MUNHouse - Project Design

ENGI - 9837 Software Engineering Capstone Project

Memorial University of Newfoundland

Submitted by :

Muhammad Shaheryar	202285690
Waqas Younis	202291526
Matti Ur Rehman	202292269
Kendesha	202286998
Subarna Chowdhury	202291603
Heshani Hettiarachchi	202292233

Introduction

Welcome to the MunHouse app, a web-based real-estate application that allows house owners and buyers to engage in real estate transactions by leveraging modern technologies, microservices, and a RESTful API.

The purpose of this document is to :
- · Define the architectural framework MunHouse app
- · Highlight the key features and functionalities for house owners and buyers.
- · Outline the database structure, and structure of microservices.
- · Describe the testing strategy.

This project will encompass the creation of a web-based real estate application, offering services for two distinct user roles: house owners and buyers. House owners will be able to list their properties, while buyers can search, filter, and request house viewings. The system will be backed by a microservices architecture, ensuring modularity and scalability.

Technology Stack

Front End
> UI Prototyping - Figma
> UI Development - React JS , Tailwind UI Framework

Backend
> Database - Mongo DB
> Backend Services - Node JS
> Communicating Between Services - Queues / API Calls

User Roles and Features
- · User roles: House Owner and Buyer.
- · List the key features for each role:
  - o Owner:
    - ● Add a house for sale.
    - ● Manage house listings.
    - ● Manage viewing requests
  - o Buyer:

- Search for real estate.
- Set price range, bedrooms, and bathrooms.
- Request house viewings.
- Add listing to their favorites
- Message Owner

Microservices

- User Authentication Service
- User profile Service
- Map integration/Geo-location Service
- House Listing Service
- Search and filter Service
- House Viewing Request Service
- Chat/Messaging service
- Notification Service
- Favorites Service
- Image Storage Service
- User History Service
- Review and Rating Service (optional)

Microservices Architecture Diagram

```
┌─────────────────┐      ┌──────────────────────────────────────────────────────────┐
│   MUNhouse      │      │                    Microservices                         │
│ Web-Application │      │                                                          │
│   (Front-End)   │      │  ┌──────┐  ┌──────────────────────────────┐     ┌───┐    │
│                 │      │  │      │──▶│  User Authentication Service │────▶│ DB│    │
│                 │      │  │      │  └──────────────────────────────┘     └───┘    │
│                 │      │  │      │──▶│  User profile Service        │────▶│ DB│    │
│                 │      │  │      │  └──────────────────────────────┘     └───┘    │
│                 │      │  │      │──▶│  Map integration Service     │────▶│ DB│    │
│   [devices]     │─────▶│  │ API  │──▶│  House Listing Service       │────▶│ DB│    │
│                 │      │  │Gateway│──▶│  Search and filter Service   │────▶│ DB│    │
│                 │      │  │      │──▶│  House Viewing Request Service│───▶│ DB│    │
│                 │      │  │      │──▶│  Messaging service           │────▶│ DB│    │
│                 │      │  │      │──▶│  Notification Service        │────▶│ DB│    │
│                 │      │  │      │──▶│  Favorites Service           │────▶│ DB│    │
│                 │      │  │      │──▶│  Image Storage Service       │────▶│ DB│    │
│                 │      │  │      │──▶│  User History Service        │────▶│ DB│    │
│                 │      │  │      │──▶│  Review and Rating Service   │────▶│ DB│    │
│                 │      │  └──────┘  └──────────────────────────────┘     └───┘    │
└─────────────────┘      └──────────────────────────────────────────────────────────┘
```
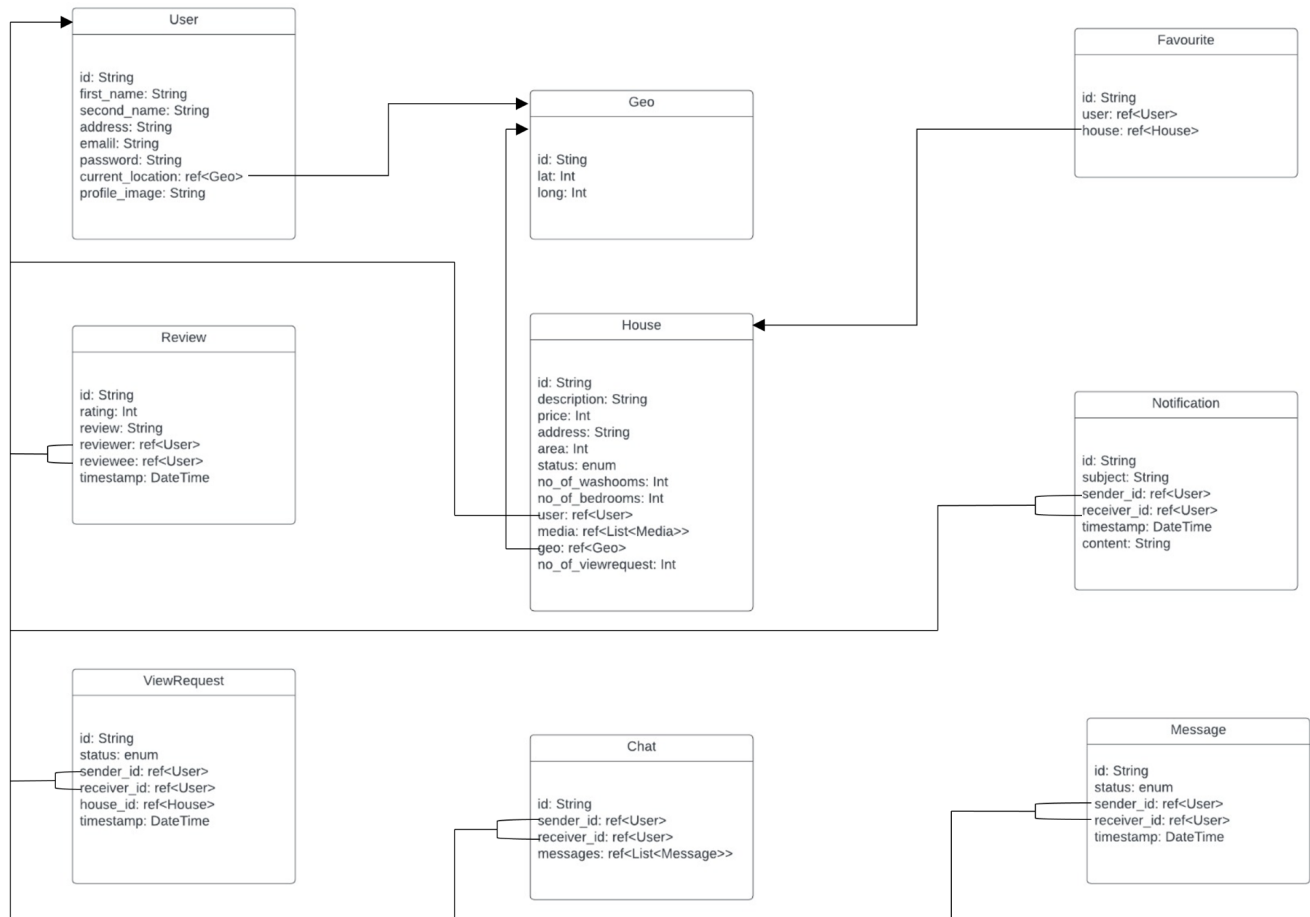
Database Design

Database design for the MUNhouse web-application is given below.

User

id: String
first_name: String
second_name: String
address: String
emalil: String
password: String
current_location: ref<Geo>
profile_image: String

Geo

id: Sting
lat: Int
long: Int

Favourite

id: String
user: ref<User>
house: ref<House>

Review

id: String
rating: Int
review: String
reviewer: ref<User>
reviewee: ref<User>
timestamp: DateTime

House

id: String
description: String
price: Int
address: String
area: Int
status: enum
no_of_washooms: Int
no_of_bedrooms: Int
user: ref<User>
media: ref<List<Media>>
geo: ref<Geo>
no_of_viewrequest: Int

Notification

id: String
subject: String
sender_id: ref<User>
receiver_id: ref<User>
timestamp: DateTime
content: String

ViewRequest

id: String
status: enum
sender_id: ref<User>
receiver_id: ref<User>
house_id: ref<House>
timestamp: DateTime

Chat

id: String
sender_id: ref<User>
receiver_id: ref<User>
messages: ref<List<Message>>

Message

id: String
status: enum
sender_id: ref<User>
receiver_id: ref<User>
timestamp: DateTime

Microservices

1. User Authentication

The User Authentication Service ensures a secure and seamless authentication process. It manages user registrations, logins, session handling, and password resets. By utilizing session tokens, it enhances system security and protects user accounts from unauthorized access.

Endpoints

POST /auth/register
Description: Creates a new user account.
Request: user object
Response: status 204 with success message

POST /auth/login
Description: Authenticates a user and generates a session token.
Request: user's login credentials
Response: status 200 with jwt token

POST /auth/validate:
Description: Validates the session token.
Response: status 200 if jwt is valid and 401 if jwt is invalid.

POST /auth/reset_password
Description: Initiates the password reset process.
Request: user's email as a string
Response: status 200 with success message

2. User Profile

The User Profile Service manages user profile details and provides endpoints to access and update this information. It includes attributes such as ID, first name, last name, email, password, location, and profile image.

