**Movie Tickets Management System**

This movie management system mainly focuses on providing users a nice experience in ordering movies online, where they can browse movie lists, make movie orders, make payments, rate the movies in 0-5 stars, post comments about movies, and get recommendations based on the selected movie category. This C/S-based system also delivers a convenient and friendly interface for managers to load the existing movie information, update or insert new movies into the system.

In the final iteration, we mainly redesigned and rebuilt the system using microservices and multi-modal clients, with the new use case of recommending the best movies for movie audiences. We established the server side to handle multiple core functions such as microservices which can be connected with both desktop-based client and web-based client. In terms of the communication between client and service, and between the microservices, we adopted RESTful APIs, and stored simple data objects like user information and product entities as key-value pairs and hash-set values in cloud database Redis, while complicated data objects like reviews and order details are stored in document-oriented database MongoDB, to enable the system to meet the requirement, satisfy the user function needs, and optimize the overall performance.

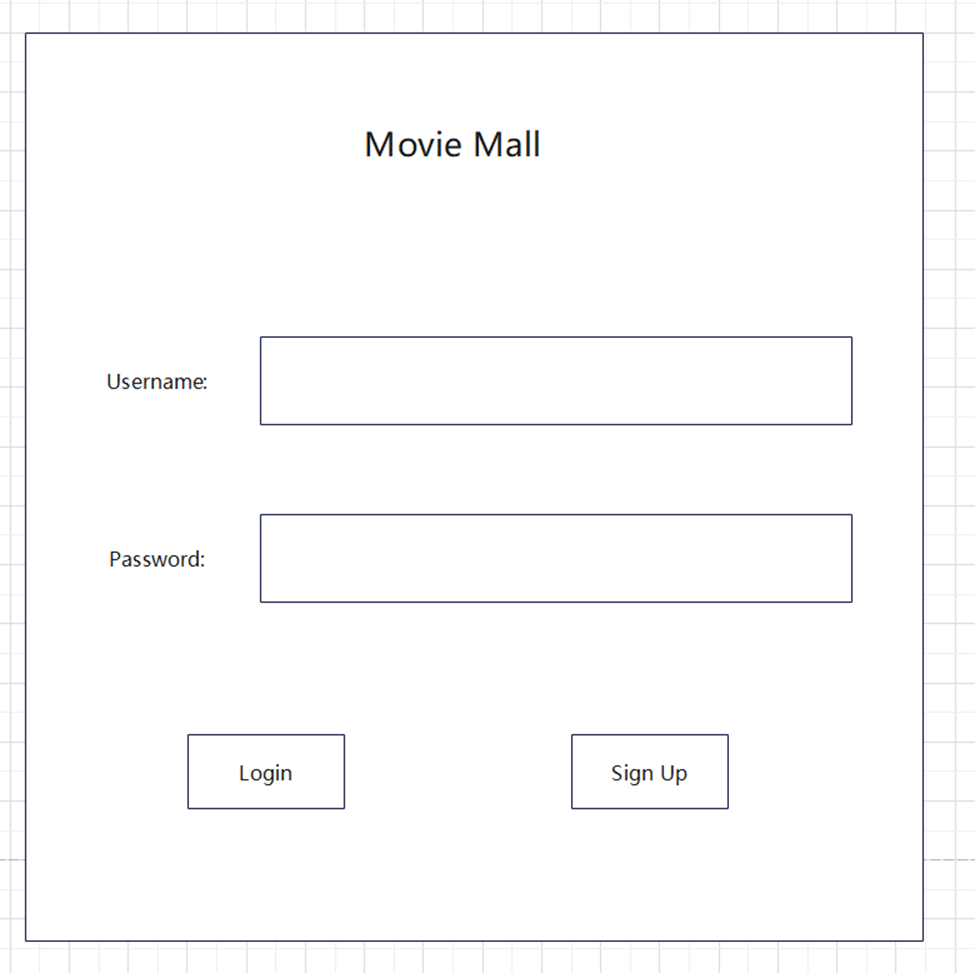
To be specific, we built on the base of iteration 2 and developed the web client of the system by adding a recommendation use case into the system. In total, this system handles 4 microservices in the function of user management, payment management, movie management, and order management. To exemplify the core principle in software engineering of low coupling and high cohesion, we totally build 4 databases with corresponding tables to execute respective operations of each microservice to enable a better user experience and complete implementation of the series of use cases.

For the final runnable system, we ran rigorous user acceptance tests, enhancing the reliability and scalability of the system, ensuring the integrity and robustness of operations and businesses. Meanwhile, the UI of the system is personalized and polished carefully to present a user-friendly interface, improving the system’s uniqueness together with the look-and-feel, enhancing user experience while fully implementing all the proposed use cases.

In the following chapters, we are going to specifically introduce the requirements (UI sketch with corresponding use cases), database design, the architectural design of clients and server, and the manual for installation about this system.

This system was developed as a course project of CSCE 606 at Texas A&M university.

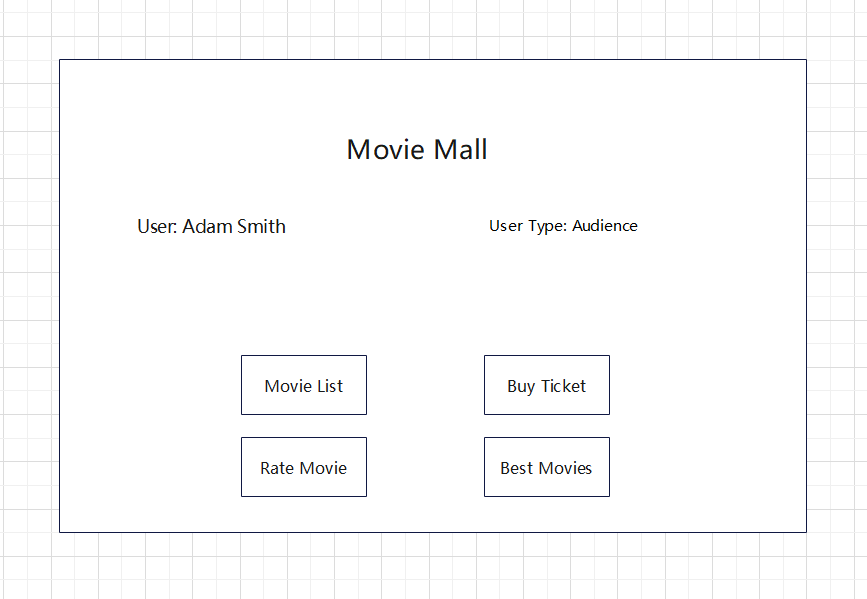
**1. Requirements (UI Sketch and Use Cases)**

