CSCI 5308 TDD

Bhargav Kanodiya
B00938588

Dhrumil Vimalbhai Gosaliya
B00929836

Gowri Prashanth Kanagaraj
B00942544

Yogish Honnadevipura Gopalakrishna
B00928029

Ramandeep Kaur
B00943241

Gitlab Link-
https://git.cs.dal.ca/courses/2023-summer/csci-5308/group04.git

TABLE OF CONTENT

1. Introduction

1.1. Purpose of The Document:

The purpose of this document is to provide a comprehensive guide to our application, "Accomatch," which facilitates the process of finding suitable accommodations worldwide. The document aims to assist users, especially first-time users, in understanding how to utilize the application effectively. By following the step-by-step instructions outlined in the document, users will be able to navigate through the application seamlessly and effortlessly find their desired accommodation.

The main objectives of this document are as follows:

User Guidance: The document seeks to offer clear and concise instructions on how to use the Accomatch application. It will help users become familiar with the app's features and functionalities, ensuring a smooth user experience.

Accommodation Selection: The document intends to explain how users can leverage the application to select accommodations that align with their specific preferences. Users will have the flexibility to choose factors such as food preferences, gender preferences, and room types.

Communication and Feedback: The document highlights the communication features of Accomatch, such as the chat functionality, review, and rating services. By explaining these features, users can easily connect directly with lease owners, fostering effective communication and enabling them to provide valuable feedback.

Collaborative Environment: Accomatch aims to create a collaborative environment that benefits both lease owners and applicants. This document will elucidate how the application facilitates communication between the two parties, allowing them to specify their criteria and find mutually beneficial arrangements.

Customization: The document will emphasize the application's ability to cater to individual preferences and requirements. Both lease owners and applicants can define their specific criteria, ensuring that the accommodation search process is tailored to their needs.

1.2. Scope of The Document:

This document covers various aspects of the "Accomatch" application, providing a detailed understanding of the system and its functionalities. The scope of the document includes the following aspects:

1. Application Overview: The document provides an overview of the Accomatch application, explaining its purpose and the benefits it offers to users in search of accommodations around the globe.

2. User Guidance: It includes step-by-step instructions on how to use the application, making it a valuable resource for first-time users. The document covers the entire process, from creating an account to searching for accommodations and contacting lease owners.

3. Accommodation Selection: The scope encompasses how users can customize their accommodation preferences using Accomatch. It explains the flexibility in selecting preferences for food, gender, and room type, enabling users to find accommodations that match their specific needs.

4. Communication and Feedback: The document delves into the communication features of the application, such as the chat functionality and the review and rating service. It describes how users can connect directly with lease owners and provide feedback on their experiences.

5. Collaborative Environment: The document covers how the application fosters a collaborative environment between lease owners and applicants. It highlights how both parties can specify their criteria and preferences, facilitating effective communication and negotiation.

6. Customization Options: The scope includes the application's ability to accommodate individual preferences and requirements. It explains how users can define their specific criteria, ensuring a personalized and tailored accommodation search process.

In summary, this document comprehensively covers all user-centric aspects of the Accomatch application, offering guidance on its usage, accommodation selection, communication features, and the creation of a collaborative environment between lease owners and applicants.

1.3. Target Audience:

1. Prospective Users: Individuals interested in using the "Accomatch" application but have not used it before.

2. New Users: Users who have recently signed up for the Accomatch application and need guidance on its features.

3. Applicants: Users actively seeking accommodations for travel, study, work, or temporary living arrangements.

4. Lease Owners: Individuals who have listed their accommodations on Accomatch for potential tenants.

5. Customer Support and Help Desk: Personnel providing assistance and addressing user queries related to the application.

6. Business Development Team: Team members are responsible for promoting and expanding the user base of Accomatch.

7. Decision Makers: Individuals involved in making strategic decisions related to the application's development and direction.

1.4. References:

“Spring boot,” *Spring Boot*. [Online]. Available: https://spring.io/projects/spring-boot. [Accessed: June 08, 2023].

“Java Archive Downloads - Java SE 20,” *Oracle.com*. [Online]. Available: https://www.oracle.com/java/technologies/javase/jdk20-archive-downloads.html. [Accessed: June 08, 2023].

S. Bechtold et al., “JUnit 5 user guide,” *Junit.org*. [Online]. Available: https://junit.org/junit5/docs/current/user-guide/. [Accessed: June 10, 2023].

“Mockito - mockito-core 5.4.0 javadoc,” *Javadoc.io*. [Online]. Available: https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html. [Accessed: June 12, 2023].

