Student Name: Linyuan Zhao
Haoyang Cui
Student Number:947588

Introduction

The online movie ticket booking system is a web-based enterprise system that allows customers to book movie tickets and cinema manager to do order management. The system stores lots of information, including customer order history, movie detail information and movie schedule information. The system allows customers to buy movie tickets, view movie information and order history, and it also allows a cinema manager to do movie schedule management and customer order management.

Roles and current features

Feature A: Movie management

Role	Scenario	Description
Manger	Add movie	This service allows cinema managers to add new movies into the system
	Delete movie	This service allows cinema managers to delete existed movies from the system.
	Update movie	This service allows cinema managers to update movies' detailed information or schedule.
	View movie information	This service allows cinema managers to view existed movies from the system.
Customer	View movie information	This service allows customers to view existed movies from the system.

High Level Architecture

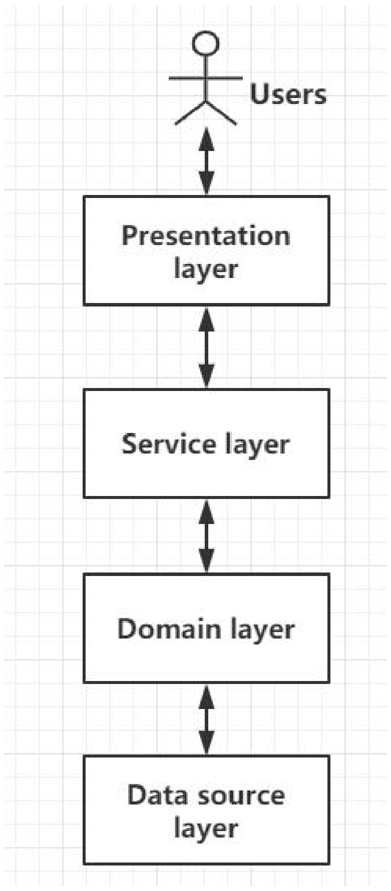

Figure 1. high level architecture

The system with a four-tier layered architecture is shown above, which are presentation layer, service layer, domain layer and data source layer from up to down. Individual layer does not have any overlaps with others but prepare interfaces to communicate, increasing the reusability and scalability.

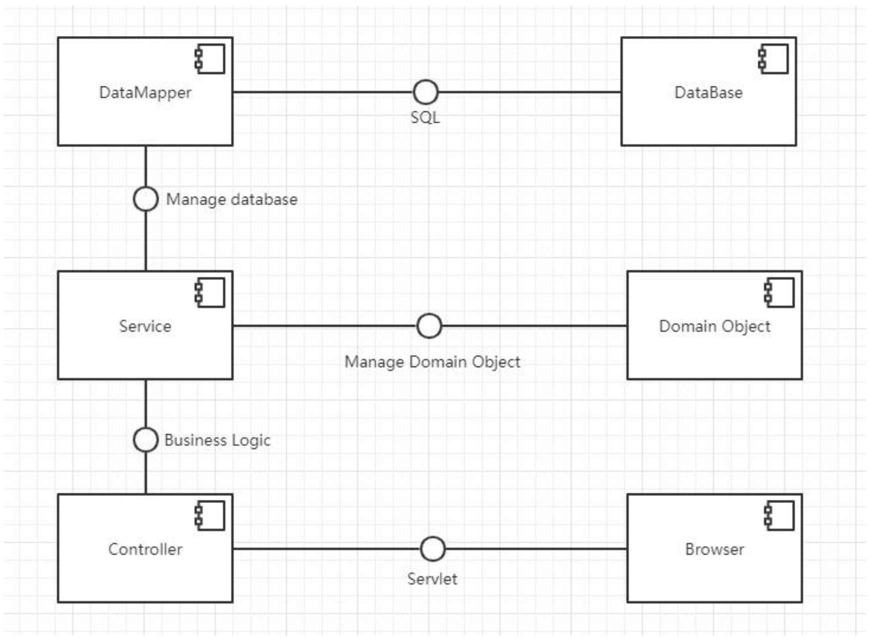

Figure2. Component diagram

The browser, as the user interface, get requests from users and pass it to the controller through servlet. Controller send requests to appropriate command to deal with business logic in service layer. Service component converts results from database to domain object by the interface of Domain Object component. Datamapper component manipulates database through SQL statement, and it also provides an interface to let service layer manage the insert/deleted/update/find operations in the database