1) Login and Sign Up

This is the sketch of our Login and Sign Up UI interface. Users can enter their username and password to login to the system, or select their username, password, displayname and usertype to sign up an account.

|  |  |  |
| --- | --- | --- |
| Use Case | Login and Sign Up | |
| Actors | Users | |
| Precondition | The Client is successfully connected to the Server | |
| Postcondition | Users log into the system and the system gets user information | |
| Main Path | Actor Action  1. Chooses to sign up an account  3. Specifies username, password, display name and selects the corresponding account type  7. Chooses to log into the system  8. Enters the correct username, password, and clicks log in button | System Response  2. Displays the Sign Up window  4. Client processes the entered information, validates data.  5. Client sends the data to server. Server gets the data.  6. Server makes sure no data conflicts in database, inserts the user information and creates new user entity in database  9. Client sends the data to server. Server gets the data.  10. Sever queries the user information in database, confirms the correctness  11. Displays the Main interface according to the account type. |
| Alternative Path (A1) | 4a. System fails to validate the entered information. System displays a window warning the user information is not valid! | |
| Alternative Path (A2) | 6a. System fails to insert user information and creates new user entity in database, or there lies data conflicts. System displays a window warning the user information is not correct! | |
| Alternative Path (A3) | 8a. User enters incorrect information  10a. System fails to confirm the correctness of user information, and displays a window warning the user information is not correct! | |

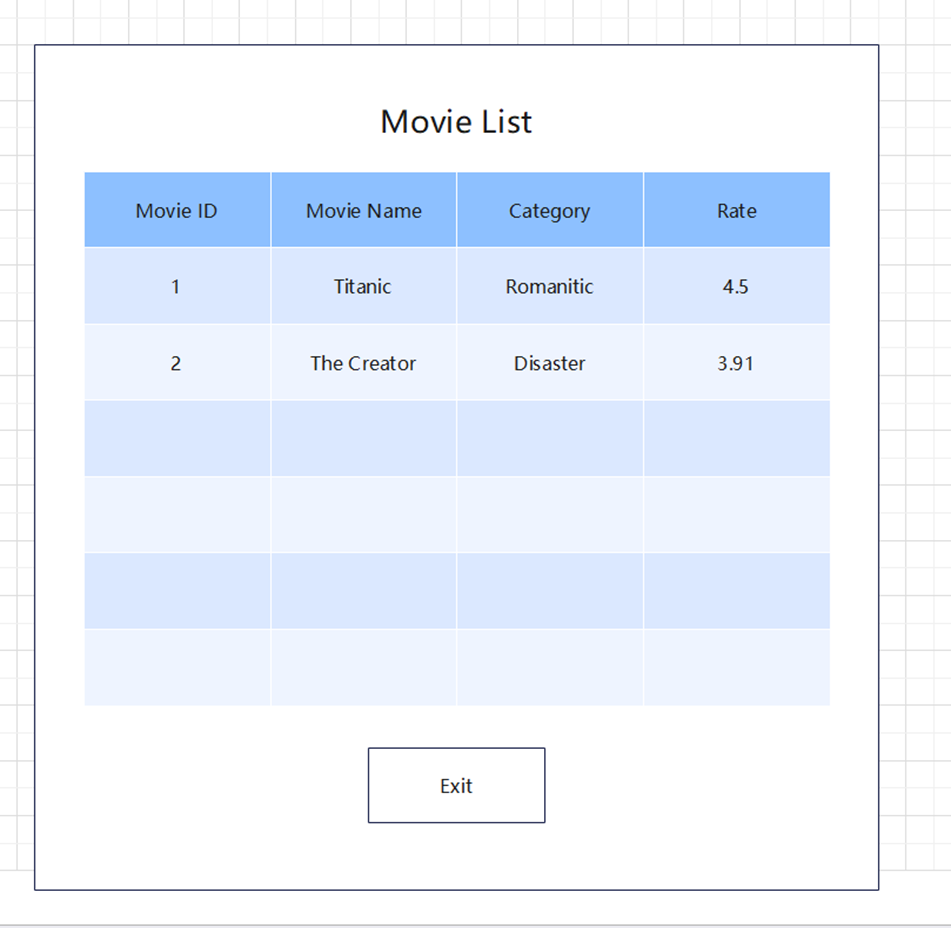
2) Main interface - Audience



This is the Main interface of system. Here we can find our current username. Click the buttons at the bottom of the interface, we are able to select system operations: browse the movie list, buy tickets, rate and write reviews for movies.

|  |  |  |
| --- | --- | --- |
| Use Case | Select the main system operations | |
| Actors | Users (Audience) | |
| Precondition | User successfully log into the system as an audience type | |
| Postcondition | System executes corresponding operations per user’s actions | |
| Main Path | Actor Action  1. Selects to browse movie list  3. Selects to Buy Movie Ticket  5. Selects to Rate & Review  7. Selects to recommend movie | System Response  2. Displays the Movie List  4. Displays the Shopping Cart  6. Displays the Rating window  8. Displays the recommend window |

3) Movie List



This is the Movie List interface. Here we can find movies available in our cinema, with the movie ID, category and rate for each movie. By clicking the Exit button, we can close the window.

|  |  |  |
| --- | --- | --- |
| Use Case | Browse the Movie List | |
| Actors | Users (Audience or Manager) | |
| Precondition | User chooses to browse movie list in the main interface | |
| Postcondition | System displays the current movie information list | |
| Main Path | Actor Action | System Response  1. Client sends the request of getting movie list to server, server gets the request.  2. Server processes the request and makes queries in database.  3. Server responses the movie list information to the client.  4. Client displays the movie list. |
| Alternative Path (A1) | 5. user clicks the Exit button, system closes the window and return to the main interface. | |