“Introduction,” *Netlify.app*. [Online]. Available: https://react-bootstrap.netlify.app/docs/getting-started/introduction. [Accessed: June 17, 2023].

“SweetAlert2,” *Github.io*. [Online]. Available: https://sweetalert2.github.io/. [Accessed: June 20, 2023].

“Flowchart maker & online diagram software,” *Diagrams.net*. [Online]. Available: https://app.diagrams.net/. [Accessed: May 20, 2023].

“Figma: The collaborative interface design tool,” *Figma*. [Online]. Available: https://www.figma.com/. [Accessed:May 18, 2023].

“Add advanced filtering to a React app using React Context API,” *Developer Handbook*. [Online]. Available: https://www.developerhandbook.com/blog/react/advanced-filtering-using-react-context-api/. [Accessed:June 15, 2023].

“How to build a advanced filter component in react?,” *StackOverFlow*. [Online]. Available: https://stackoverflow.com/questions/59350360/how-to-build-a-advanced-filter-component-in-react . [Accessed:June 29, 2023].

“Add login to your Spring Webapp,” *StackOverFlow*. [Online]. Available:https://auth0.com/docs/quickstart/webapp/java-spring-boot/interactive. [Accessed:June 3, 2023].

“Integrate Google OAuth via the API,” *Stych*. [Online]. Available:https://stytch.com/docs/guides/oauth/api. [Accessed:June 24, 2023].

“Creating review system with Ajax, Spring MVC and JavaScript animation,” *Wordpress*. [Online]. Available: https://javawithloveblog.wordpress.com/2017/05/10/creating-review-system-with-ajax-spring-mvc-and-javascript-animation/. [Accessed:June 25, 2023].

“Using WebSocket to build an interactive web application,” *Spring*. [Online]. Available: https://spring.io/guides/gs/messaging-stomp-websocket/. [Accessed:July 7, 2023].

“Real Time Application using WebSocket (Spring Boot (JAVA), React.Js, Flutter),” *Medium*. [Online]. Available:https://umes4ever.medium.com/real-time-application-using-websocket-spring-boot-java-react-js-flutter-eb87fe95f94f. [Accessed:June 17, 2023].

“IntelliJ IDEA – the Leading Java and Kotlin IDE,” *Jetbrains*. [Online]. Available: https://www.jetbrains.com/idea/. [Accessed:May 20, 2023].

“Netlify Connect,” *Netlify*. [Online]. Available: https://www.netlify.com/?attr=homepage-modal. [Accessed:June 12, 2023].

“Render,” *Render*. [Online]. Available:https://render.com/. [Accessed:June 10, 2023].

“GitLab CI/CD,” *GitLab*. [Online]. Available:https://docs.gitlab.com/ee/ci/. [Accessed:June 5, 2023].

“Jira Software boards tutorials,” *Jira Software*. [Online]. Available:https://www.atlassian.com/software/jira/guides/boards/tutorials#configure-columns. [Accessed:May 17, 2023].

“Apartment image 1,” *images1 apartments*. [Online]. Available:https://images1.apartments.com/i2/991otg0L0cjJomys4aNphYO9A-iDOaMaGC1NHsWzYSQ/111/baker-arms-wexford-apartments-dartmouth-ns-primary-photo.jpg. [Accessed:June 1, 2023].

“Apartment image 2,” *cbc*. [Online]. Available: https://i.cbc.ca/1.5993148.1618850303!/fileImage/httpImage/image.jpg_gen/derivatives/16x9_78

0/st-f-x-taking-extra-precautions-as-students-leave-campus-for-school-year-image-1.jpg. [Accessed:June 7, 2023].

“Apartment image 3,” *capreit*. [Online]. Available: https://www.capreit.ca/wp-content/uploads/2021/09/capreit-ns-halifax-1333southpark-photo-5.jpg . [Accessed:June 12, 2023].

“Apartment image 4,” *wikimedia*. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/thumb/e/e6/Victoria-park-kitchener-lake.jpg/800px-Victoria-park-kitchener-lake.jpg. [Accessed:July 10, 2023].

2. System Overview

2.1 System Description and Context:

The "Accomatch" system is an online application designed to facilitate the process of finding accommodations worldwide. It serves as a platform connecting individuals in need of housing with lease owners offering their properties for rent.

At its core, Accomatch is a web and mobile-based application that empowers users to discover suitable accommodations seamlessly.Users can create accounts on the platform, providing essential information to help customize their accommodation search.