presentation layer

➢ Front controller pattern

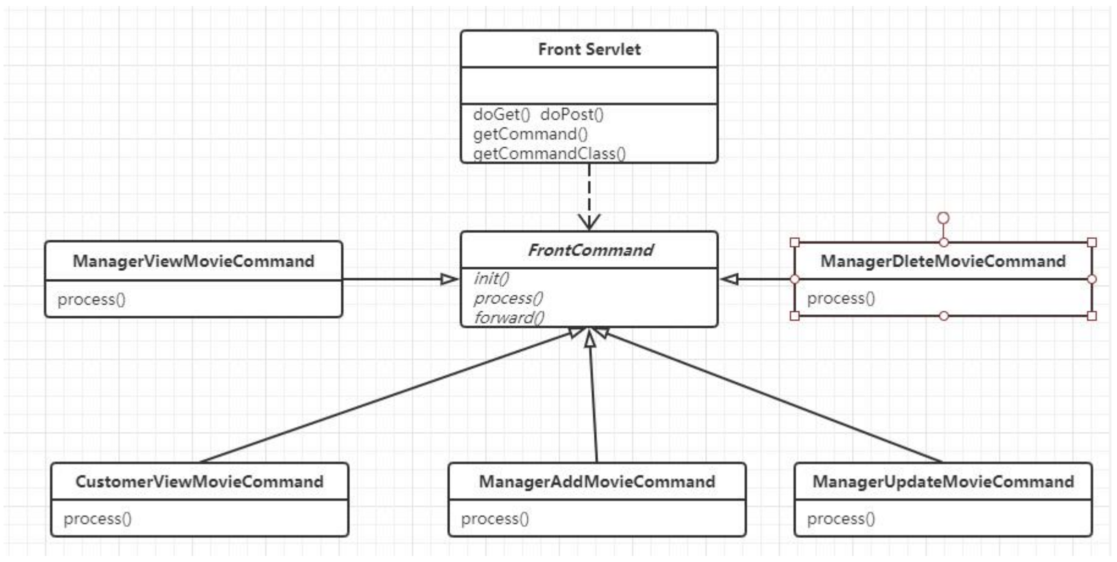

Figure3. Structure of the Front Controller design pattern.

(1) a handler called frontServlet, whose responsibilities are receiving the request and passing to different command classes, for example: view movie command and delete movie command.

(2) an abstract command class and command classes that extend it. Each command provides their own process by overriding the process method.

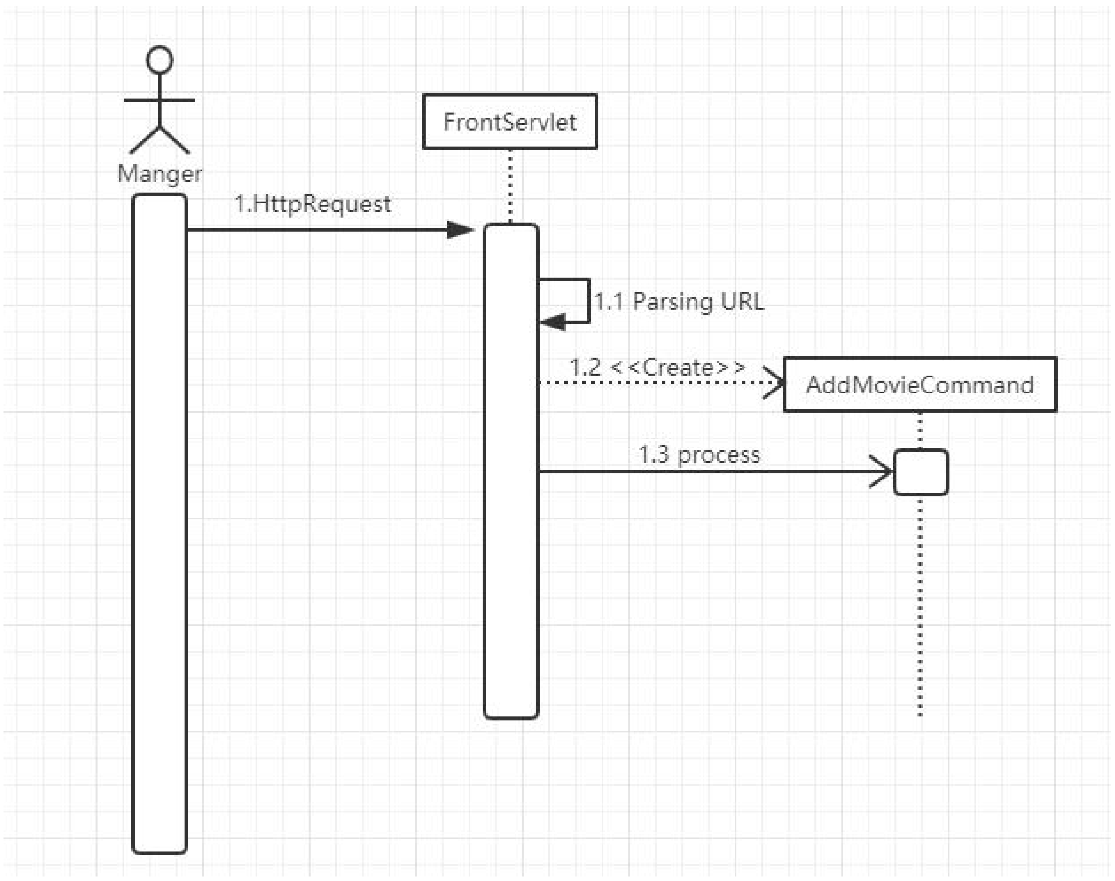

Figure4. Behaviour of the Front Controller design pattern.

The figure above explains the behaviors of front controller design pattern. When manager request to add a new movie, the processes is as below:

1. The http request will be sent to frontservlet.
2. Front servlet will examine the URL and find corresponding command.
3. An instance of addMovieCommand class will be created and it will process this request.

➢ View Pattern

The pattern used for handling the view in our project currently is template view. In this phase, we are focusing on business logic and serializing all components. As a result, we have to sacrifice maintainability and testability. At the next sprint, we are going to improve this part to transform view.