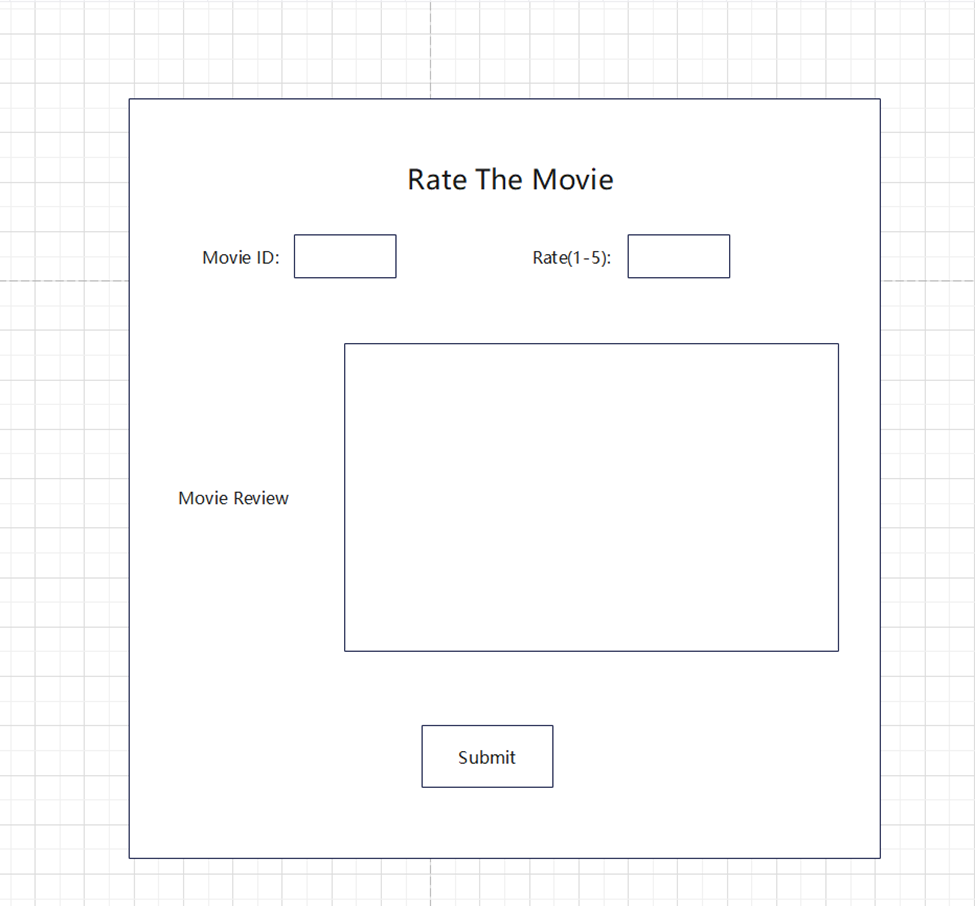
4) Shopping Cart



This is our Shopping Cart interface. By clicking the Add new tickets button, we can add movie tickets to our shopping cart. By clicking the Finish and Pay button, we can enter our credit card number to buy and make orders.

|  |  |  |
| --- | --- | --- |
| Use Case | Add movie to cart and make an order | |
| Actors | Users (Audience) | |
| Precondition | User chooses to buy tickets in the main interface | |
| Postcondition | System saves the order information and generates receipt | |
| Main Path | Actor Action  1. Clicks the Add new tickets button  3. Enters the movieID, quantity  7. Clicks the Finish and Pay button  9. Enters the payment information (credit card, addresses) and clicks the Pay button | System Response  2. System displays the window allowing users to enter the ticket information they want to buy  4. Client sends the entered information to server, server gets the information  5. Server queries the movieID in database, checks if there exist enough tickets, and responses the movie information to client.  6. Client displays the movie information, purchase quantity, cost and total cost.  8. Client displays a Payment Page asking user about the payment information (credit card, addresses)  10. Client generates a receipt, sending the receipt content to server and saving in database |
| Alternative Path (A1) | 3a. User enters invalid movieID or quantity, system will pop up a window warning users the entered information is invalid! | |
| Alternative Path (A2) | 5a. Server fails to query the movieID in database or the tickets are not enough, system will pop up a window warning users the movie information is invalid! | |
| Alternative Path (A3) | 9a. User enters the invalid payment information (credit card, addresses), such as the credit card number is too short, system will pop up a window warning users the information is invalid! | |