Accomatch implements a robust filtering and matching algorithm that enables users to narrow down their search results based on factors such as location, price range, amenities, room types, food preferences, and gender-specific accommodation.

The Accomatch system operates within the context of the global accommodation market, catering to individuals seeking housing solutions for various purposes. The application targets a broad audience, including travelers, students, professionals, and anyone in need of temporary or long-term housing.

2.2 Design Goals and Constraints:

Design Goals:

1. User-Friendly Interface: The primary design goal is to create a user-friendly interface that is intuitive and easy to navigate. Users should be able to use Accomatch without confusion, regardless of their technical expertise.

2. Efficient Accommodation Matching: The design aims to implement a robust and efficient accommodation matching algorithm. It should accurately match users' preferences with suitable accommodation options, providing relevant and personalized search results.

3. Seamless Communication: Accomatch should facilitate seamless communication between applicants and lease owners. The chat feature should be user-friendly and reliable, allowing real-time interactions to discuss accommodation details and arrangements.

4. Customization and Flexibility: The application design should allow users to customize their search criteria based on various factors, such as location, budget, amenities, and preferences for food, gender, and room type. The design should ensure flexibility to cater to diverse user requirements.

5. Data Security and Privacy: A key design goal is to prioritize data security and privacy. The application must implement robust encryption and authentication mechanisms to safeguard user data, financial transactions, and personal information.

6. Scalability and Performance: The design should be scalable to accommodate a growing user base and an increasing number of accommodation listings. The application should maintain high performance, ensuring quick response times even during peak usage.

Design Constraints:

1. Technical Limitations: The design must work within the constraints of the available technology stack and infrastructure. It should consider the limitations of hardware, software, and network resources.

2. Regulatory Compliance: Accomatch must comply with relevant legal and regulatory requirements concerning user data, financial transactions, and accommodation listings in various countries or regions.

3. Cross-Platform Compatibility: The design should address cross-platform compatibility, making Accomatch accessible and functional on various devices and operating systems.

4. User Feedback and Adaptation: The design should be open to user feedback and continuous improvement, adapting to user needs and preferences over time.

5. Quality Assurance: The design should undergo rigorous testing to identify and resolve any bugs, usability issues, or vulnerabilities.

By setting clear design goals and working within these constraints, Accomatch can provide a seamless, secure, and user-centric experience, establishing itself as a reliable platform for finding accommodations worldwide.

2.3 Stakeholder Identification:

Stakeholder identification is crucial to ensuring that the design of Accomatch meets the needs and expectations of all parties involved. Here is a list of people or organizations who have an interest in the design or will be affected by it:

1. Users (Applicants and Lease Owners): The primary stakeholders are the users of Accomatch, including individuals searching for accommodations (applicants) and property owners offering accommodations for rent (lease owners). Their experience and satisfaction with the platform are of utmost importance.

2. Development Team: The team responsible for designing, developing, and maintaining the Accomatch application. They are crucial stakeholders in ensuring that the design is technically feasible, scalable, and meets the desired requirements.

3. Product Managers and Business Owners: Product managers and business owners within the Accomatch organization have a vested interest in the design as it directly impacts the success and profitability of the platform.

4. Customer Support Team: The customer support team plays a significant role in assisting users and addressing their queries or concerns related to the design and functionality of the application.

5. Marketing Team: The marketing team needs to understand the design to effectively promote the application's features and benefits to potential users.

6. Data Security and Compliance Team: Stakeholders responsible for data security and regulatory compliance need to ensure that the design adheres to relevant data protection laws and industry standards.

7. Financial Team: The financial team has an interest in the design's cost implications, budget allocation, and financial feasibility.

8. Third-Party Service Providers: If Accomatch integrates with third-party services, the providers of those services become stakeholders in the design process.

Effectively addressing the needs and expectations of these stakeholders throughout the design process is essential for creating a successful and well-received Accomatch platform. Regular communication, feedback gathering, and collaboration with stakeholders can lead to a design that caters to a diverse range of interests and ensures a positive overall user experience.

3. Architecture and Component Design

3.1 High-Level Architecture:

The system is designed to facilitate the process of finding accommodation between two types of users: Leaseholders and HouseSeekers. Leaseholders are individuals who have an available property to offer for rent, while HouseSeekers are those looking for rental accommodations. The system also includes features for chat communication, rating, and preferences for both types of users.