Service Layer

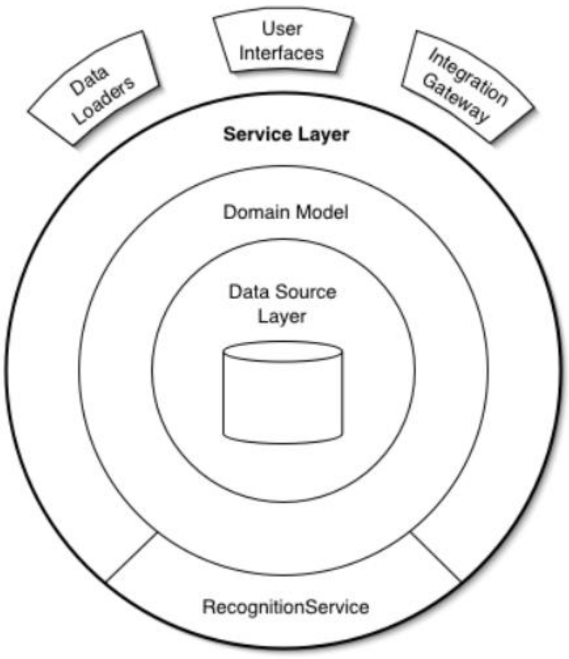

Figure5. Structure of service layer

A service layer defines a set of operations that can be accessed directly by the interfacing clients. When the requests of user has been parsed and sent to a particular command class to process, service layer will take over the business logic. After that, the result will be transferred to data source layer through interface.

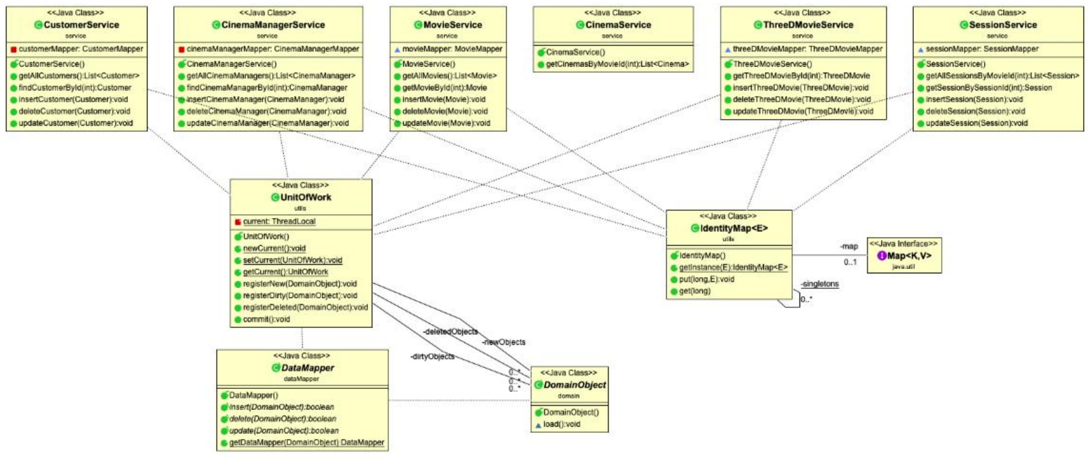
<<Java Class>>
CustomerService
service
customerMapper: CustomerMapper
CustomerService()
getAllCustomers():List<Customer>
findCustomerById(int):Customer
insertCustomer(Customer):void
deleteCustomer(Customer):void
updateCustomer(Customer):void
<<Java Class>>
CinemaManagerService
service
cinemaManagerMapper: CinemaManagerMapper
CinemaManagerService()
getAllCinemaManagers():List<CinemaManager>
findCinemaManagerById(int):CinemaManager
insertCinemaManager(CinemaManager):void
deleteCinemaManager(CinemaManager):void
updateCinemaManager(CinemaManager):void
<<Java Class>>
MovieService
service
movieMapper: MovieMapper
MovieService()
getAllMovies():List<Movie>
getMovieById(int):Movie
insertMovie(Movie):void
deleteMovie(Movie):void
updateMovie(Movie):void
<<Java Class>>
CinemaService
service
CinemaService()
getCinemasByMovieId(int):List<Cinema>
<<Java Class>>
ThreeDMovieService
service
threeDMovieMapper: ThreeDMovieMapper
ThreeDMovieService()
getThreeDMovieById(int):ThreeDMovie
insertThreeDMovie(ThreeDMovie):void
deleteThreeDMovie(ThreeDMovie):void
updateThreeDMovie(ThreeDMovie):void
<<Java Class>>
SessionService
service
sessionMapper: SessionMapper
SessionService()
getAllSessionsByMovieId(int):List<Session>
getSessionBySessionId(int):Session
insertSession(Session):void
deleteSession(Session):void
updateSession(Session):void
<<Java Class>>
UnitOfWork
utils
current: ThreadLocal
UnitOfWork()
newCurrent():void
setCurrent(UnitOfWork):void
getCurrent():UnitOfWork
registerNew(DomainObject):void
registerDirty(DomainObject):void
registerDeleted(DomainObject):void
commit():void
<<Java Class>>
IdentityMap<E>
utils
IdentityMap()
getInstance(E):IdentityMap<E>
put(long,E):void
get(long)
-map
0..1
<<Java Interface>>
Map<K,V>
java.util
-singletons
0..*
-deletedObjects
-newObjects
-dirtyObjects
0..*
0..*
0..*
<<Java Class>>
DataMapper
dataMapper
DataMapper()
insert(DomainObject):boolean
delete(DomainObject):boolean
update(DomainObject):boolean
getDataMapper(DomainObject):DataMapper
<<Java Class>>
DomainObject
domain
DomainObject()
load():void

Figure6. Class diagram of service layer

As shown in class diagram of service layer, the domain logic is divided into domain logic and workflow logic.