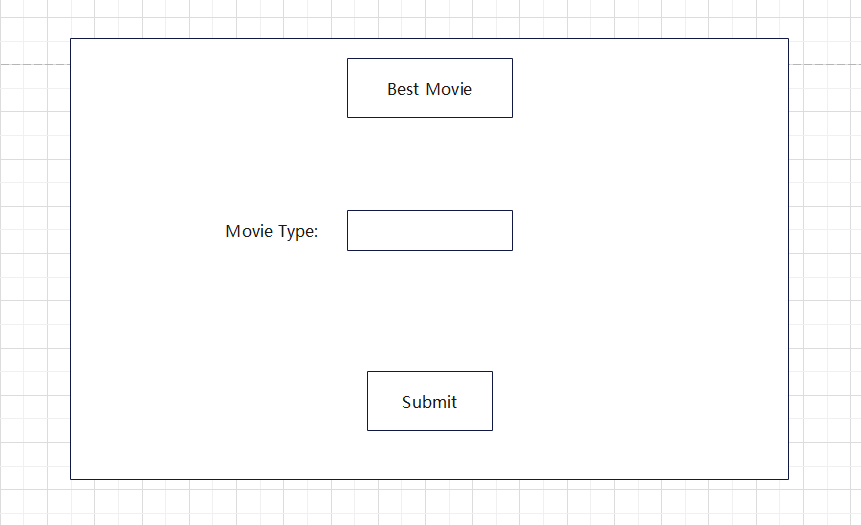
5) Rate the Movie



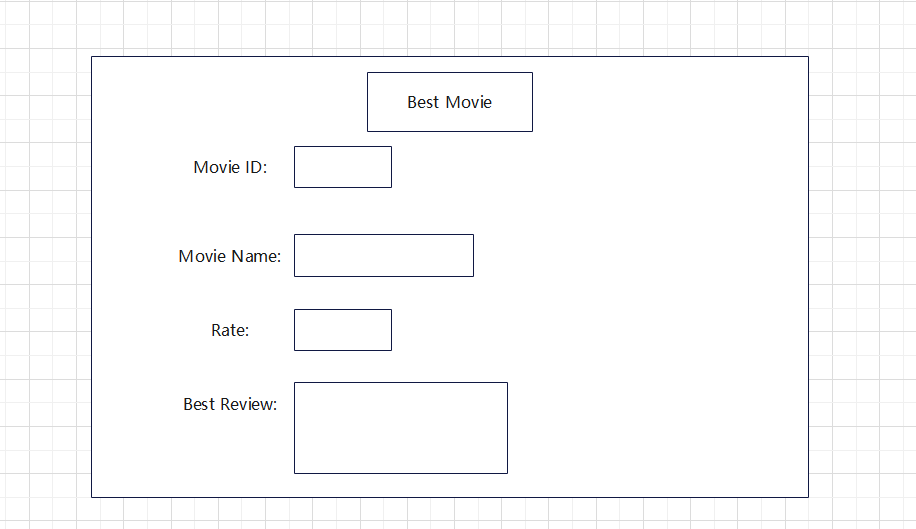
This is our Rate the Movie interface. Users can enter the Movie ID and the rate from 1 to 5 they want to rank for this movie and write watching reviews for the movie. By clicking the Submit button, we can finish the rate.

|  |  |  |
| --- | --- | --- |
| Use Case | Rate and review a movie | |
| Actors | Users (Audience) | |
| Precondition | User chooses to rate and review in the main interface | |
| Postcondition | System saves the review information in database | |
| Main Path | Actor Action  1. Enters the movieID, rate and review content  2. Clicks the Submit button | System Response  3. Client validates the entered information.  4. Client sends the entered information to server, server gets the data information  5. Server computes the weighted rate of a movie, stores the information in database, and responses the status to the client. |
| Alternative Path (A1) | 1a. User enters invalid movieID, rate or review content  3a. Client fails to validate the entered information, system will pop up a window warning users the entered information is invalid! | |

6) Recommend the best movie

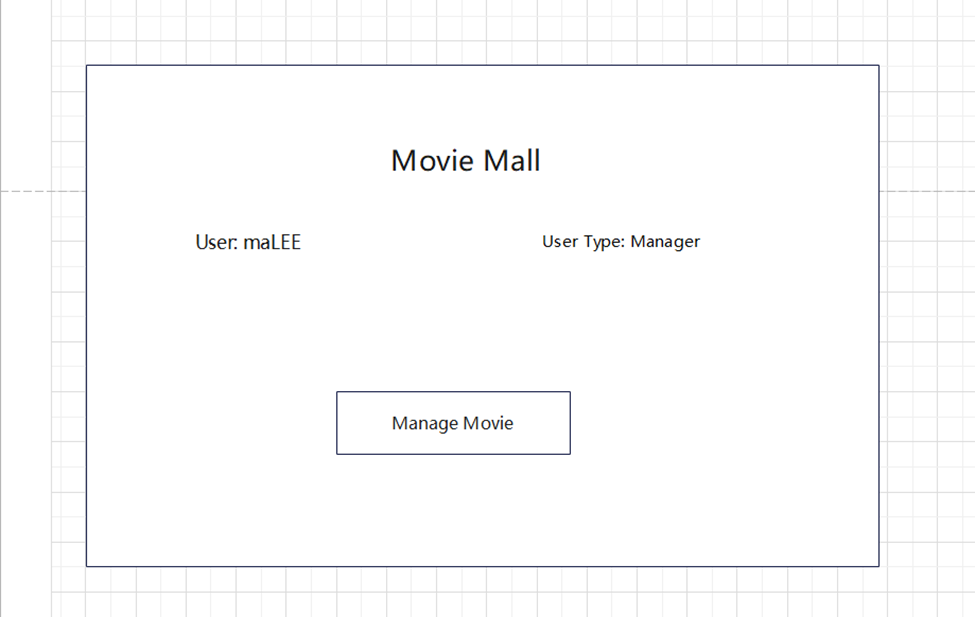


This is the sketch of our Recommend movie interface. Users can enter the Movie Type they want. By clicking the Submit button, users can get the movie information in this type with the highest rate and stored movie review regarding this most popular movie.



|  |  |  |
| --- | --- | --- |
| Use Case | Recommend Movie (load / save) | |
| Actors | Users (Audience) | |
| Precondition | User chooses to Best Movies in the main interface and type in the movie type they like | |
| Postcondition | System displays the movie information with the highest rate | |
| Main Path | Actor Action  1.Clicks the Best Movies Button  3. Enters the movie type and Clicks the Submit button | System Response  2. Displays the Recommend Movie window and interface  4. Displays the Recommend Movie information |
| Alternative Path (A1) | 3a. User enters invalid movie type,  4a. Client fails to validate the entered information, system will pop up a window warning users the entered information is invalid! | |
| Alternative Path (A2) | 3b. User enters a valid movie type, but the movie with this type does not exist!  4b. Server fails to query the movie information in the database, the system will pop up a window warning users that the movie does not exist! | |

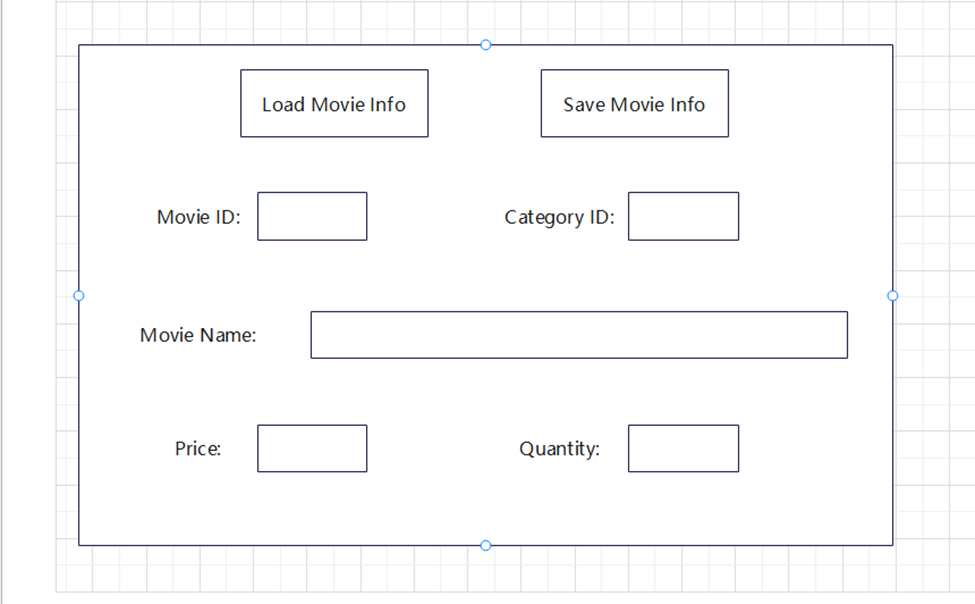
7) Main Interface - Manager



When the Manager logs into the system, the Main Interface is different from the audience’s. By clicking the Manage Movie button, the Manager can Manage the movie information by loading existing movie information or saving new ones.