Key Components:

User Table: The central table that stores information about all users of the system. Each user has a unique user_id, and the table contains various user details such as email, name, password, age, gender, mobile, address, and two flags (is_admin and is_leaseholder) to differentiate between regular users and administrators, and between Leaseholders and HouseSeekers.

Leaseholder Ads Table: This table stores advertisements created by Leaseholders. It contains information about the properties they offer for rent, such as title, subtitle, address, size, room type, rent, start date, age preferences, and verification status.

HouseSeeker Ads Table: This table stores advertisements created by HouseSeekers. It contains information about their accommodation preferences, such as location_city, room type, start date, and other preferences.

Leaseholder Food Preferences Table: This table holds the food preferences of Leaseholders. It is linked to the Leaseholder Ads table using the application_id.

Leaseholder Images Table: This table stores images associated with the Leaseholder Ads. It is linked to the Leaseholder Ads table using the application_id.

Leaseholder Gender Preferences Table: This table holds the gender preferences of Leaseholders. It is linked to the Leaseholder Ads table using the application_id.

HouseSeeker Food Preferences Table: This table holds the food preferences of HouseSeekers. It is linked to the HouseSeeker Ads table using the application_id.

HouseSeeker Gender Preferences Table: This table holds the gender preferences of HouseSeekers. It is linked to the HouseSeeker Ads table using the application_id.

Leaseholder Applicant Table: This table stores the applications made by HouseSeekers on Leaseholder Ads. It contains information about the status of the application, the associated room_id if applicable, and links to the Leaseholder Ads and User tables using application_id and user_id, respectively.

HouseSeeker Applicant Table: This table stores the applications made by Leaseholders on HouseSeeker Ads. It contains information about the status of the application, the associated room_id if applicable, and links to the HouseSeeker Ads and User tables using application_id and user_id, respectively.

Chat Table: This table keeps track of the messages exchanged between users in individual chat rooms. It includes a unique message_id, the room_id where the chat occurred, the user_id of the sender, the message content, and the timestamp.

Room Table: This table represents individual chat rooms where users can communicate with each other. It contains user_1_id and user_2_id fields, which reference the user_id from the User table to identify the participants in each chat room.

Rating Table: This table stores ratings and reviews given by users for a particular Leaseholder Ad. It contains information about the user who provided the rating, the Leaseholder Ad associated with the rating, the rating value, and the review text.

Leaseholder Current Residents Table: This table maintains the list of current residents for each Leaseholder Ad. It is linked to the Leaseholder Ads and User tables using application_id and user_id, respectively.

3.2 Component or Module Description:

<u>User Module:</u>

Purpose: Handles user-related endpoints, such as user registration (signup), login, retrieving user information, and password reset functionalities.
Inputs: UserModel objects in the request body for signup, login, and password reset.
Outputs: Strings for signup, login, and password reset operations. A map will be returned for the Login method and a Usermodel Instance will be returned for getting the user information.
Interaction: The controller interacts with the UserService and MailSenderClass to perform user-related operations and send emails for password reset.