Endpoints

GET /user
Description: Retrieves user profile information.
Response: status 200 with user object

PUT /user
Description: Updates user profile information.
Request: partial user object
Response: status 200 with user object

POST /user/image
Description: Uploads a new profile image.
Request: image as a file
Response: status 200 with success message

GET /user/image
Description: Retrieves the user's profile image.
Response: status 200 with image url

3. Map Integration/Geo-Location

The Map Integration/Geo-location Service is a critical component of the proposed real-estate app for MUNhouse. This service is responsible for handling all functionalities related to geographic location and map-based interactions within the application by integrating Google map and its APIs. Its primary purpose is to provide a user-friendly map interface for both house owners and buyers, allowing them to search for real estate listings in specific neighborhoods and view relevant property information on a map.

Endpoints

- **Google Maps JavaScript API:** Allows to embed interactive maps on web pages and customize them with various features, markers, and overlays. Provides functionality for map display, interaction, and customization.
- **Google Maps Geocoding API:** Converts addresses into geographic coordinates (latitude and longitude) and vice versa. Useful for geocoding user input or displaying locations on a map.
- **Google Maps Directions API:** Provides directions and estimated travel times between locations. Useful for displaying routes and turn-by-turn directions on maps.
- **Google Places API:** Offers access to a wide range of location-based data, including places of interest, establishments, and geographic details. Includes sub-services like Place Details, Place Search, and Autocomplete.
- **Google Maps Distance Matrix API:** Calculates distances and travel times between multiple origins and destinations. Useful for optimizing routes and determining the closest points of interest.

4. House Listing

The House Listing Service within the MUNhouse real estate application is an essential microservice responsible for facilitating the listing and management of real estate properties. It serves as a central hub for house owners (sellers) to showcase their properties for potential buyers, ensuring that property details are accurately presented and efficiently managed within the platform.

Endpoints

POST /listing

Description: creates a new listing
Request: listing object
Response: status 204 status code and listing object

GET /listing/{listingId}
Description: Retrieves a listing by Id
Response: status 200 status code and listing object

GET /listing
Description: retrieves all listing.
Response: status 200 status code and are of listing object

PUT /listing/{listingId}
Description: updates an existing listing
Request: partial listing object
Response: status 204 status code and listing object

5. Search and Filter

This module enables real-time search and filter capability for houses listed. The search feature mainly offers searching by keyword, which can be the location name, and the filter feature allows users to narrow down further their preference, including the price range, area, number of bedrooms, number of washrooms, and number of stories.

Endpoints

GET /search/houses
Description: Allows users to search for house listings based on keywords.
Query parameters: Filter by Location, Price, and Bedrooms
Response: Response: status 200 code and list of house listing objects

GET /search/houses/filter
Description: Allows users to filter house listings based on location, price range, and the number of bedrooms. Filter by Location, Price, and Bedrooms
Query parameters: {
location: The location or city where the house is located.
min_price: The minimum price of the house.
max_price: The maximum price of the house.
min_bedrooms: The minimum number of bedrooms in the house.
max_bedrooms: The maximum number of bedrooms in the house.

min_washrooms: The minimum number of washrooms in the house.
max_washrooms: The maximum number of washrooms in the house.
}
Response: status 200 code and list of house listing objects

6. House Viewing Request

This microservice provides the coordination between seller and buyer to create a view request. The buyer will be given a list of availability of the seller that the buyer can choose from according to his/her preferences.

Endpoints

POST /request
Description: Allows a buyer to create a new view request. The request is sent to the seller for confirmation.
Request: request object
Response: status 204 and success message

PUT /request/{Id}
Description: Updates status, time for a specific seller for confirmation.
Request: partial request object
Response: status 200 and request object

GET /requests/buyer/{Id}
Description: Retrieves a list of view requests created by a specific buyer.
Response: : status 200 and array of request objects

7. Chat/Messaging

This module is responsible for enabling real-time messaging and communication between buyers and owners allowing them to discuss property details, arrange viewings, and negotiate terms. It manages the creation of chat conversations, sending and receiving messages.

Endpoints

POST /chats
Description: Sends a message to a user

Request : message object
Responses: status 204 with message object

GET chats/{ receiverId }
Description: Retrieves list of conversations for a specific user
Response : 200 with array of message objects

8. Notification

This module manages the sending of notifications to users based on various events and actions within the app.

Endpoints

POST /notifications
Description: Save notification in database
Request: notification object
Response: 204 with notification object

POST /nofitications/subscribe
Description: Allows users to subscribe to various notification channels.
Request: list of channels user want to subscribe to
Response: 200 with success message

POST /notifications/unsubscribe
Description: Allows users to unsubscribe to various notification channels.
Request: list of channels user want to unsubscribe from
Response: 200 with success message

9. Favorites

The Favorites microservice is responsible for managing the favorite properties of users in the real estate application. Users can add and remove properties from their list of favorites. This service ensures that users can easily access properties they are interested in without the need to search for them repeatedly.

Endpoints

POST /favorites/add
Description: Allows a user to add a property to their list of favorites.
Request: { "userId": "user_id", "houseId": "house_id" }

Response: Status code (e.g., 204 OK)

DELETE /favorites/remove
Description: Allows a user to remove a property from their list of favorites.
Request: { "userId": "user_id", "houseId": "house_id" }
Response: Status code (e.g., 200 No Content)

GET /favorites/user/{userId}
Description: Retrieves a list of favorite properties for a specific user.
Request: {}
Response: JSON array of favorite property objects

10. Image Storage

The Image Storage Service is responsible for handling the storage and retrieval of property images and profile images in the real estate application. It ensures efficient and secure storage of images, making them accessible to the application as needed.

Endpoints

POST /images/property
Description: Allows users to upload property images.
Request: Image file upload
Response: Status code and image URL

POST /images/user
Description: Allows users to upload profile images.
Request: Image file upload
Response: Status code and image URL

GET /images/{imageName}
Description: Allows retrieving images by their filenames.
Request: Image filename
Response: Image content

DELETE /images/{imageName}
Description: Allows users to delete images by their filenames.
Request: Image filename
Response: Status code (e.g., 204 No Content)

11. User History

The User History Service module tracks and stores all user activities and interactions within the application, such as search queries and viewed listings.

Endpoints:

POST /user-history
Description: Records a user's activity within the application
Request: user-history object
Response: status 204 with success message

GET / **user-history**
Description: Retrieves a list of recorded activities for a specific user.
Response: status 200 with array of user-history object

DELETE /user-history/**{historyId}**
Description: allows user to remove a history
Response: status 200 with success message

12. Review & Rating

The Review & Rating Service manages and displays user feedback about properties. This module allows users to post their reviews and assign ratings to users, offering valuable insights to future renters or buyers.

Endpoints:

POST **/reviews**
Description: Allows a buyer to submit a review and rating for an owner
Request: review object
Response: status 204 with success message

GET **/reviews**
Description: Retrieves all reviews for a specific user
Request: review object
Response: status 200 status with an array of review objects

PUT /reviews/{reviewId}

Description: Allows a reviewer to update their review and rating.
Request: partial review object
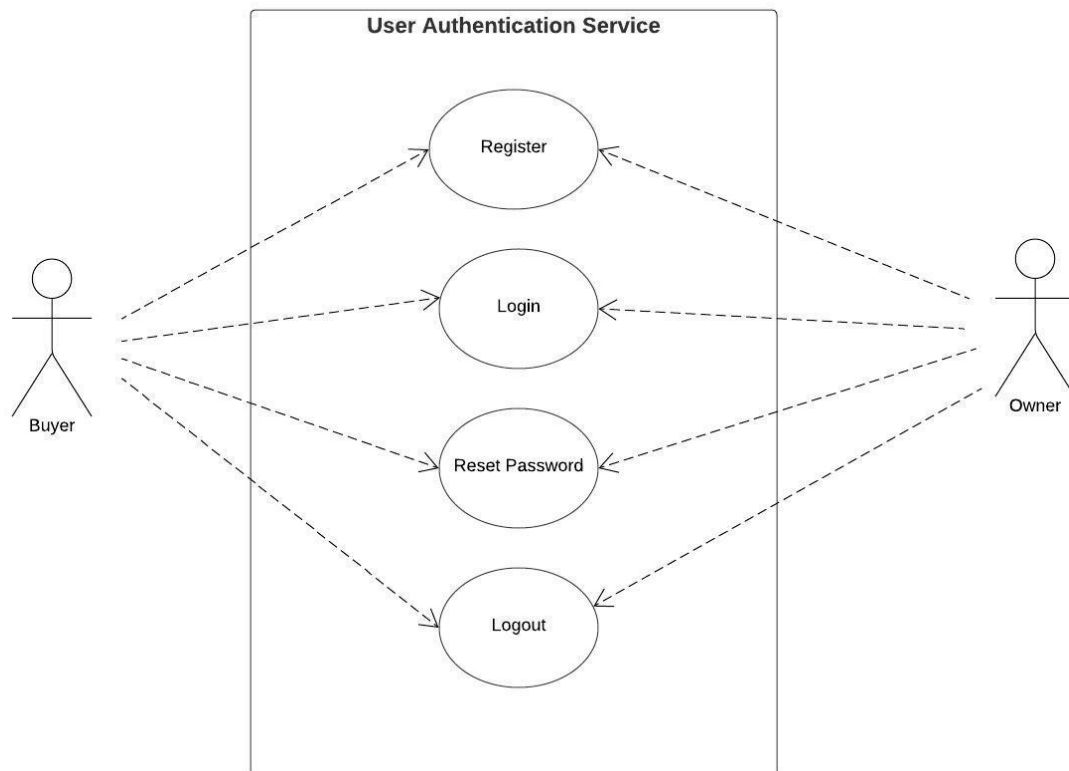Response: status 200 status with success message