|  |  |  |
| --- | --- | --- |
| Use Case | Select the system operations | |
| Actors | Users (Manager) | |
| Precondition | User successfully log into the system as a manager type | |
| Postcondition | System executes corresponding operations per user’s actions | |
| Main Path | Actor Action  1. Selects to Manage Movie | System Response  2. Displays the Manage Movie window and interface |

8) Manage Movie



This is our Manage Movie interface sketch. By entering the Movie ID and clicking the Load Movie Info Button, Manager can load the movie information from database. Besides, the manager can add a new movie or modify the existing movie information and click the Save Movie Info Button to save the movie into database.

|  |  |  |
| --- | --- | --- |
| Use Case | Manage movie information (load / save) | |
| Actors | Users (Manager) | |
| Precondition | User chooses to manage movie information in the main interface | |
| Postcondition | System saves the movie information changes in database | |
| Main Path | Actor Action  1. Enters the movieID to load  2. Clicks the Load Movie Info button  7. Enters the movie information to save or update  8. Clicks the Save Movie Info button | System Response  3. Client validates the entered information.  4. Client sends the entered information to server, server gets the data information  5. Server queries the movie information in database, and responses to client.  6. Client displays the movie information (Movie ID, Category ID, Movie name, Price, Quantity)  9. Client validates the entered information.  10. Client sends the entered information to server, server gets the data information  11. Server checks the movie information against database, make changes in database, and responses to client.  12. Client displays a window informing user of success action |
| Alternative Path (A1) | 1a. User enters invalid movieID,  2a. Client fails to validate the entered information, system will pop up a window warning users the entered information is invalid! | |
| Alternative Path (A2) | 1b. User enters valid movieID, but the movie does not exist!  5a. Server fails to query the movie information in database, and the system will pop up a window warning users the movie does not exist! | |
| Alternative Path (A3) | 7a. User enters invalid movie information  9a. Client fails to validate the entered information, system will pop up a window warning users the entered information is invalid! | |
| Alternative Path (A3) | 7b. User enters the movieID of existing movies  11a. Server updates existing movie information in database | |
| Alternative Path (A4) | 7c. User enters the movieID of non-existing movies  11a. Server inserts the new movie information in database | |

**2. Database Design**

**2.1 Entity and Relationship**

In this system, we mainly deal with 9 data entities to handle the use cases, including: User, Ticket, Movie, Order, Payment (Credit Card, Address), Movie Category, Movie Review, receipt. The relationships between these entities are described as follows:

1) User to Movie: One to Many

Each User can buy multiple tickets.

2) Movie to Order: Many to Many

Multiple Movies corresponds to multiple Orders.

3) Movie Category to Movie: One to Many

Each Category corresponds to many movies.

4) Movie to Movie Review: One to Many

Each movie corresponds to multiple reviews.

5) Order to Credit Card: Many to One

Each Credit Card can be used in multiple orders.

6) Order to Address: Many toOne

Each Address can be used in multiple orders.

7) Order to Receipt: One to One

Each Order corresponds to one Receipt.

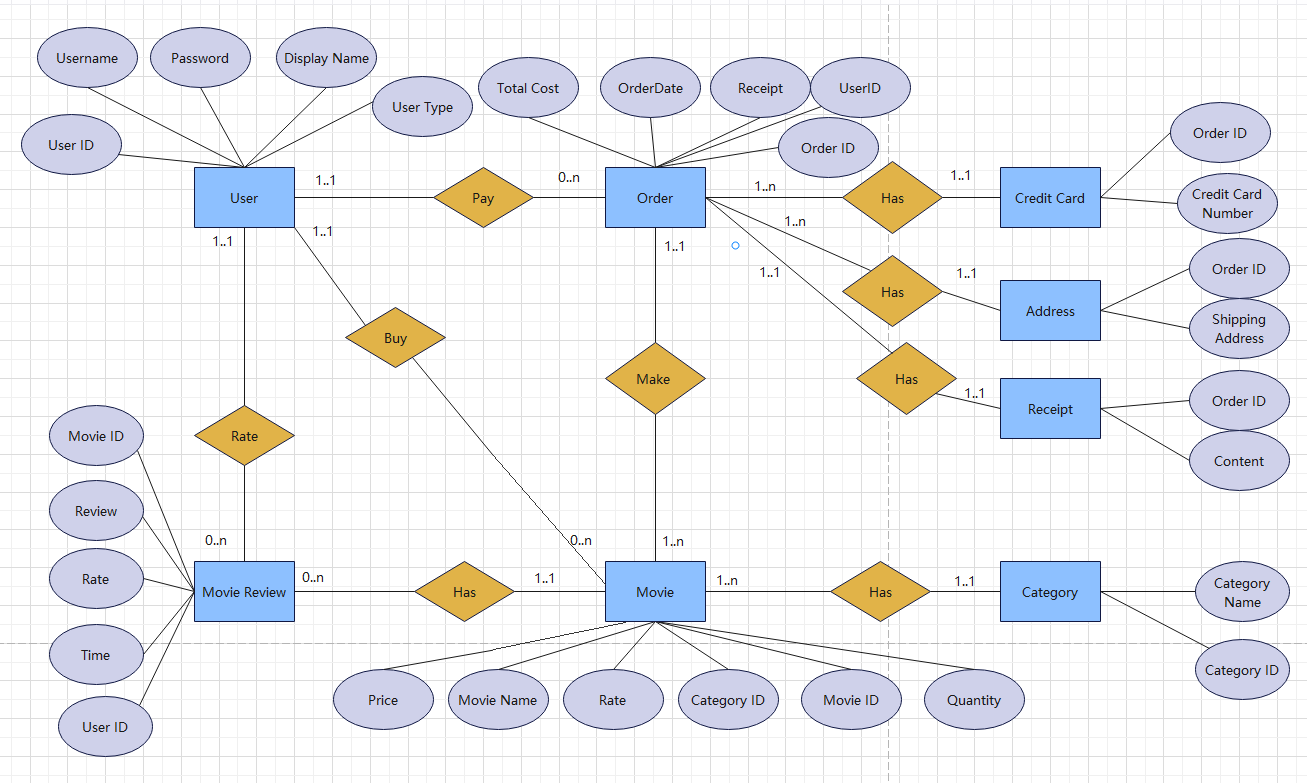
8) User to Movie Review: One to Many

Each User can write multiple Movie Reviews.