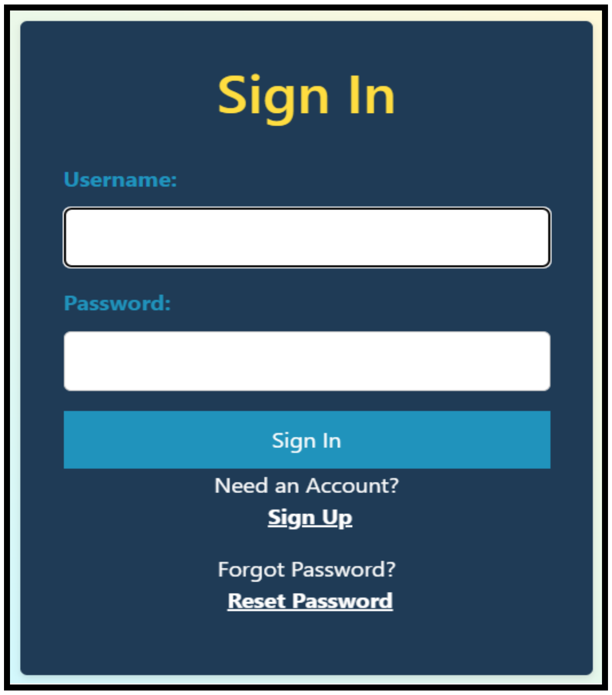


Fig 3.2.1: Sign In Page

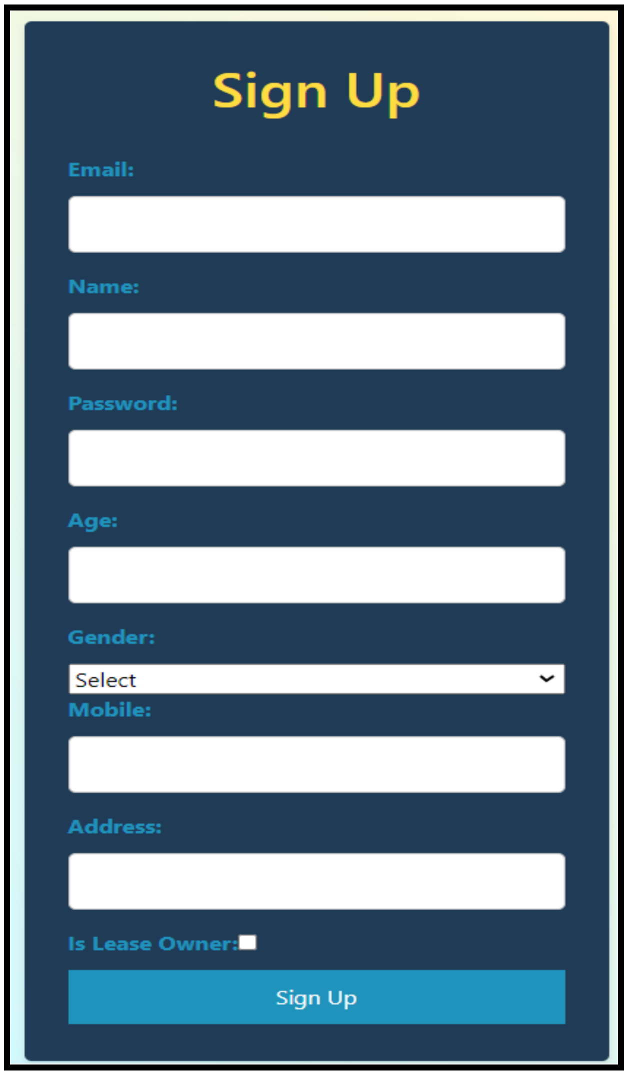
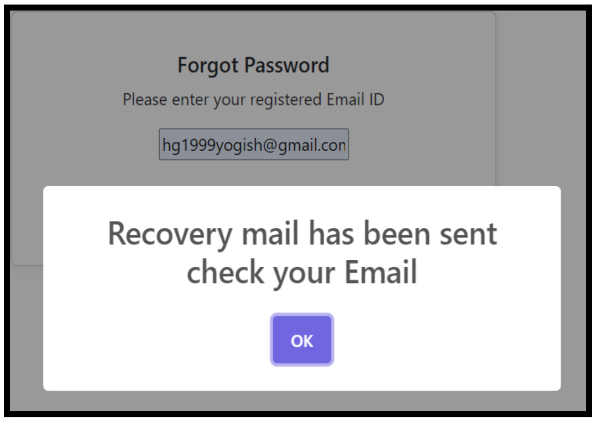


Fig 3.2.2 : Sign Up page

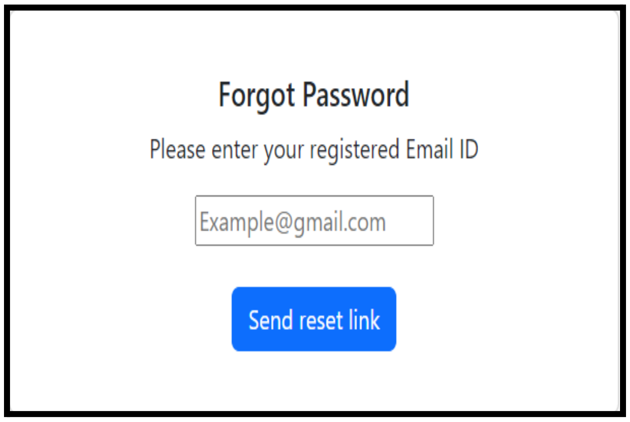


Please enter your registered Email ID

hg1999yogish@gmail.con

Recovery mail has been sent check your Email

Fig 3.2.3 : Mail sent Confirmation page



Please enter your registered Email ID

Fig 3.2.4 : Forgot Password Page

Review Module:

Purpose: Manages endpoints related to posting and retrieving reviews for user applications.
Inputs: Review objects in the request body for creating reviews.
Outputs: None for creating reviews (void method).
Interaction: The controller interacts with the ReviewService to handle review-related operations.