DELETE **/reviews/{reviewId}**
Description: Allows a reviewer to delete a review
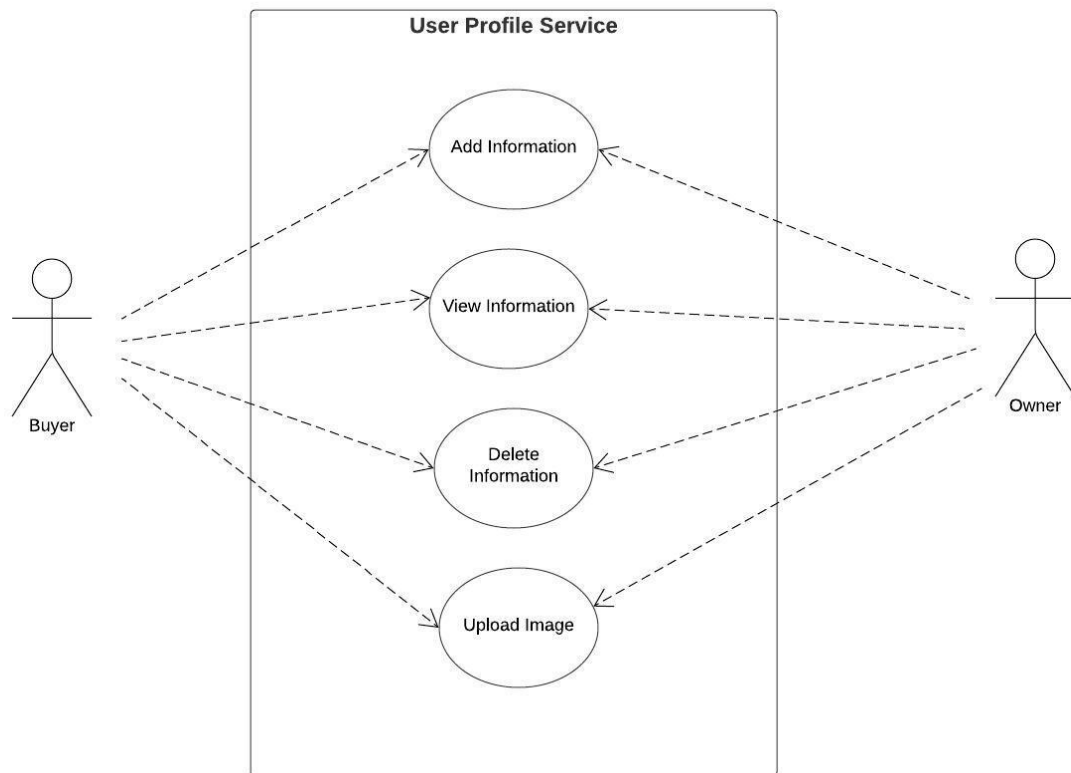Response: status 200 status with success message

UML Diagrams
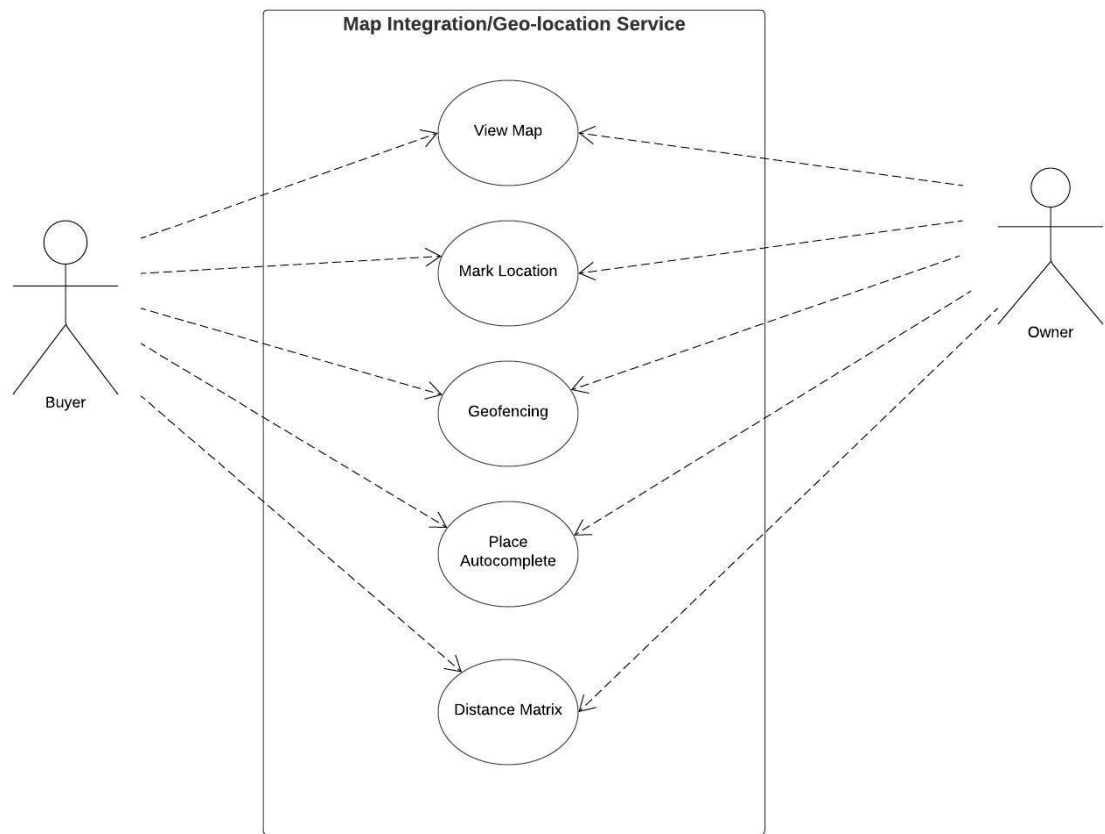
The Use Case Diagram for each microservice is given below:

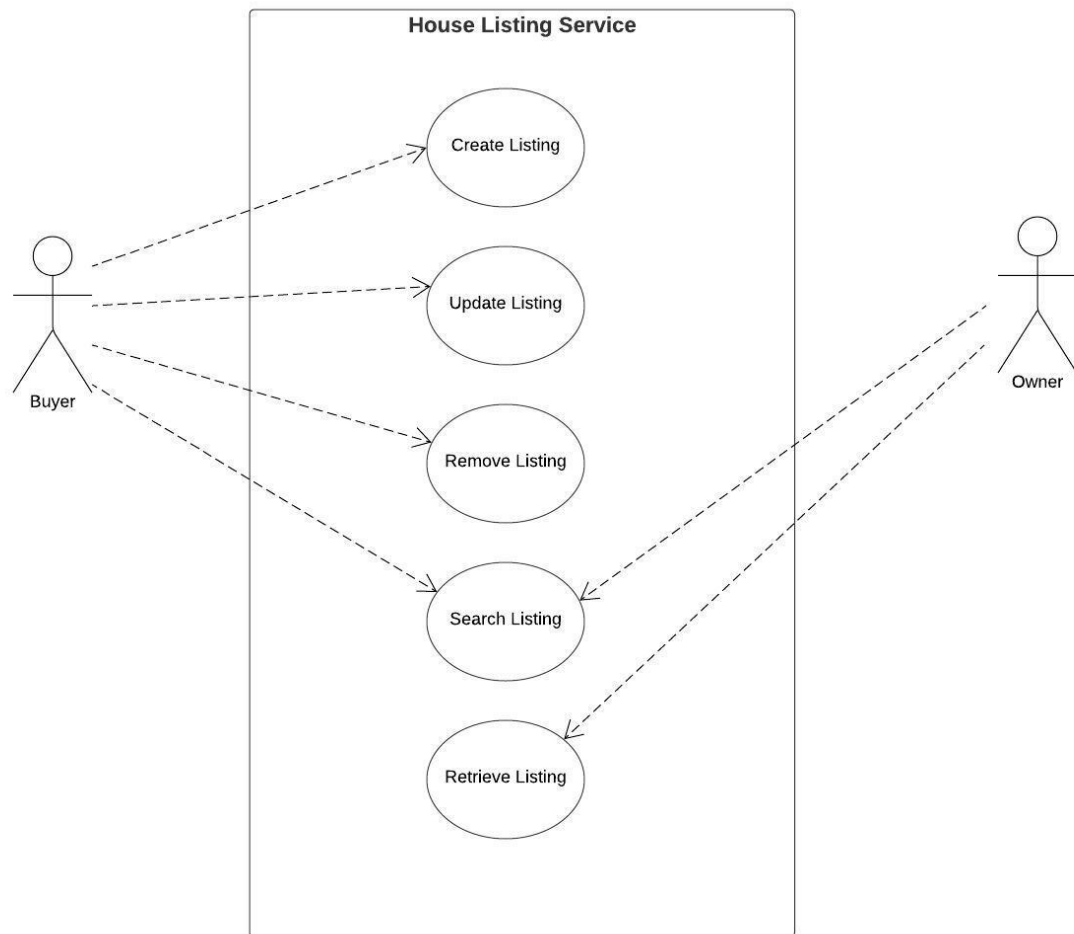1. **User Authentication Service**

2. User Profile Service
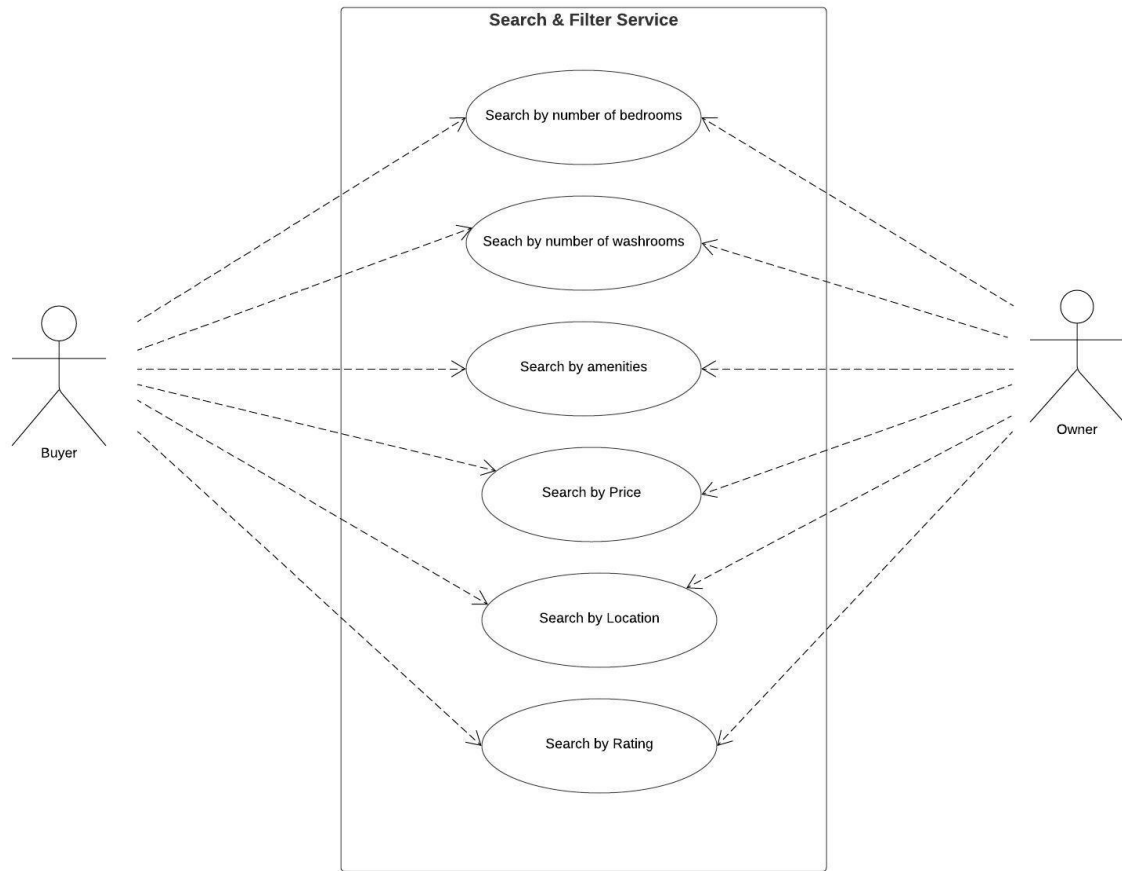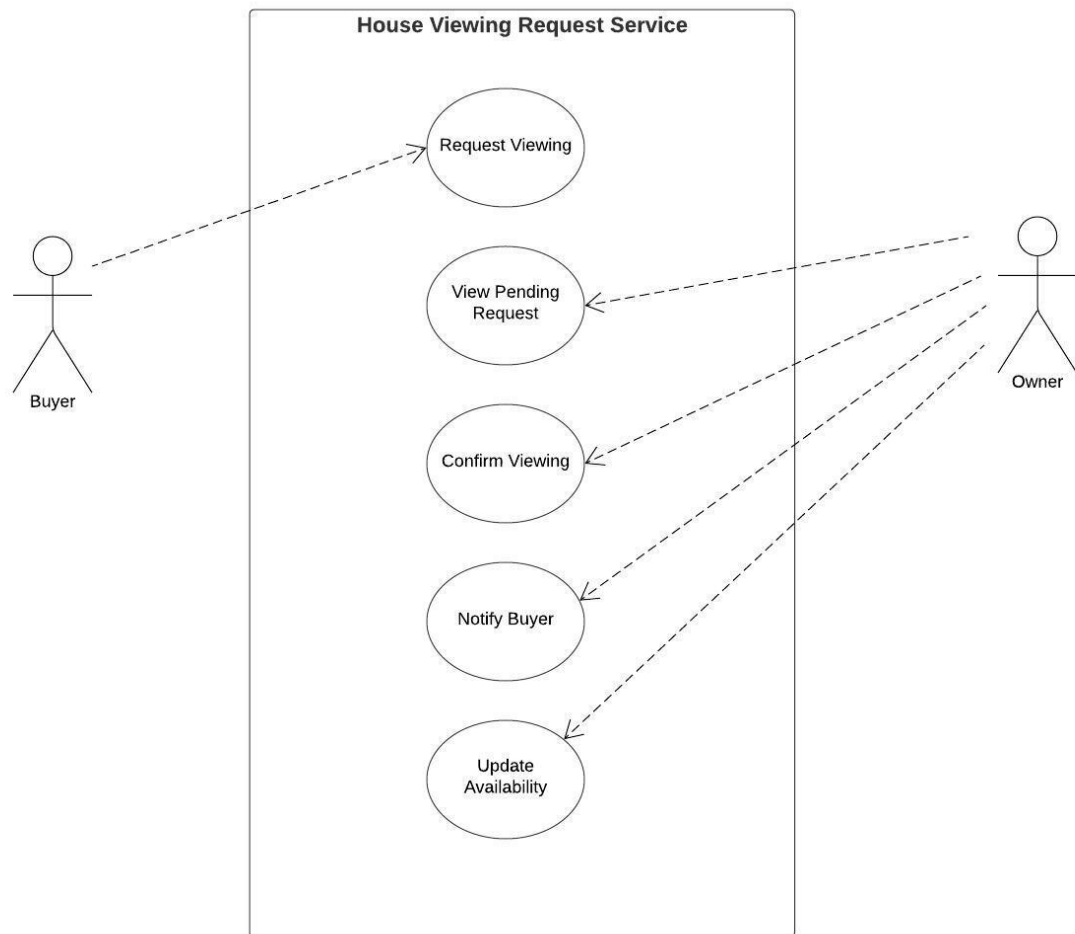
3. Map Integration/Geo-location Service
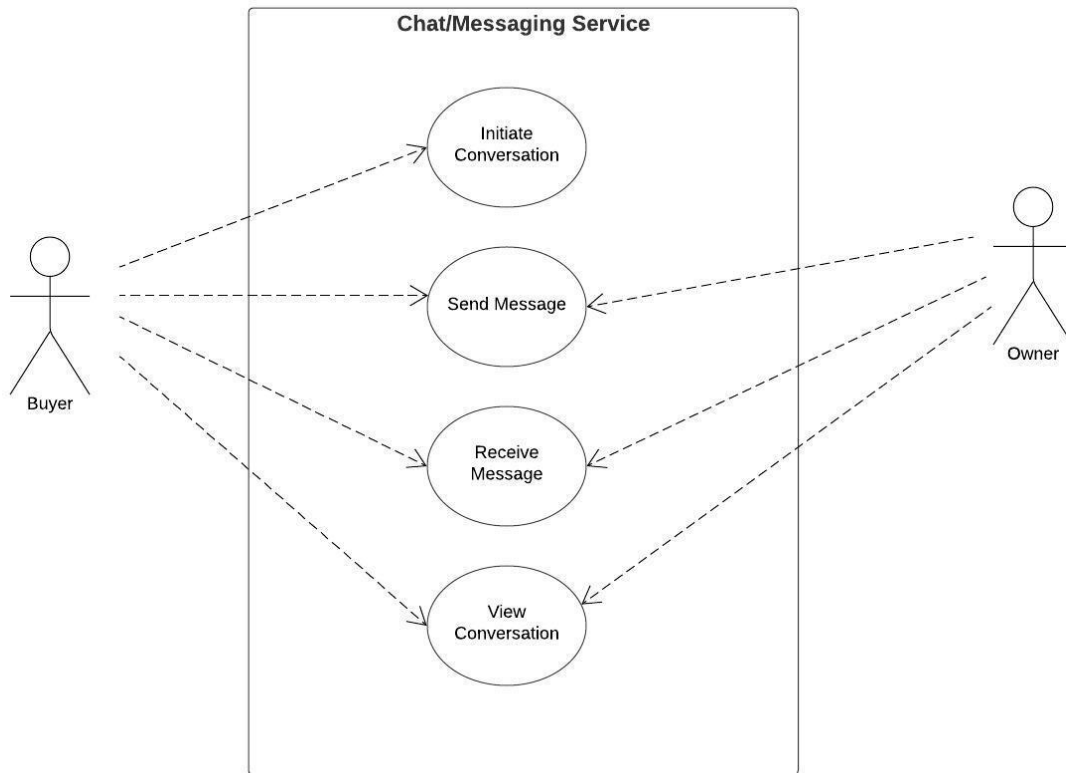
4. House Listing Service
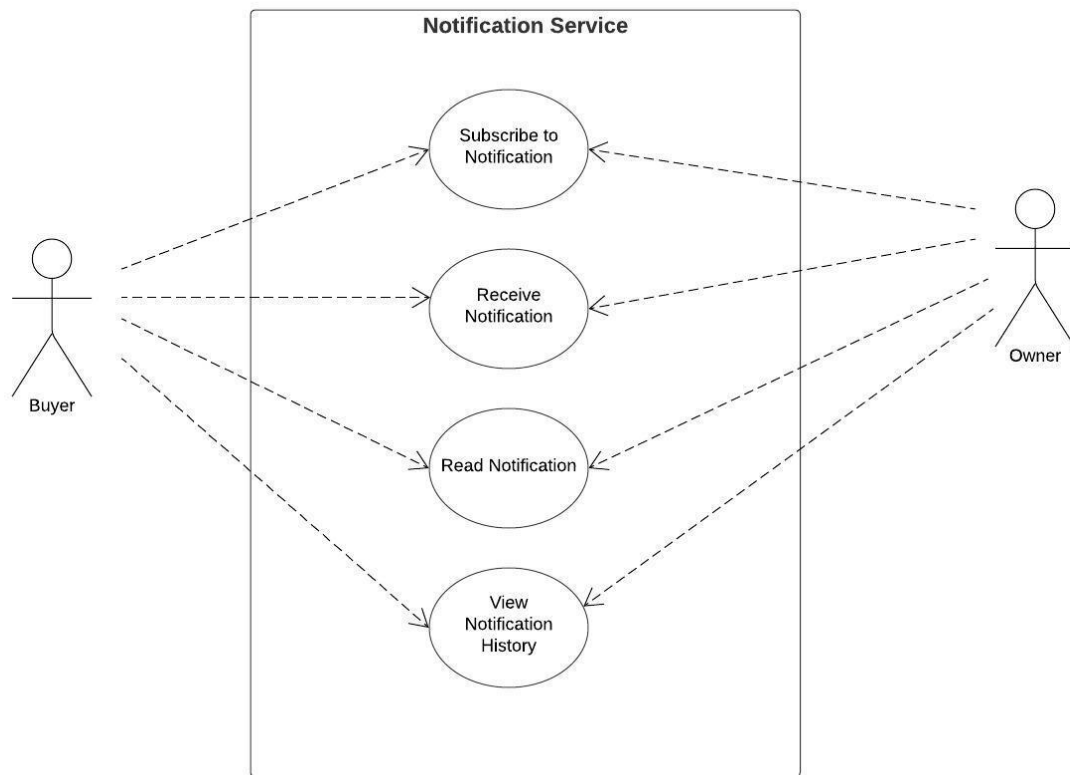
5. Search & Filter Service