1. SessionService is responsible for handling creating, updating, and deleting a schedule.
2. MovieService is responsible for the operations about movies, such as creating a movie, view all the movies, edit the basic information of a movie, delete a movie.
3. CustomerService is responsible for the operations about customers

4. CinemaManagerService is similar with customer service
5. CinemaService is responsible for find cinemas by movie id
6. 3DMovieService is responsible for the operations about 3D movies

Unit of work and identity map
UnitOfWork class and identity map class are used as interfaces to link service classes with datamapper. The unit of work aims at reducing the amount of data that is written back to database. The identity map prevents the same data from being loaded into more than one object.

Domain Class E-R diagram:

➢ Domain model pattern

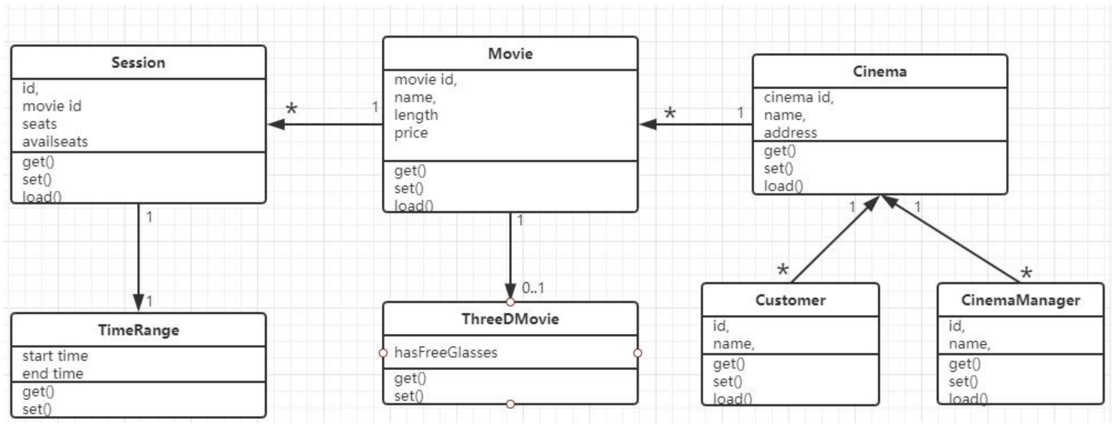

Figure7. Domain class diagram

The domain class ER diagram above demonstrates our design for domain model:
1. Cinema: contains the information of a cinema(name, address, cinema_id).
2. Movie: contain basic information of a movie(movie_Id, name, price and length).
3. 3DMovie: contains extra information of a 3D movie(if it has free glasses).
4. Customer: contain basic information of a customer(first name, last name, username, password).
5. Cinema manager: similar with customer class.
6. Session: contains session_Id, seat_id available seat and movie_id.
7. TimeRange: contains startTime and endTime.
8. CinemaMovie: contains movie id and cinema id.

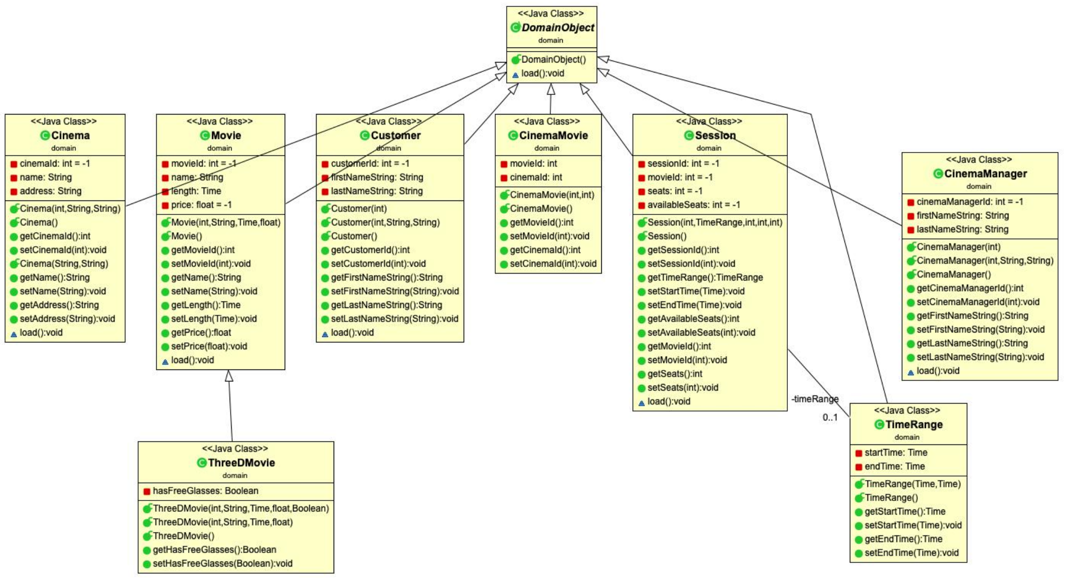

Figure 8. Domain class diagram

- Lazy load pattern

In order to reduce the amount of date read from data, we uses ghost to implement this pattern. For example, when a manager is going to delete a movie by id, he only needs to input the movie id. A movie object is created but other fields(like movie name, price) are null. When load function is called, the entire movie object will be initialized at the same time.