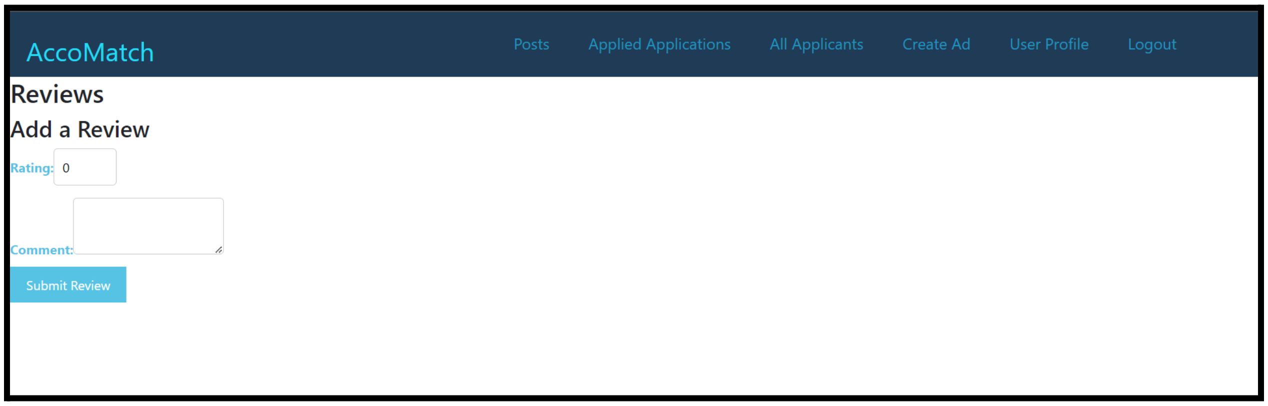
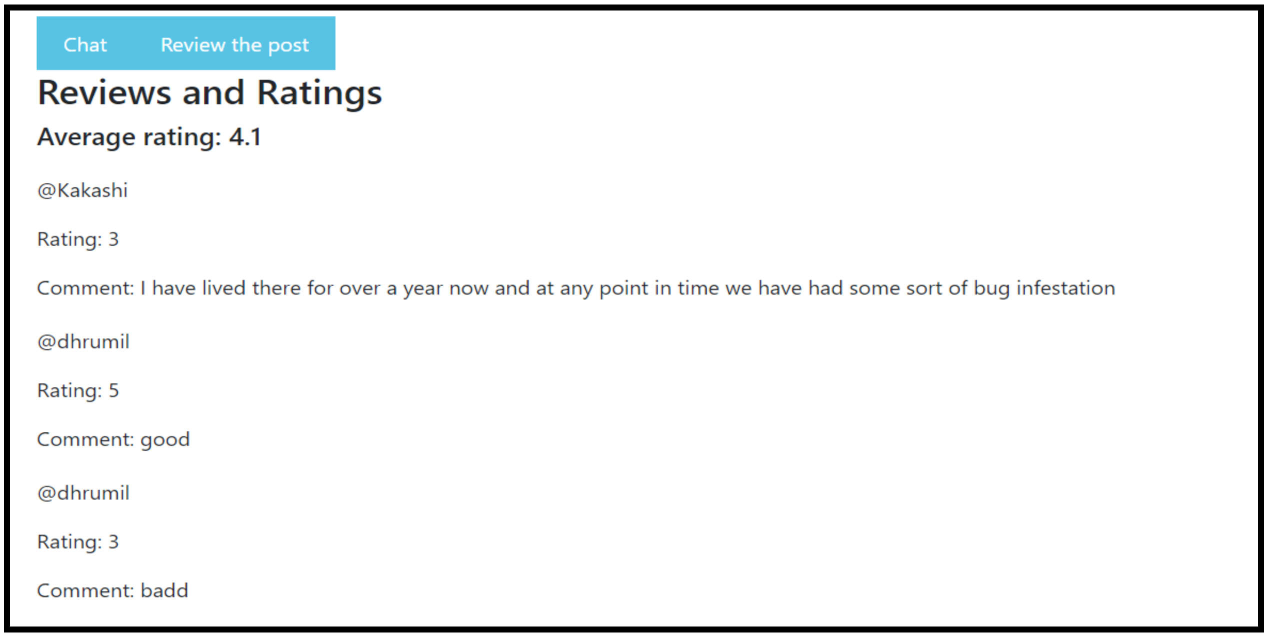


Fig 3.2.5 : Review and rating Upload Page

Post Ratings View Module:

Purpose: Provides endpoints to retrieve all reviews and average ratings for a specific application post.
Inputs: Integer values representing the application ID.
Outputs: Lists of Review and Ratings objects containing review details and average ratings.
Interaction: The controller interacts with the ReviewServiceImplementation to fetch review and rating data.