6. House Viewing Request Service

7. Chat/Messaging Service

8. Notification Service

9. Favorites Service

Favorites Service

Add a property for favorites

Remove a property from favorties

View Favorites List

Buyer
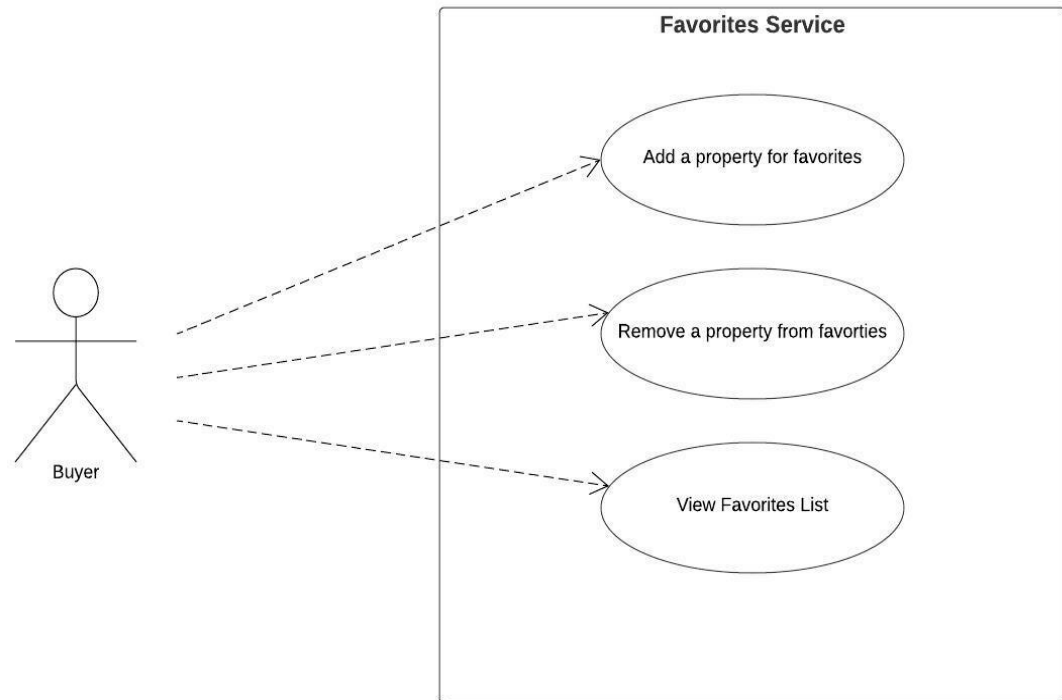
10. Image Storage Service
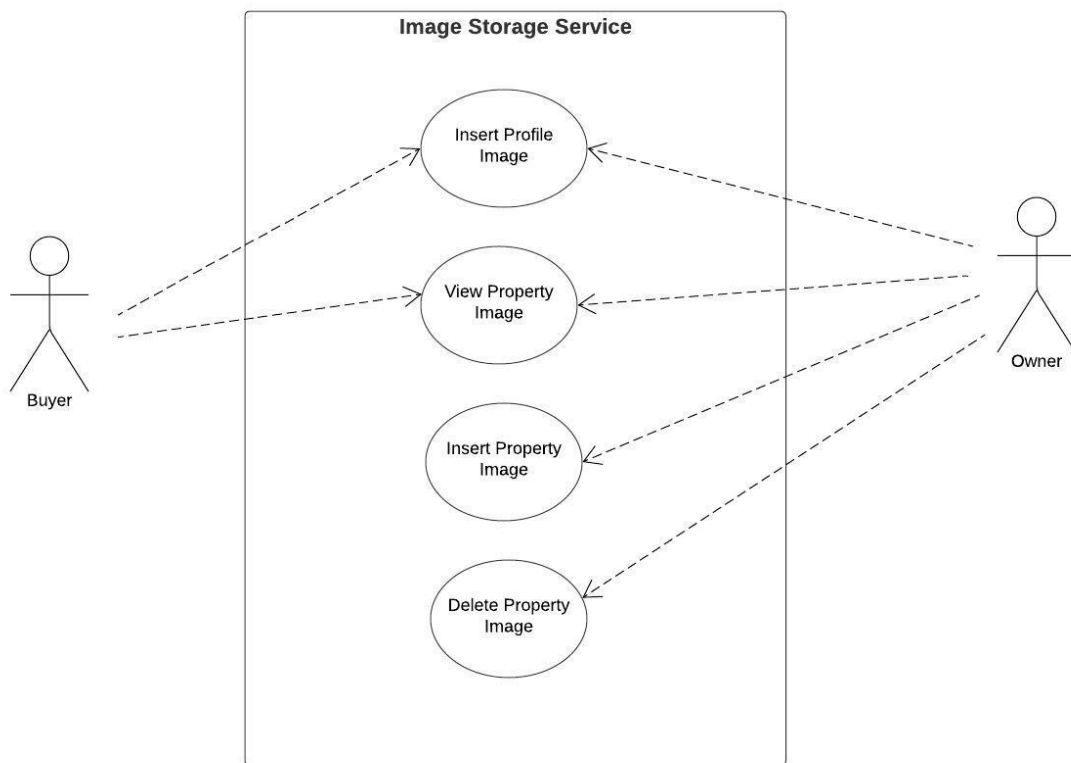
11. User History Service
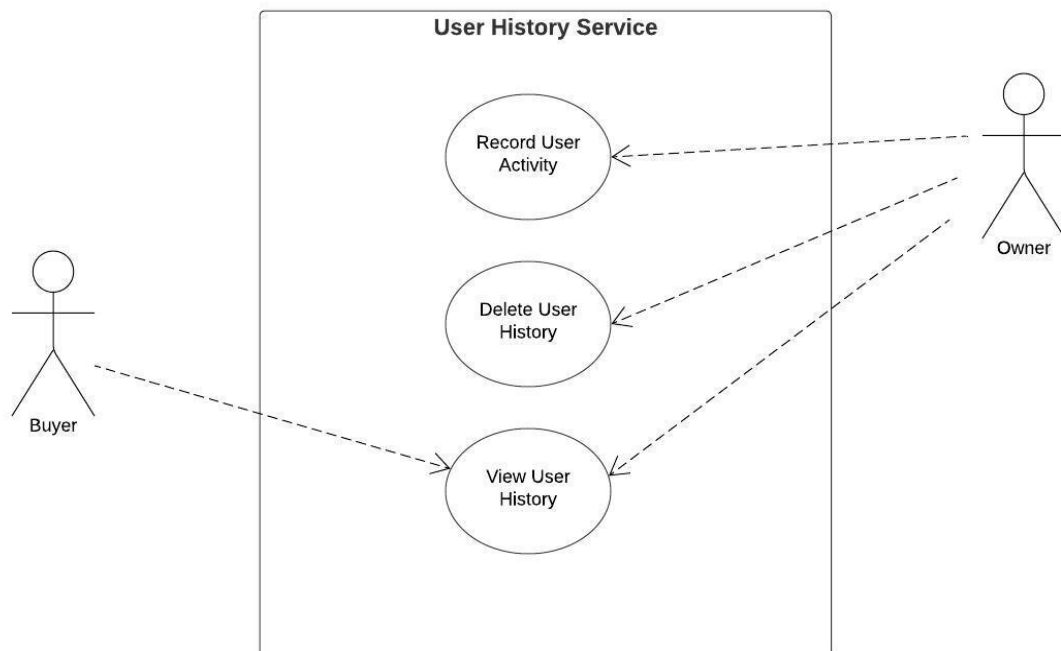
12. Review & Rating Service

