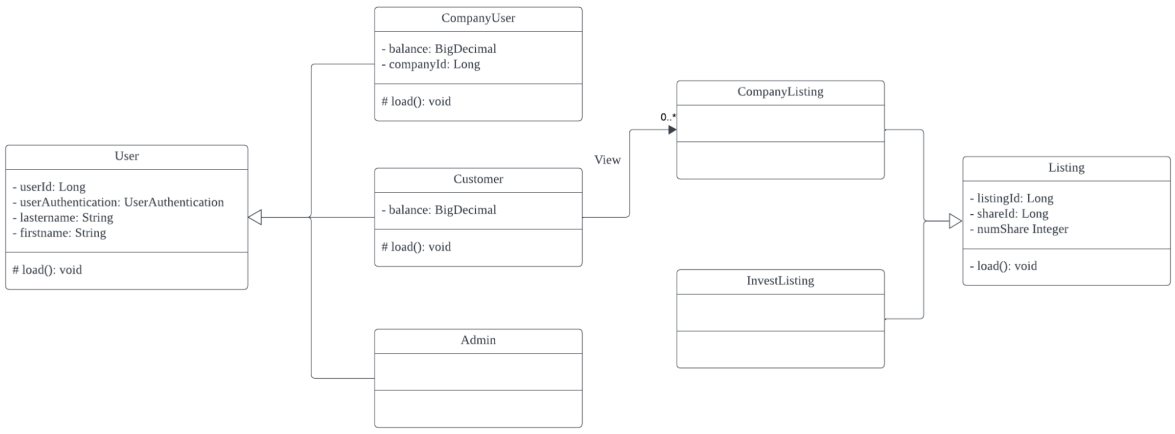
Pattern	Reason
Domain Model	Justification: The domain model pattern is used to solve complex business logic. It is object-oriented as each domain only deals with the parts related to it. As a result, it is also easy for us to add new domains. Adding more behavior only needs to add more classes or objects to the domain model. Comparison: The reason why we do not use the transaction script pattern is there are many duplicated behaviors occurring in the system. The duplication problem is one of the side effects of the transaction script. In addition, when the business logic becomes more complex, the complexity of the domain layer increases exponentially.

2.10.1 Implementation



2.10.1.1 class diagram

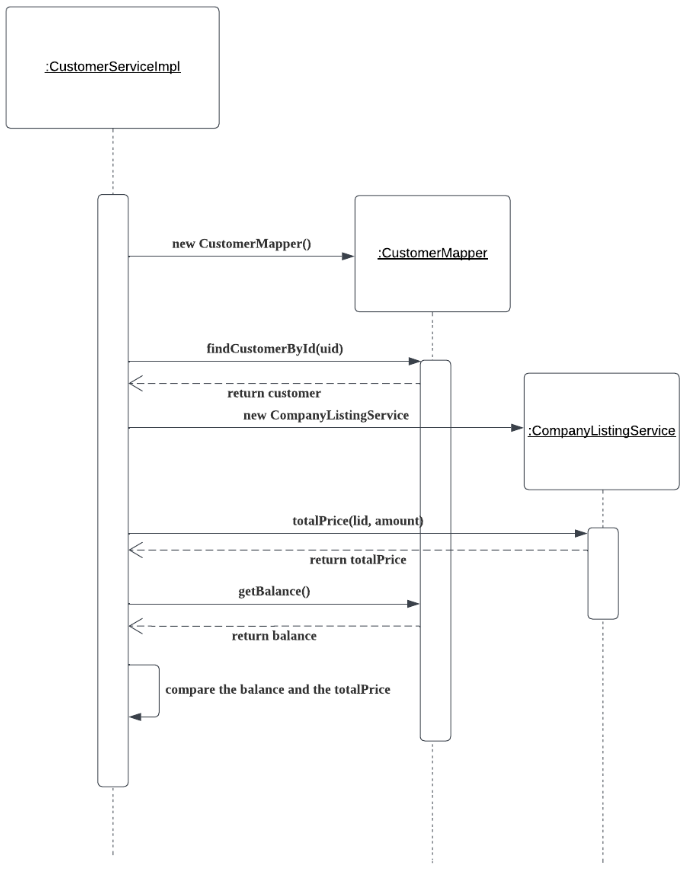
The class diagram of the action “the customer buys the shares’ can demonstrate the domain model pattern.



2.10.1.2 sequence diagram

Here is the sequence diagram for the customer to compare his own balance with the market total price.

The customer mapper deals with finding the specific customer to check his balance. The companyListing service is used to check the current total price of the share. Each domain deals with its own business.


:CustomerServiceImpl
:CustomerMapper
:CompanyListingService
new CustomerMapper()
findCustomerById(uid)
return customer
new CompanyListingService
totalPrice(lid, amount)
return totalPrice
getBalance()
return balance
compare the balance and the totalPrice