Reviews and Ratings

Average rating: 4.1

@Kakashi

Rating: 3

Comment: I have lived there for over a year now and at any point in time we have had some sort of bug infestation

@dhrumil

Rating: 5

Comment: good

@dhrumil

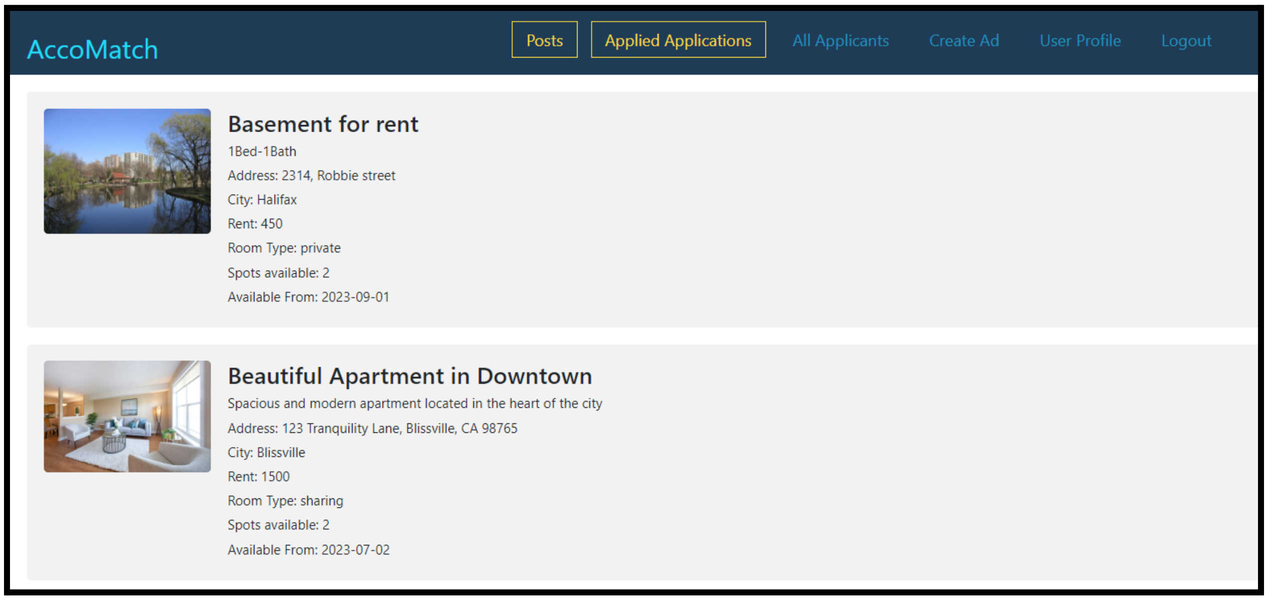
Rating: 3

Comment: badd

Fig 3.2.6 : Review and Rating Page

Lease Posts Logged In Module:

Purpose: Manages endpoints to retrieve a list of posts for a logged-in leaseholder applicant.
Inputs: Integer value representing the user ID of the leaseholder applicant.
Outputs: Lists of Posts objects representing the logged-in applicant's posts.
Interaction: The controller interacts with the LeasePostsLoggedinService to fetch posts for the logged-in leaseholder applicant.