Testing

Testing the MUNHOuse application will happen along with the development and as a whole, after the development to ensure the application's reliability, functionality, and usability. Testing will be conducted in two phases: frontend testing and backend testing, with a focus on component testing, usability testing, and microservice-level testing.

Frontend Testing

1. Component Testing

Component testing involves a manual examination of individual components and UI elements within the frontend of the application.Components include pages, forms, buttons, navigation elements, and any interactive elements.

Approach:
- A checklist will be maintained to ensure that each component is tested for functionality, appearance, and usability.
- Also we will be following pseudocodes to conduct structured tests.

2. Usability Testing

Usability testing evaluates the overall user experience and user-friendliness of the application.
This phase focuses on how easily users can navigate the application, perform tasks, and achieve their goals. The feedback from the selected users will be considered to improve UI/UX of the application.

Backend Testing

Backend testing will be conducted at the microservice level to ensure the functionality, reliability, and integrity of each microservice.Each microservice will be tested independently before integrated testing.

Approach:

For each microservice, unit testing will be performed to validate individual functions, APIs, and endpoints.
Since the backend services will be developed in "Node JS" the test cases are done in "Mocha".

System Testing

System testing is the final phase of backend testing and aims to validate the overall behavior of the entire application which verifies interactions and data flows between microservices, ensuring seamless communication in the MUNHouse application.

Once individual microservices have passed unit and integration testing, system testing will be conducted.System-level test cases will cover end-to-end scenarios, including user journeys that span multiple microservices.

Unit Tests Based on each microservice

1. **User Authentication Service**
   a. Register a New User: This test registers a new user by sending a POST request to /register with valid registration data. It expects a successful response with a status code of 201 (Created).

   > *Test user registration*
   > *it('should register a new user', async () => {*
   >   *const response = await request(app)*
   >   *.post('/register')*
   >   *.send({*
   >   *username: 'testuser',*
   >   *password: 'testpassword',*
   >   *});*
   >
   >   *expect(response.statusCode).toBe(201); // 201 Created*
   >   *expect(response.body).toHaveProperty('message', 'User registered successfully');*
   >   *});*

   b. Authenticate User and User Login: This test logs in a user by sending a POST request to /login with valid login credentials. It expects a successful response with a status code of 200 and receives an authentication token.

   > *Test user login*
   > *it('should authenticate a user and return an authentication token', async () => {*
   >   *const response = await request(app)*
   >   *.post('/login')*
   >   *.send({*
   >   *username: 'testuser',*
   >   *password: 'testpassword',*
   >   *});*
   >
   >   *expect(response.statusCode).toBe(200); // 200 OK*
   >   *expect(response.body).toHaveProperty('token');*
   >   *authToken = response.body.token; // Store the authentication token for later tests*
   >   *});*

   c .Log Out User: This test logs out the user by sending a POST request to /logout with the valid authentication token. It expects a successful logout response with a status code of 200.

   > *Test user logout*

```
it('should log out a user and invalidate the authentication token', async () => {
  const response = await request(app)
    .post('/logout')
    .set('Authorization', `Bearer ${authToken}`); // Set the authentication token

  expect(response.statusCode).toBe(200); // 200 OK
  expect(response.body).toHaveProperty('message', 'User logged out successfully');
});
```

2. User profile Service

a. **Retrieve User Profile:** This test verifies that a user can retrieve their own profile information by sending a GET request to /user with a valid authentication token. It expects a successful response with a status code of 200 and checks the retrieved user profile attributes.

```
Test retrieving user profile
it('should retrieve user profile information', async () => {
  const response = await request(app)
    .get('/user')
    .set('Authorization', `Bearer ${authToken}`);

  expect(response.statusCode).toBe(200); // 200 OK
  expect(response.body).toHaveProperty('username', 'testuser'); // Check the username
  expect(response.body).toHaveProperty('email', 'testuser@example.com'); // Check the email
});
```

b. **Update User Profile:** This test updates the user's profile by sending a PUT request to /user with updated profile attributes. It expects a successful response with a status code of 200 and checks if the profile has been updated correctly.

```
Test updating user profile
it('should update user profile information', async () => {
  const updatedProfile = {
    username: 'newusername',
    email: 'newuser@example.com',
  };

  const response = await request(app)
    .put('/user')
    .set('Authorization', `Bearer ${authToken}`)
    .send(updatedProfile);

  expect(response.statusCode).toBe(200); // 200 OK
  expect(response.body).toHaveProperty('message', 'User profile updated successfully');
});
```

3. Map integration/Geo-location Service

a. **Map Interface Interaction Test:** This test ensures that the map interface is responsive and allows users to interact with it properly. It checks if users can zoom in and out, pan to different locations, and that the map responds accurately to their actions.

> *Test Map Interface Interaction Test*
> *it('should allow users to interact with the map interface', function () {*
>   *// Simulate user interactions with the map interface*
>   *const userInteractionResult = mapService.interactWithMap();*
>
>   *// Assert that the result of user interaction is as expected*
>   *assert.strictEqual(userInteractionResult, true);*
> *});*

b. **Geolocation Accuracy Test:** This test validates that the geolocation data used to display properties on the map is accurate. It should verify that properties are displayed at their correct coordinates and within the correct neighborhood boundaries.

> *Test Geolocation Accuracy Test*
> *it('should display properties accurately on the map', function () {*
>   *// Simulate retrieving property location from the database*
>   *const propertyCoordinates =*
> *mapService.getPropertyCoordinates(mockPropertyData.id);*
>
>   *// Assert that the retrieved coordinates match the expected coordinates*
>   *assert.deepStrictEqual(propertyCoordinates, mockPropertyData.coordinates);*
> *});*

c. **Database Integration Test:** This test checks whether the Map Integration/Geo-location Service can successfully interact with the database to create and retrieve property location information, such as property coordinates.

> *Test Database Integration*
> *it('should integrate with the database for property location information', function () {*
>   *// Simulate fetching property location from the database*
>   *const propertyCoordinates =*
> *mapService.getPropertyCoordinates(mockPropertyData.id);*
>
>   *// Assert that the retrieved coordinates match the expected coordinates*
>   *assert.deepStrictEqual(propertyCoordinates, mockPropertyData.coordinates);*
> *});*

d. **Fault Tolerance Test:** This test evaluates the fault tolerance of the service. It ensures that even when other microservices are temporarily unavailable, users can still use basic map

functionality, such as browsing the map, viewing basic property information, and accessing their search history.

> ***Test Fault Tolerance Test***
> *it('should remain functional with reduced functionality during service outages',*
> *function () {*
>   *// Simulate unavailability of other microservices*
>   *mapService.simulateServiceOutage();*
>
>   *// Simulate user interaction during the service outage*
>   *const userInteractionResult = mapService.interactWithMap();*
>
>   *// Assert that the map interaction is still possible during the outage*
>   *assert.strictEqual(userInteractionResult, true);*
> *});*

e. **Integration with Mapping API Test:** This test confirms that the service effectively integrates with mapping APIs to retrieve geographic information. It should validate that the API calls return the expected data.

> ***Test Integration with Mapping API Test***
> *it('should integrate with mapping APIs for geographic information', function () {*
>   *// Simulate fetching neighborhood data from the mapping API*
>   *const neighborhoodData = mapService.getNeighborhoodData(40.7128, -74.0060);*
>
>   *// Assert that the retrieved neighborhood data is as expected*
>   *assert.deepStrictEqual(neighborhoodData, { neighborhood: 'Sample*
> *Neighborhood' });*
> *});*

4. House Listing Service

a. **Create a New Listing:** This test creates a new listing by sending a POST request to /listing with the necessary listing data. It expects a successful response with a status code of 204 (No Content) and stores the ID of the created listing for future tests.