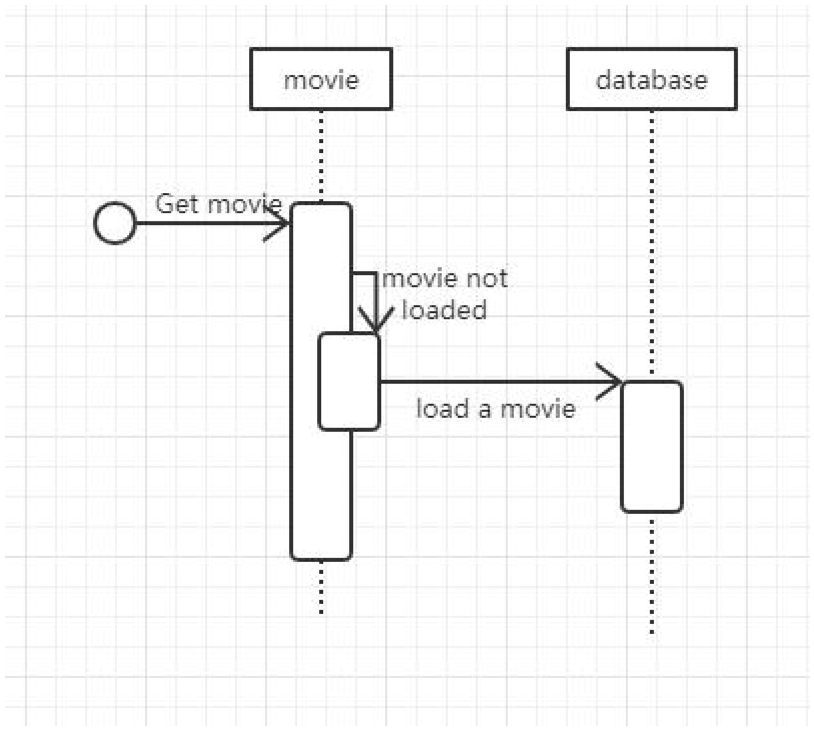

Figure9. The behavior of lazy load pattern

Data Source Layer

- Data mapper pattern

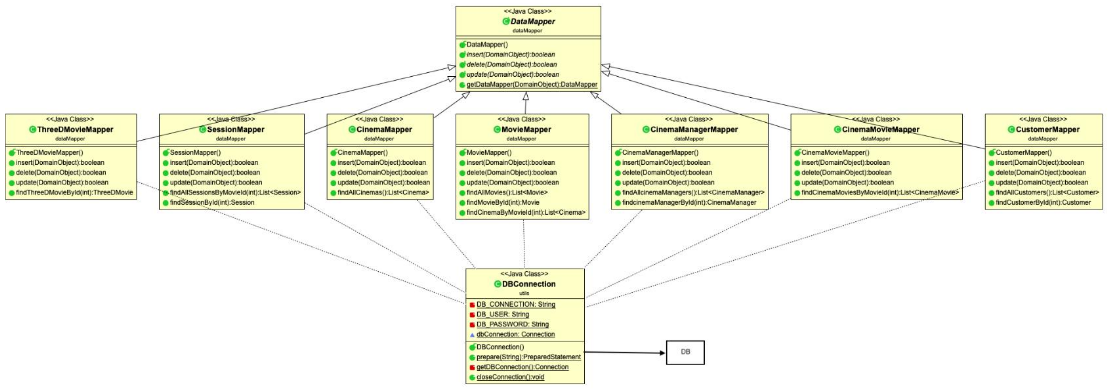
<<Java Class>>
DataMapper
dataMapper
DataMapper()
insert(DomainObject):boolean
delete(DomainObject):boolean
update(DomainObject):boolean
getDataMapper(DomainObject):DataMapper
<<Java Class>>
ThreeDMovieMapper
dataMapper
ThreeDMovieMapper()
insert(DomainObject):boolean
delete(DomainObject):boolean
update(DomainObject):boolean
findThreeDMovieById(int):ThreeDMovie
<<Java Class>>
SessionMapper
dataMapper
SessionMapper()
insert(DomainObject):boolean
delete(DomainObject):boolean
update(DomainObject):boolean
findAllSessionsByMovieId(int):List<Session>
findSessionById(int):Session
<<Java Class>>
CinemaMapper
dataMapper
CinemaMapper()
insert(DomainObject):boolean
delete(DomainObject):boolean
update(DomainObject):boolean
findAllCinemas():List<Cinema>
<<Java Class>>
MovieMapper
dataMapper
MovieMapper()
insert(DomainObject):boolean
delete(DomainObject):boolean
update(DomainObject):boolean
findAllMovies():List<Movie>
findMovieById(int):Movie
findCinemaByMovieId(int):List<Cinema>
<<Java Class>>
CinemaManagerMapper
dataMapper
CinemaManagerMapper()
insert(DomainObject):boolean
delete(DomainObject):boolean
update(DomainObject):boolean
findAllcinemaManagers():List<CinemaManager>
findcinemaManagerById(int):CinemaManager
<<Java Class>>
CinemaMovieMapper
dataMapper
CinemaMovieMapper()
insert(DomainObject):boolean
delete(DomainObject):boolean
update(DomainObject):boolean
findCinemaMoviesByMovieId(int):List<CinemaMovie>
<<Java Class>>
CustomerMapper
dataMapper
CustomerMapper()
insert(DomainObject):boolean
delete(DomainObject):boolean
update(DomainObject):boolean
findAllCustomers():List<Customer>
findCustomerById(int):Customer
<<Java Class>>
DBConnection
utils
DB_CONNECTION: String
DB_USER: String
DB_PASSWORD: String
dbConnection: Connection
DBConnection()
prepare(String):PreparedStatement
getDBConnection():Connection
closeConnection():void
DB

Figure10. the class diagram of data mapper

This pattern is used as the architectural design in the data source layer. In consideration of maintainability and reusability, we selected domain model pattern in our system. Thus, we chose data mapper pattern for data source layer because it is compatible with domain model pattern. The six data mappers in the data source layer connects the database with the website. Since they all have insert/update/delete operations to database, they implement the interface from dataMapper

➢ Unit of work pattern

The unit of work pattern describes a way to keep track of which domain objects have changed (or new objects created), so that only those objects that have changed need to be updated in the database.