9) User to Order: One to Many

Each User can pay multiple Orders.

**2.2 ERD**



**2.3 Database storage strategy**

The data entities and objects are the same as previous iterations, the definition of which has not been changed. In this iteration, to enhance the performance of this movie ticket management system, we deploy all of our database storages on Redis and MongoDB cloud database platform, and select the specific cloud database according to the type and characteristics of each data entity and object:

1) Redis

Simple data objects like User, Ticket, Movie, Payment (Credit Card, Address), Movie Category are stored in Redis Database as HASH data structure. As these objects are mainly integer or float value, or short texts, using HASH data structure to store these objects could save storage space and reduce query or other operation complexity.

2) MongoDB

More complicated data objects like Order, Review, Receipt are stored in MongoDB as documents, considering these objects are primarily text oriented.

During constructing databases on these cloud database platforms and assigning their storage strategies, the data entities and objects are separated and divided into independent databases to ensure a relatively low coupling between different microservices, promoting smoother development and test process on user cases.

**2.4 Sample Data**

jedis.hset("user:admin", "username", "admin");

jedis.hset("user:admin", "password", "password");

jedis.hset("user:admin", "displayname", "Jing Cao");

jedis.hset("user:admin", "userType", "0");

jedis.hset("movie:1", "movieId", "1");

jedis.hset("movie:1", "movieName", "Titanic");

jedis.hset("movie:1", "price", "5.55");

jedis.hset("movie:1", "quantity", "94.0");

jedis.hset("movie:1", "rate", "5.0");

jedis.hset("movie:1", "categoryId", "1");

jedis.hset("category:1", "categoryId", "1");

jedis.hset("category:1", "categoryName", "Drama");

jedis.hset("category:2", "categoryId", "2");

jedis.hset("category:2", "categoryName", "Action");

jedis.hset("address:1", "orderId", "1");

jedis.hset("address:1", "address", "444 ball street");

jedis.hset("creditCard:1", "orderId", "1");

jedis.hset("creditCard:1", "creditCardNumber", "1234123412341234");

MongoCollection<Rate> collection = database.getCollection("reviews", Rate.class);

collection.insertOne(review);

MongoCollection<Order> collection = database.getCollection("orders", Order.class);

Bson filter = eq("orderID", id);

Bson update = set("receipt", URLDecoder.decode(receipts,"UTF-8"));

collection.updateOne(filter,update);

**3. Architecture Design: MVC**

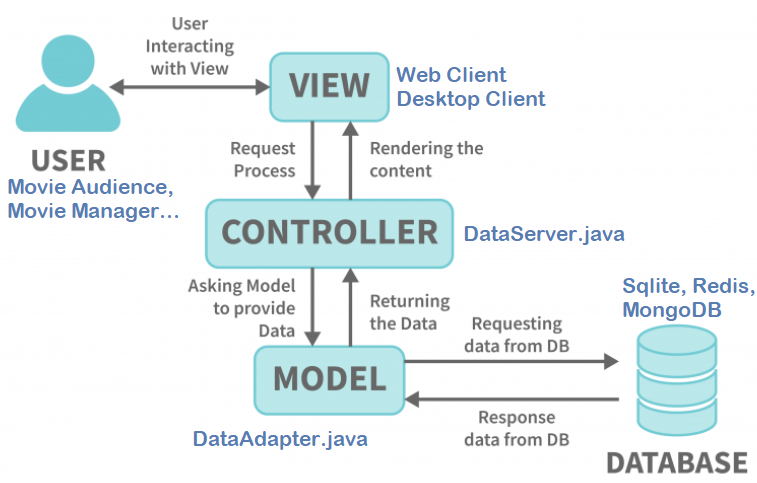
**3.1 Overall Diagram of System Architecture**

To ensure the maintainability and extensibility of our project, and enhance the test on code and enable the parallel development cycle of system components, we adopted the architecture to develop this system, where the desktop-based and web-based client are the view component that interacting directly with users. The server end built on Java functions as the controller, while the DataAdapter class in Java back-end logics functions as the model component within the MVC framework.

The MVC architecture breaks down the system into different components, allowing for a reduced complexity and clear analysis of overall objective and user functional needs, simplifying the development process with enhanced user experience.

The users of this system are targeted as movie audience and movie manager, while the databases used in the system include traditional sqlite database and non-SQL cloud databases as Redis and MongoDB. In this final iteration version of system, we replace the traditional SQL database storage with cloud databases services to enhance the system performance and allow for remote control and cloud deployment.

The diagram of our overall system architecture is as follows:



**3.1 Cloud Database: Redis and MongoDB**

We use Redis and MongoDB cloud database platforms to connect with Java backend logics and store respective entities according to data objects’ types.

To handle the different microservices of the system, we use independent databases for each microservice. In this system, since we are handling 4 microservices in the function of user management, payment management, movie management, and order management, we build 4 databases in total to execute respective operations. The databases and their corresponding tables are described as below:

**1) User Database**

Users: record the user information about UserID, Username, Password, DisplayName and UserType.

**2) Payment Database**

Addresses: record the address information entered by the user when making an order.

CreditCards: record the credit card information entered by the user when making an order.

**3) Movie Database**

Movies: record the movie information (Movie ID, CategoryID, Rate, Price, Quantity) that existed in the database or entered manually by the movie manager.

Category: record the category name corresponding to the CategoryID, which constructs a mapper between category name and categoryID.