> ***Test creating a new listing***
> *it('should create a new listing', async () => {*
>   *const newListing = {*
>   *};*
>
>   *const response = await request(app)*
>   *.post('/listing')*
>   *.set('Authorization', `Bearer ${authToken}`)*
>   *.send(newListing);*

```
            expect(response.statusCode).toBe(204);
            listingId = response.body.id;
        });
```

b. **Retrieve a Listing by ID:** This test retrieves a listing by its ID using a GET request to /listing/{listingId}. It expects a successful response with a status code of 200 (OK) and validates the structure of the retrieved listing object.

```
        Test retrieving a listing by ID
        it('should retrieve a listing by ID', async () => {
          const response = await request(app)
            .get(`/listing/${listingId}`)
            .set('Authorization', `Bearer ${authToken}`);

          expect(response.statusCode).toBe(200); // 200 OK
        });
```

c. **Retrieve All Listings:** This test retrieves all listings using a GET request to /listing. It expects a successful response with a status code of 200 (OK) and validates the structure of the response, which should include an array of listing objects.

```
        Test retrieving all listings
        it('should retrieve all listings', async () => {
          const response = await request(app)
            .get('/listing')
            .set('Authorization', `Bearer ${authToken}`);

          expect(response.statusCode).toBe(200); // 200 OK
         });
```

d. **Update an Existing Listing:** This test updates an existing listing by sending a PUT request to /listing/{listingId} with partial listing data. It expects a successful response with a status code of 204

```
        Test updating an existing listing
        it('should update an existing listing', async () => {
          const updatedListing = {
          };

          const response = await request(app)
            .put(`/listing/${listingId}`)
            .set('Authorization', `Bearer ${authToken}`)
            .send(updatedListing);

          expect(response.statusCode).toBe(204); // 204 No Content
        });
```

5. **Search and filter Service**

a. **Search for House Listings Based on Keywords:** This test allows users to search for house listings based on keywords such as location, price, and bedrooms. It sends a GET request to /search/houses with query parameters and expects a successful response with a status code of 200. The test validates the structure of the response, which should include a list of house listing objects.

> ***Test searching for house listings based on keywords***
> *it('should allow users to search for house listings based on keywords', async () => {*
>   *const queryParameters = {*
>   *location: 'New York',*
>   *price: 200000, // Example price filter*
>   *bedrooms: 3, // Example bedrooms filter*
>   *};*
>
>   *const response = await request(app)*
>   *.get('/search/houses')*
>   *.query(queryParameters);*
>
>   *expect(response.statusCode).toBe(200); // 200 OK*
>   *});*

b. **Filter House Listings Based on specific filters:** This test allows users to filter house listings based on specific criteria such as location, price range, and the number of bedrooms. It sends a GET request to /search/houses/filter with query parameters for filtering and expects a successful response with a status code of 200. The test validates the structure of the response, which should include a list of house listing objects.

> ***Test filtering house listings based on location, price range, and bedrooms***
> *it('should allow users to filter house listings based on location, price, and bedrooms', async () => {*
>   *const filterParameters = {*
>   *location: 'Los Angeles',*
>   *min_price: 150000, // Example minimum price*
>   *max_price: 300000, // Example maximum price*
>   *min_bedrooms: 2, // Example minimum bedrooms*
>   *max_bedrooms: 4, // Example maximum bedrooms*
>   *min_washrooms: 2, // Example minimum washrooms*
>   *max_washrooms: 3, // Example maximum washrooms*
>   *};*
>
>   *const response = await request(app)*
>   *.get('/search/houses/filter')*
>   *.query(filterParameters);*
>
>   *expect(response.statusCode).toBe(200); // 200 OK*
>   *});*

6. House Viewing Request Service

a. **Create a New View Request:** This test allows a buyer to create a new view request by sending a POST request to /request with the necessary view request data.

> *Test creating a new view request*
> *it('should allow a buyer to create a new view request', async () => {*
>   *const newViewRequest = {*
>   *};*
>
>   *const response = await request(app)*
>   *.post('/request')*
>   *.set('Authorization', `Bearer ${authToken}`)*
>   *.send(newViewRequest);*
>
>   *expect(response.statusCode).toBe(204); // 204 No Content*
>   *requestId = response.body.id;*
> *});*

b. **Update a View Request for Seller Confirmation:** This test updates a view request for seller confirmation by sending a PUT request to /request/{Id} with partial view request data.

> *Test updating a view request for seller confirmation*
> *it('should update a view request for seller confirmation', async () => {*
>   *const updatedRequest = {*
>   *};*
>
>   *const response = await request(app)*
>   *.put(`/request/${requestId}`)*
>   *.set('Authorization', `Bearer ${authToken}`)*
>   *.send(updatedRequest);*
>
>   *expect(response.statusCode).toBe(200); // 200 OK*
>   *});*

c. **Retrieve View Requests Created by a Specific Buyer:** This test retrieves view requests created by a specific buyer by sending a GET request to /requests/buyer/{Id} with the buyer's ID.

> *Test retrieving view requests created by a specific buyer*
> *it('should retrieve view requests created by a specific buyer', async () => {*
>   *const buyerId = 'buyer123'; // Replace with the specific buyer's ID*
>
>   *const response = await request(app)*
>   *.get(`/requests/buyer/${buyerId}`)*
>   *.set('Authorization', `Bearer ${authToken}`);*
>
>   *expect(response.statusCode).toBe(200); // 200 OK*
>   *});*

7. **Chat/Messaging service**

   a. **Send a Message to a User:** This test allows a user to send a message to another user by sending a POST request to /chats with the necessary message data.

   Test sending a message to a user
   ```
   it('should send a message to a user', async () => {
     const newMessage = {
     };

     const response = await request(app)
       .post('/chats')
       .set('Authorization', `Bearer ${authToken}`)
       .send(newMessage);

     expect(response.statusCode).toBe(204);
   });
   ```

   b. **Retrieve List of Conversations for a Specific User:** This test retrieves a list of conversations for a specific user by sending a GET request to /chats/{receiverId} with the receiver's ID.

   Test retrieving list of conversations for a specific user
   ```
   it('should retrieve list of conversations for a specific user', async () => {
     const response = await request(app)
       .get(`/chats/${receiverId}`)
       .set('Authorization', `Bearer ${authToken}`);

     expect(response.statusCode).toBe(200); // 200 OK
   });
   ```

8. **Notification Service**

   a. **Save a Notification in the Database:** This test allows a user to save a notification in the database by sending a POST request to /notifications with the necessary notification data.

   Test saving a notification in the database
   ```
   it('should save a notification in the database', async () => {
     const newNotification = {
       };

     const response = await request(app)
       .post('/notifications')
       .set('Authorization', `Bearer ${authToken}`)
       .send(newNotification);

     expect(response.statusCode).toBe(204); // 204 No Content
     });
   ```

   b. **Subscribe to Notification Channels:** This test allows a user to subscribe to

various notification channels by sending a POST request to /notifications/subscribe with a list of channels to subscribe to.

Test subscribing to notification channels
```
it('should allow users to subscribe to notification channels', async () => {
  const channelsToSubscribe = ['channel1', 'channel2', 'channel3'];

  const response = await request(app)
    .post('/notifications/subscribe')
    .set('Authorization', `Bearer ${authToken}`)
    .send(channelsToSubscribe);

  expect(response.statusCode).toBe(200); // 200 OK
});
```

c. **Unsubscribe from Notification Channels:** This test allows a user to unsubscribe from various notification channels by sending a POST request to /notifications/unsubscribe with a list of channels to unsubscribe from.

Test unsubscribing from notification channels
```
it('should allow users to unsubscribe from notification channels', async () => {
  const channelsToUnsubscribe = ['channel1', 'channel2'];

  const response = await request(app)
    .post('/notifications/unsubscribe')
    .set('Authorization', `Bearer ${authToken}`)
    .send(channelsToUnsubscribe);

  expect(response.statusCode).toBe(200); // 200 OK
});
```

9. Favorites Service

a. **Add a Property to the List of Favorites:** This test allows a user to add a property to their list of favorites by sending a POST request to /favorites/add with the necessary data.

Test adding a property to the list of favorites
```
it('should allow a user to add a property to their list of favorites', async () => {
  const favoriteProperty = {
    userId: userId,
    houseId: 'house123', // Replace with the specific house ID
  };

  const response = await request(app)
    .post('/favorites/add')
    .set('Authorization', `Bearer ${authToken}`)
    .send(favoriteProperty);
```

```
        expect(response.statusCode).toBe(204); // 204 No Content
        });
```

b. **Remove a Property from the List of Favorites:** This test allows a user to remove a property from their list of favorites by sending a DELETE request to /favorites/remove with the necessary data.

Test removing a property from the list of favorites
```
it('should allow a user to remove a property from their list of favorites', async () => {
  const favoriteProperty = {
    userId: userId,
    houseId: 'house123', // Replace with the specific house ID
  };

  const response = await request(app)
    .delete('/favorites/remove')
    .set('Authorization', `Bearer ${authToken}`)
    .send(favoriteProperty);

  expect(response.statusCode).toBe(200); // 200 OK
  });
```

c. **Retrieve List of Favorite Properties for a Specific User:** This test retrieves a list of favorite properties for a specific user by sending a GET request to /favorites/user/{userId} with the user's ID. It expects a successful response with a status code of 200 (OK) and checks if the response includes a JSON array of favorite property objects.