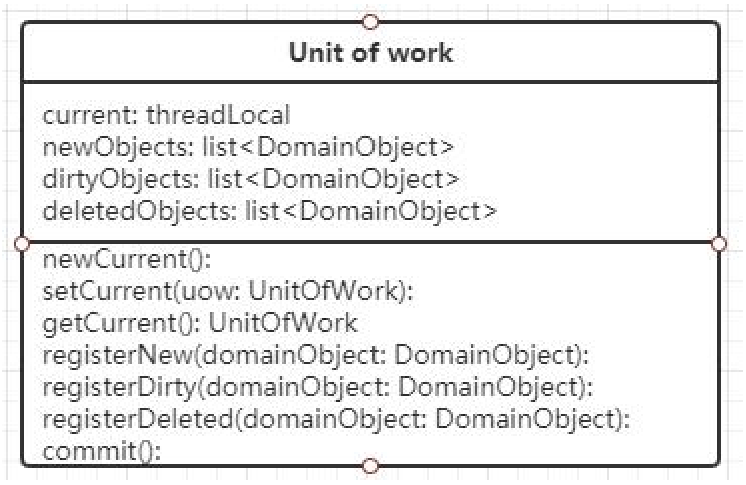
Unit of work
current: threadLocal
newObjects: list<DomainObject>
dirtyObjects: list<DomainObject>
deletedObjects: list<DomainObject>
newCurrent():
setCurrent(uow: UnitOfWork):
getCurrent(): UnitOfWork
registerNew(domainObject: DomainObject):
registerDirty(domainObject: DomainObject):
registerDeleted(domainObject: DomainObject):
commit():

Figure11. the class diagram of unit of work

The class unit of work contains:

1. An instance of threadLocal to prevent the same instance from accessing by other threads.
2. Lists of new(new objects that have been created), dirty(existing objects whose attributes have changed values), deleted(existing objects that need to be removed). Each object that has been changed must be exactly in one of these lists. After the commit operation, only objects in these lists will be changed

➢ Identity map pattern

We also use Identity map pattern to make sure every domain object is read from database only once.

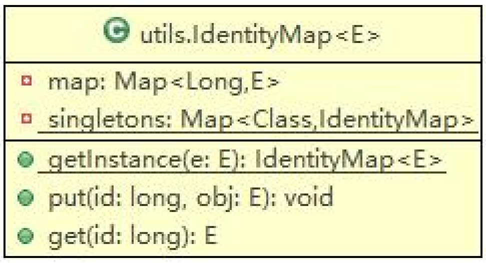
utils.IdentityMap<E>
map: Map<Long,E>
singletons: Map<Class,IdentityMap>
getInstance(e: E): IdentityMap<E>
put(id: long, obj: E): void
get(id: long): E

Figure12. Identity map class diagram

In database, each movie has an unique movie_id as primary key. Before a movie is read from the database, a lookup in hash map will be executed first. If an object already exists, the map return that instance, else a new instance will be created and added to hash map.

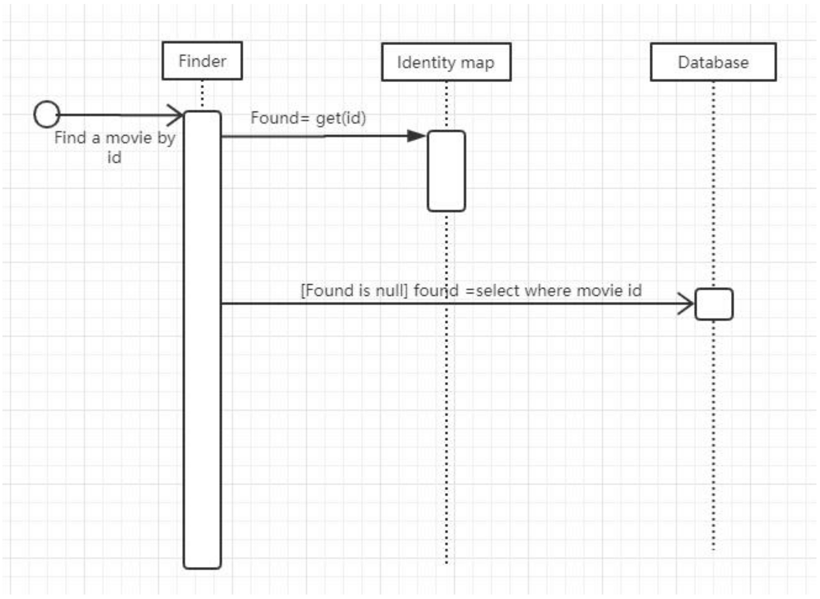

Figure13. Behavior of identity map

➢ Identity Field

Each object in the domain correspond to an unique row in the database, so we need to identify which is the corresponding row of an domain object when we read or write the information from or to the database. For example, in the movie table, the movie_id is unique. Therefore, we adopted identify field pattern and stored it as a primary key.