**4) Order Database**

Orders: record each order the audience makes with its OrderID, OrderDate, CustomerID and Total Cost.

OrderLine: record every product purchased during making an order by a customer.

Receipts: record the text content of generated receipt every time a user makes an order.

Reviews: record the review content and rate ranked by users to movies.

**3.2 Client Components and Logics**

**1) GUI-based Client**

The main function of the GUI-based client is integrated in the Application class. It establishes the connection between client and server, with initializing basic logic, functions and corresponding interfaces (views) used in this application.

The LoginScreen class generates the Login interface and allows users to login in the system by correctly entering the username and password, or to sign up for a new account that enables users to create its own username and password with selecting the corresponding usertype.

The MainScreen class generates the Main interface that is responsible for interacting with movie audiences, allowing them to browse the movie list, buy their favorite movie tickets and rate movies.

The ManagerScreen class generates the Manager interface that enables movie managers to manage the movie information by loading or saving movie detailed information to the database.

The Movie class stores the structure of movies that can be used as an entity in other logics, with private attributes and getter and setter functions.

The MovieView class presents users the interface to browse the movie list, showing them the currently available movies, to enable users to have a better experience in deciding which movies to watch.

The Order class constructs the structure of an order, representing an order entity made by a user.

The Orderline class represents every item purchased during making orders by a user.

The OrderView class renders the interface of making an order for users, allowing them to add favorite items in the order list and submit their order for payment.

The Payment class is used to coordinate the connection between orderID, address and credit card, bind the databases with their corresponding orders, enable the user to store or access their order details and make a payment.

The PaymentView class provides an interface for users to enter their address and credit card information when making an order, and generate a receipt after a user successfully submits its payment.

The Product class constructs a structure that contains the basic information of a product, which regards every movie as a product that can be purchased by a user. It includes the quantity and price attribute in the structure.

The ProductView class provides a basic interface for movie managers to upload new movie information into the system, or update the existing movie information in the database, allowing users to acquire timely information about the movie.

The RateView class enables an interface for users to rank their likes and dislikes about the movie, where they can rate between 1 to 5 to show their preference, and type their comments and reviews, which will also be recorded in the database to adjust the recommendation weights.

The SignScreen class presents an interface for users to sign up for a new account, where users can create their own accounts by entering username, password, displayname and selecting the usertype. This user information will be recorded in the database for which users can directly log in next time.

The User class represents a user entity and stores the private information about a user and relevant getter and setter functions to ensure account security and privacy. The user entity can then be easily called elsewhere to get the current userID, userType, Displayname, etc., helping to identify the user’s status.

**2) Web-based Client**

The Web-based Client was crafted using HTML, CSS, and JavaScript, with all code in the ./web folder. HTML defines the structure of web pages, CSS enhances their appearance, and JavaScript introduces dynamic behavior and interactivity into web interfaces. We also use JavaScript to further manage the Document Object Model (DOM), allowing real-time updates, event handling, and seamless server communication via RESTful APIs and socket connections, facilitating smooth user-system interaction. This client provides interfaces for users to access the system on web browsers, allowing them to remotely interact with the system. The components and logic are the same as GUI-based Client, with integration of socket communication on RESTful APIs and DOM object handling in HTTP protocols and JavaScript logics.

The login folder contains the components representing the Login and Sign up webpage interface and corresponding functionality logics.

The homepage folder component represents the MainScreen webpage and relevant functionality, with dynamic logics to identify the current user type.

The menu folder component represents the respective webpages and functionalities in audience menu and manager menu, allowing them to access corresponding pages according to the user’s choice. This includes the buy tickets page, rate movie page and get recommendation page for movie audience, the manage movie information page for movie manager, and the browse movie list page for both movie audience and manager.

The payment folder contains components for the payment webpage and its functionality.

**3.3 Server Components and Logics**

Different from the previous version where the system is established on only one server and one sqlite database, this program deals with 4 microservices: user management, movie management, payment management, and review management. Those four microservices are constructed based on 4 different independent databases, which are based on cloud database services including Redis and MongoDB.

The DataServer class is used to build a server program to answer both GUI and Web clients about all requests on business data objects (e.g., movies, orders, users, receipts, credit cards...). It will initialize a socket to listen to the request from clients and identify the request type, executing corresponding functions, and asking DataAdapter class to provide requested data from cloud databases.

The DataAdapter class mainly functioned as connecting the server with the Redis and MongoDB databases, processing CRUD queries by submitting Redis and MongoDB statements to the databases, performing the demanded actions and designated operations, and returning the requested data to the server.

**3.4 Communication Protocol and Data Format**

We build a runnable system with GUI-based and Web-based client and microservices functions in the server, with accessing cloud databases like Redis and MongoDB.

To handle the communication between Web-based client and server, we mainly adopt the HTTP protocol and RESTful APIs, receiving and sending the request and response information according to the URL and HTTP body content. We also apply Sockets and RESTful APIs to manage the communication between GUI-based client and server, service registry, together with the communications between different microservices.

The data format we used in the communication includes RESTful API, Serialized String, URLEncoder (UTF-8), JSON and Stream Buffer. Meanwhile, we took several Redis and MongoDB statements and protocols to handle the communication with these cloud database platforms and execute demanded operations on them.

Here is the sample of data communication on web-based client:

'http://localhost:7777/login?username=' + encodeURIComponent(email) + '&password=' + encodeURIComponent(password)

fetch(url, {

method: 'GET',

credentials: 'include' })

In this example, we use HTTP protocol on the web client to request a response from the server by sending a URL with ‘GET’ method through websocket to the server. The URL follows RESTful API design pattern and integrates UTF-8 encoded information.