Test retrieving a list of favorite properties for a specific user
```
it('should retrieve a list of favorite properties for a specific user', async () => {
  const response = await request(app)
    .get(`/favorites/user/${userId}`)
    .set('Authorization', `Bearer ${authToken}`);

  expect(response.statusCode).toBe(200); // 200 OK
  });
```

10. Image Storage Service

a. **Upload Property Images:** This test allows users to upload property images by sending a POST request to /images/property with a simulated image file upload.

Test uploading property images
```
it('should allow users to upload property images', async () => {
  const imageFile = Buffer.from('image_data');

  const response = await request(app)
    .post('/images/property')
    .set('Authorization', `Bearer ${authToken}`)
    .attach('image', imageFile, 'property_image.jpg');
```

```
    expect(response.statusCode).toBe(200); // 200 OK
  });
```

b. **Upload User Profile Images:** This test allows users to upload profile images by sending a POST request to /images/user with a simulated image file upload.

```
    Test uploading user profile images
     it('should allow users to upload profile images', async () => {
      // Simulate an image file upload.
      const imageFile = Buffer.from('image_data');

      const response = await request(app)
       .post('/images/user')
       .set('Authorization', `Bearer ${authToken}`)
       .attach('image', imageFile, 'profile_image.jpg');

      expect(response.statusCode).toBe(200); // 200 OK
     });
```

c. **Retrieve Images by Their Filenames:** This test allows users to retrieve images by their filenames by sending a GET request to /images/{imageName} with the image filename.

```
    Test retrieving images by their filenames
     it('should allow retrieving images by their filenames', async () => {
      const imageName = 'property_image.jpg';
      const response = await request(app)
       .get(`/images/${imageName}`)
       .set('Authorization', `Bearer ${authToken}`);

      expect(response.statusCode).toBe(200); // 200 OK
     });
```

d. **Delete Images by Their Filenames:** This test allows users to delete images by their filenames by sending a DELETE request to /images/{imageName} with the image filename.

```
    Test deleting images by their filenames
     it('should allow users to delete images by their filenames', async () => {
      const imageName = 'property_image.jpg';

      const response = await request(app)
       .delete(`/images/${imageName}`)
       .set('Authorization', `Bearer ${authToken}`);

      expect(response.statusCode).toBe(204); // 204 No Content
    });
```

11. User History Service

a. **Record a User's Activity within the Application:** This test allows recording a user's activity within the application by sending a POST request to /user-history with the necessary user history data.

> ***Test recording a user's activity within the application***
> *it('should record a user\'s activity within the application', async () => {*
> * const userHistory = {*
> * };*
>
> * const response = await request(app)*
> *   .post('/user-history')*
> *   .set('Authorization', `Bearer ${authToken}`)*
> *   .send(userHistory);*
>
> * expect(response.statusCode).toBe(204); // 204 No Content*
> *});*

b. **Retrieve a List of Recorded Activities for a Specific User:** This test retrieves a list of recorded activities for a specific user by sending a GET request to /user-history.

> ***Test retrieving a list of recorded activities for a specific user***
> *it('should retrieve a list of recorded activities for a specific user', async () => {*
> * const response = await request(app)*
> *   .get('/user-history')*
> *   .set('Authorization', `Bearer ${authToken}`);*
>
> * expect(response.statusCode).toBe(200); // 200 OK*
> *});*

c. **Allow a User to Remove a History:** This test allows a user to remove a specific history by sending a DELETE request to /user-history/{historyId} with the history ID.

> ***Test allowing a user to remove a history***
> *it('should allow a user to remove a history', async () => {*
> * const response = await request(app)*
> *   .delete(`/user-history/${historyId}`)*
> *   .set('Authorization', `Bearer ${authToken}`);*
>
> * expect(response.statusCode).toBe(200); // 200 OK*
> *});*

12. Review and Rating Service

a. **Allowing a Buyer to Submit a Review and Rating:** This test allows a buyer to submit a review and rating for an owner by sending a POST request to /reviews with the necessary review data.

> **_Test allowing a buyer to submit a review and rating for an owner_**
> ```
>  it('should allow a buyer to submit a review and rating for an owner', async () => {
>    const review = {
>   };
>
>    const response = await request(app)
>     .post('/reviews')
>     .set('Authorization', `Bearer ${authToken}`)
>     .send(review);
>
>    expect(response.statusCode).toBe(204); // 204 No Content
> });
> ```

b. **Retrieve All Reviews for a Specific User:** This test retrieves all reviews for a specific user by sending a GET request to /reviews. It expects a successful response with a status code of 200 (OK) and checks if the response includes an array of review objects.

> **_Test retrieving all reviews for a specific user_**
> ```
>  it('should retrieve all reviews for a specific user', async () => {
>    const response = await request(app)
>     .get('/reviews')
>     .set('Authorization', `Bearer ${authToken}`);
>
>    expect(response.statusCode).toBe(200); // 200 OK
>    });
> ```

c. **Allowing a Reviewer to Update Their Review and Rating:** This test allows a reviewer to update their review and rating by sending a PUT request to /reviews/{reviewId} with the review ID and the updated review details. It expects a successful response with a status code of 200 (OK).

> **_Test allowing a reviewer to update their review and rating_**
> ```
>  it('should allow a reviewer to update their review and rating', async () => {
>    const updatedReview = {
>        };
>
>    const response = await request(app)
>     .put(`/reviews/${reviewId}`)
>     .set('Authorization', `Bearer ${authToken}`)
>     .send(updatedReview);
>
>    expect(response.statusCode).toBe(200); // 200 OK
> });
> ```

d. **Allowing a Reviewer to Delete a Review:** This test allows a reviewer to delete a specific review by sending a DELETE request to /reviews/{reviewId} with the review ID. It expects a successful response with a status code of 200 (OK).

Test allowing a reviewer to delete a review

```
it('should allow a reviewer to delete a review', async () => {
  const response = await request(app)
    .delete(`/reviews/${reviewId}`)
    .set('Authorization', `Bearer ${authToken}`);

  expect(response.statusCode).toBe(200); // 200 OK
});
```

Map Integration:

UML Class Diagram

The UML Class diagram for map and geolocation micro-service is given below.
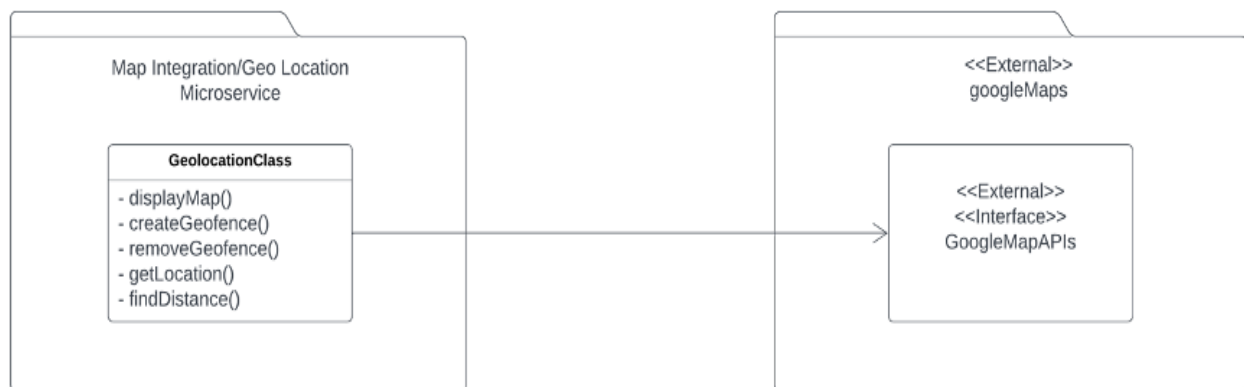
a) UML Class Diagram

The UML Class diagram for 'House Listing micro-service' is given below.

User Class

+ id: String
+ firstName: String
+ lastName: String
+ email: String
+ userAddress: String
+ role: String
+ status: String
+ password: String
+ geoLocation: ref geoLocation object
+ profileImage: String

+ addUser()
+ editUser()
+ deleteUser()
+ searchUser()

Property Class

+ id: String
+ propertyName: String
+ propertyDescription: String
+ price: Float
+ propertyAddress: String
+ area: Float
+ status: String
+ washrooms: Int
+ bedrooms: Int
+user: ref user object
+media: String
+geoLocation: ref geoLocation object
+totalViewRequest: Int
+propertyType: ref propertyType object

+ addProperty()
+ editProperty()
+ deleteProperty()
+ searchProperty()

Property Type Class

+ id: String
+ propertyTypeName: String
+ propertyTypeDescription: String

+ addPropertyType()
+ editPropertyType()
+ deletePropertyType()
+ searchPropertyType()

GeoLocation Class

+ id: String
+ latitiude: Float
+ longitude: Float

+ addGeoLocation()
+ editGeoLocation()
+ deleteGeoLocation()
+ searchGeoLocation()

<<External>> googleMaps

<<External>>
<<Interface>>
GoogleMapAPIs