➢ Foreign Key Mapping

Some domain objects have one-to-many association with other domain objects, such as one movie can have multiple sessions. A session can not exist without a movie_id or it cannot have a movie_id that does not exist. As a result, in the design of session object, we store movie_id in the session table and reference it to movie_id in movie table.

➢ Association table mapping

In our system, movie and cinema has a many-to-many association. A cinema should have many movies and a movie could hit many cinemas. Mapping this using the association table mapping pattern will result in the table design shown below

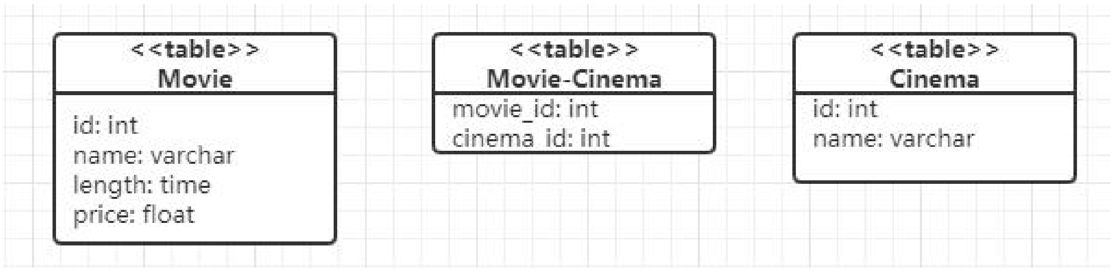