The server will handle the request from client and generate a response, integrating the json formatted objects into HTTP body, and setting the Content-Type to be "application/json” in the HTTP header:

StringBuilder responseHeaders = new StringBuilder();

responseHeaders.append("HTTP/1.1 200 OK\r\n");

responseHeaders.append("Content-Type: "application/json”\r\n");

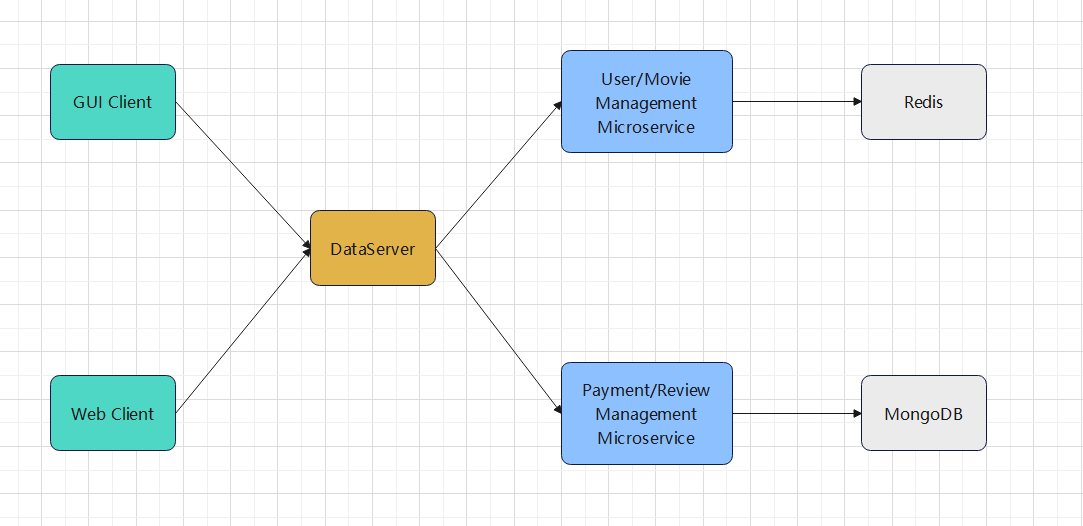
responseHeaders.append("Access-Control-Allow-Origin: http://localhost:63342\r\n");

responseHeaders.append("Access-Control-Allow-Credentials: true\r\n");

responseHeaders.append("Content-Length: ").append(responseBody.getBytes("UTF-8").length).append("\r\n");

StringBuilder responseBody = new StringBuilder();

responseBody.append(gson.toJson(user));



To use JSON data in communication, we adopt packages from com.google.gson.Gson library to support the encapsulation and decapsulation of json data. If we want to send a save product request to the server, we can simply create a json object and set the code to be corresponding request type, and we will integrate the encapsulated product object into the body of the HTTP request. After the server gets the request, it will extract the json object from the request body and decapsulate the json object by corresponding class attributes, and get the wanted data information.

In some cases, we can also simplify the communication process by using Stream Buffer format, where we use the space to split data segments, and integrate them together in one string sending to the server. On the server side, it will use the splitter to parse the information to different segments, extract the necessary information, assign them to relevant variables, and execute the corresponding actions. For example, when we try to save the receipt information, the client simply sends a string combining order id and order details, we can split the combined information once we get it on server, extract the needed data and perform the saving receipt operations on databases. Under certain circumstances, especially when the string we are sending contains space as a necessary content, then we can no longer use space as the splitter character. Instead, we adopt the “+” character as the splitter, or use UTF-8 encoder, the character encoding standard widely used in web communication, to encode the original string. In this way, we can allow for a proper transformation of data communication between client and server without losing or misinterpreting significant information.

**3.4 User Interface and View**

We totally designed 10 view interfaces: Login, SignUp, Main, Manager, Browse Movie List, Buy Tickets, Payment, Rate Movie, Manage Movies, Recommendation.

Each interface has two versions: GUI-based and web-based, which correspond to its respective frontend and backend logic, and executes necessary processes and actions per user’s input to generate a user-friendly and responsive interface.

The overall functionality of user interface is to interact with user’s input and process user request with proper actions and operations, sending requests to server and display the response information from server with proper extraction and rendering.

The specific functionality of each interface is defined according to the UI Sketch.

**4. Runnable System**

Please refer to the source code, database file and documentation in the attachment.

You are supposed to build a runnable system using the code in zip file following the installation manual steps.

**5. User Acceptance Tests Video (all in one)**

https://www.youtube.com/watch?v=SOKLPBlHaWU

All of the User Acceptance Tests passed on both GUI-based and web-based client (frontend) for both movie audience and manager, with its smooth communication with socket-based server and Redis and MongoDB database access.

The main functions, logics and operations are identical to the previous iteration version except for the new web-client designs and the new use case of recommending best movies to the movie audience according to the movie type entered by the user.

**6. Installation Manual**

Prerequisites:

* Download iteration\_3.zip from the provided source.
* Ensure the Java and JDK environment is properly established on your system.
* Ensure the latest version of IntelliJ IDEA (IDEA) is installed on your system.

Installation Steps and Running Guidelines:

1. Download and Unzip the Software
   1. Download iteration\_3.zip.
   2. Unzip the downloaded file to extract the contents from the zip file.
2. Open the Project in IntelliJ IDEA
   1. Launch IntelliJ IDEA.
   2. Navigate to File > Open and select the iteration\_3 folder.
3. Configure Project Dependencies
   1. Go to File > Project Structure.
   2. Under Modules, select the project module.
   3. Add the following JAR libraries as module dependencies:
      1. jedis-5.0.2.jar
      2. mongo-java-driver-3.12.14.jar
      3. gson-2.10.1.jar
   4. If dependencies aren't successfully added, configure Maven to ensure compatibility environment for building the project.
4. Adjust Java Language Level (if required)
   1. In case of a build failure, set the language level to Java 17 or above:
      1. Navigate to File > Project Structure.
      2. Under Project Settings, select Project.
      3. Set the language level to Java 17 or above.
5. Start the Server
   1. Run DataServer.java to start the server component.
6. Launch the GUI-Based Client
   1. Run Application.java to initiate the GUI-based client.
7. Access the Web-Based Client
   1. Open web/login/login.html using Chrome or Edge to access the web-based client.
8. Accessing MongoDB Data and Redis Data
   1. Note: MongoDB Cloud might restrict IP access.
   2. If required, send your IP to obtain permission for accessing MongoDB data.
   3. Redis Data can be directly accessed using correct URL without granting user permission.
9. Run User Acceptance Tests
   1. If you launch the GUI-Based Client, you can test every use cases on this GUI-based Client.
   2. If you launch the Web-Based Client, you can test every use cases on this Web-based Client.
   3. The GUI-Based Client and Web-Based Client shares the same server, thus there’s no costs changing between these two kinds of clients.