Figure14. association table mapping

Interaction diagram

1. View all movies

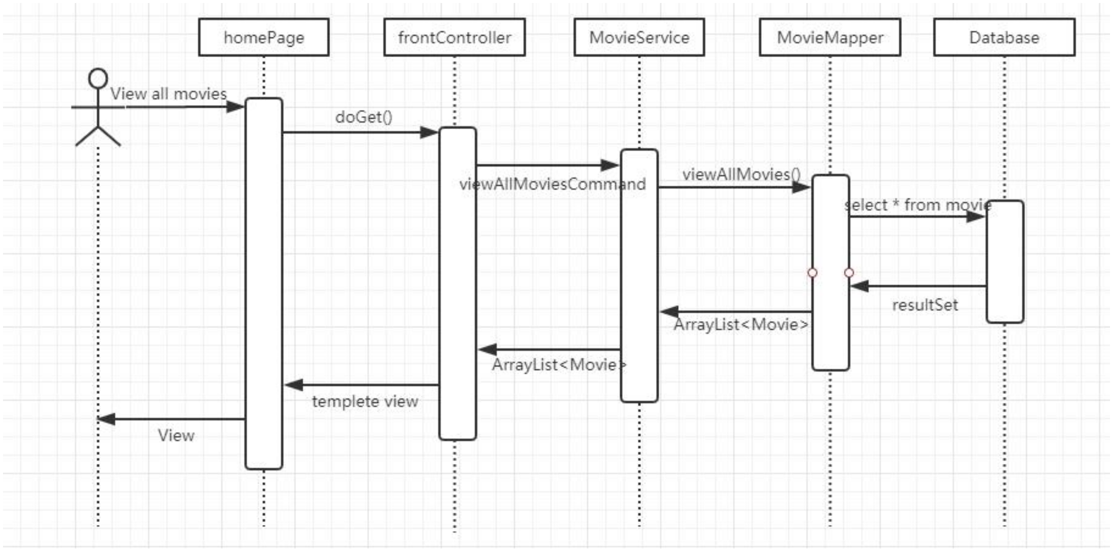

Figure15. interaction diagram of view all movies

2. Delete a movie

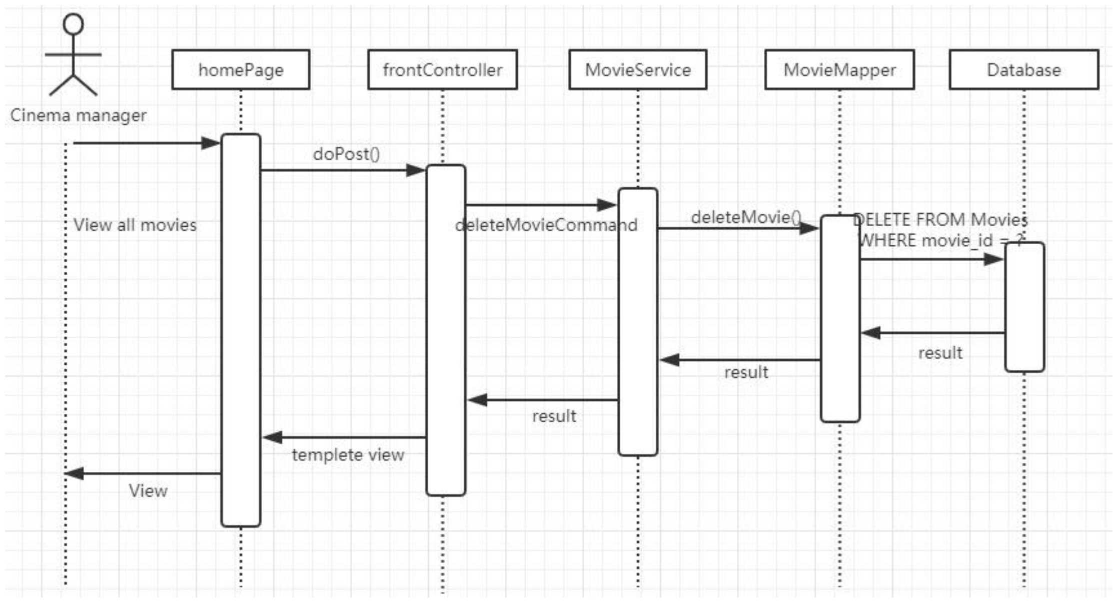

Figure16. interaction diagram of deleting a movie

3. Add a movie

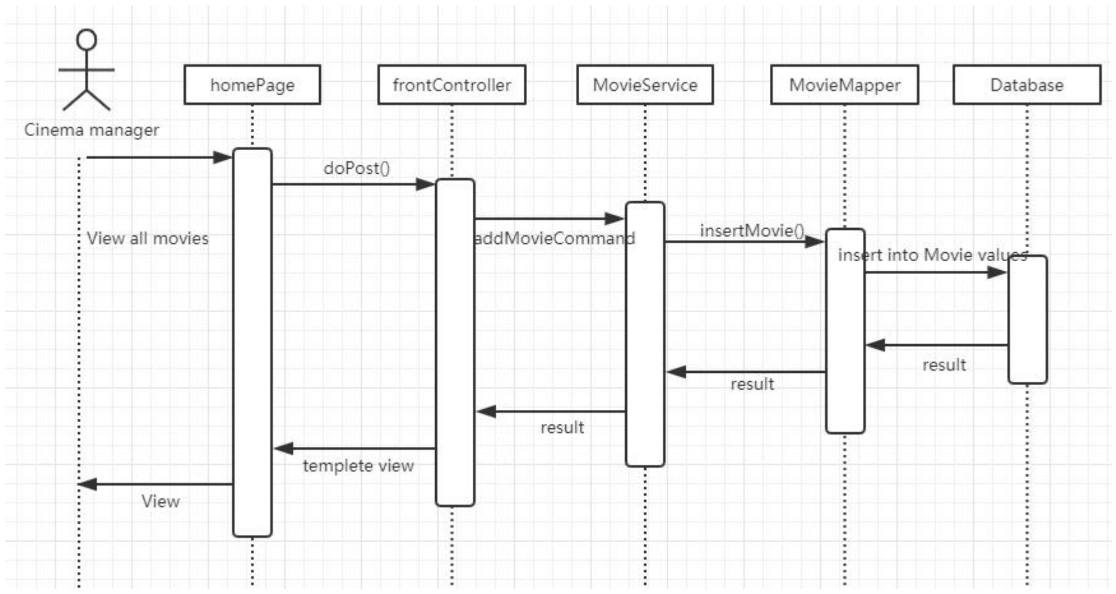

Figure17. interaction diagram of adding a movie

4. Update a movie

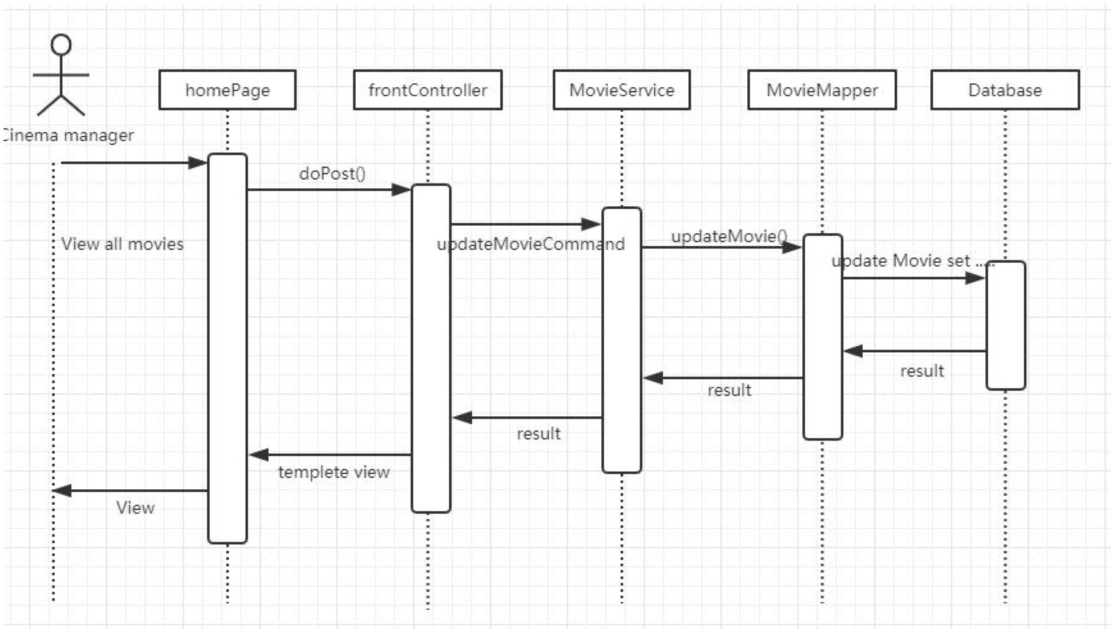

Figure18. interaction diagram of updating a movie

Code and depolyment

Github:
https://github.com/HaoyangCui0830/SWEN90007-Project-OnlineMovieTicketBookingSystem

Heroku: https://online-movie-ticket-booking.herokuapp.com/