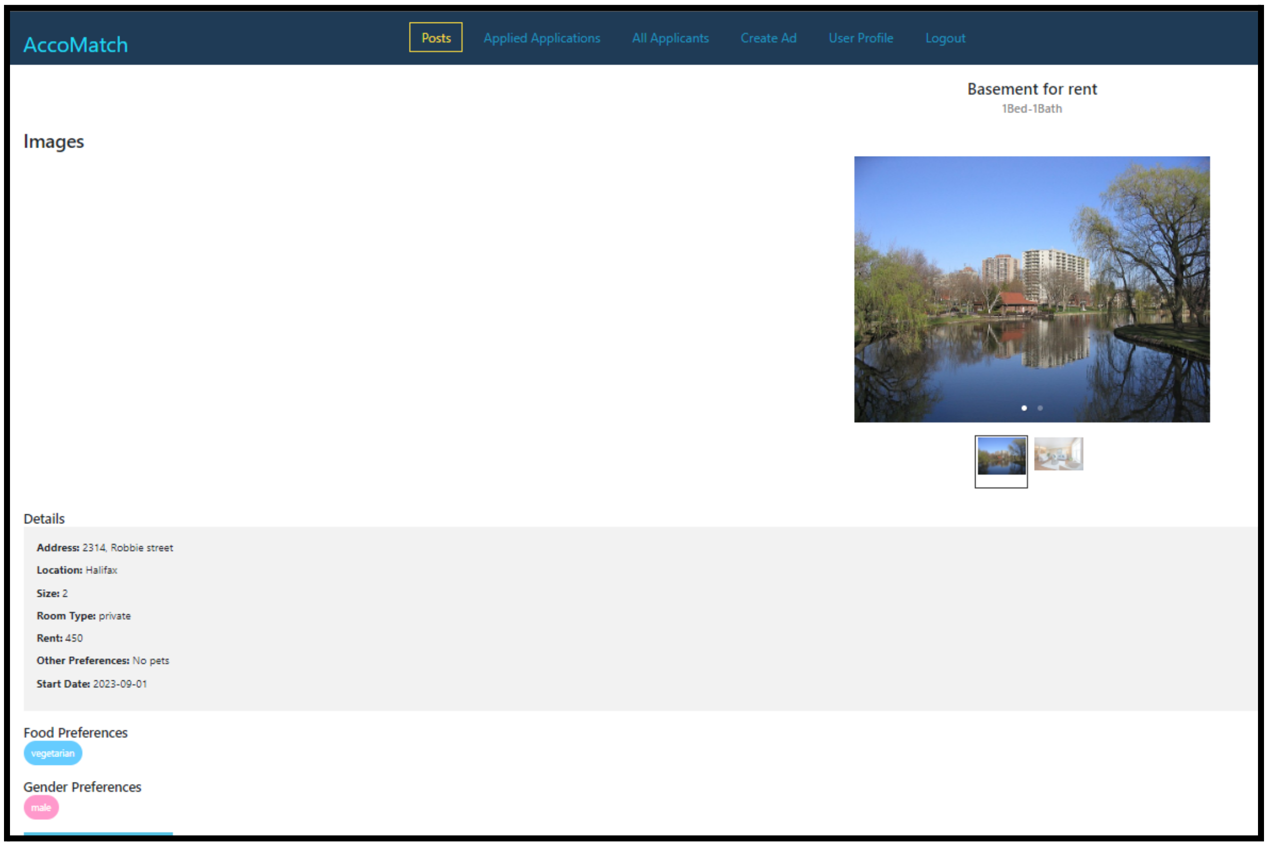
Basement for rent
1Bed-1Bath
Address: 2314, Robbie street
City: Halifax
Rent: 450
Room Type: private
Spots available: 2
Available From: 2023-09-01

Beautiful Apartment in Downtown
Spacious and modern apartment located in the heart of the city
Address: 123 Tranquility Lane, Blissville, CA 98765
City: Blissville
Rent: 1500
Room Type: sharing
Spots available: 2
Available From: 2023-07-02

Fig 3.2.7 : Applied Applicants Page

LeaseHolder Dashboard Controller:

Purpose: Handles endpoints for the leaseholder dashboard, such as retrieving posts, post details, images, food preferences, and gender preferences.
Inputs: Integer values representing application IDs for post details, images, food preferences, and gender preferences.
Outputs: Lists of Posts objects for posts, images, food preferences, and gender preferences, and single Posts objects for post details.
Interaction: The controller interacts with the LeaseHolderDashboardInterface to fetch data related to the leaseholder dashboard.



Basement for rent
1Bed-1Bath

Images

Details

Address: 2314, Robbie street
Location: Halifax
Size: 2
Room Type: private
Rent: 450
Other Preferences: No pets
Start Date: 2023-09-01

Food Preferences
vegetarian

Gender Preferences
male

Fig 3.2.8 :Post details Page

HouseSeeker Controller:

Purpose: Provides an endpoint to retrieve a list of all applicant posts.
Inputs: None (GET request).
Outputs: List of HouseSeekerModel objects representing the applicant posts.
Interaction: The controller interacts with the HouseSeekerService to fetch all applicant posts.

__Create Application Controller:__

Purpose: Handles endpoints for creating applications for house seekers.
Inputs: JSON object with application details in the request body.
Outputs: String indicating the status of the application creation process.
Interaction: The controller interacts with the CreateApplicationFactory to create applications.

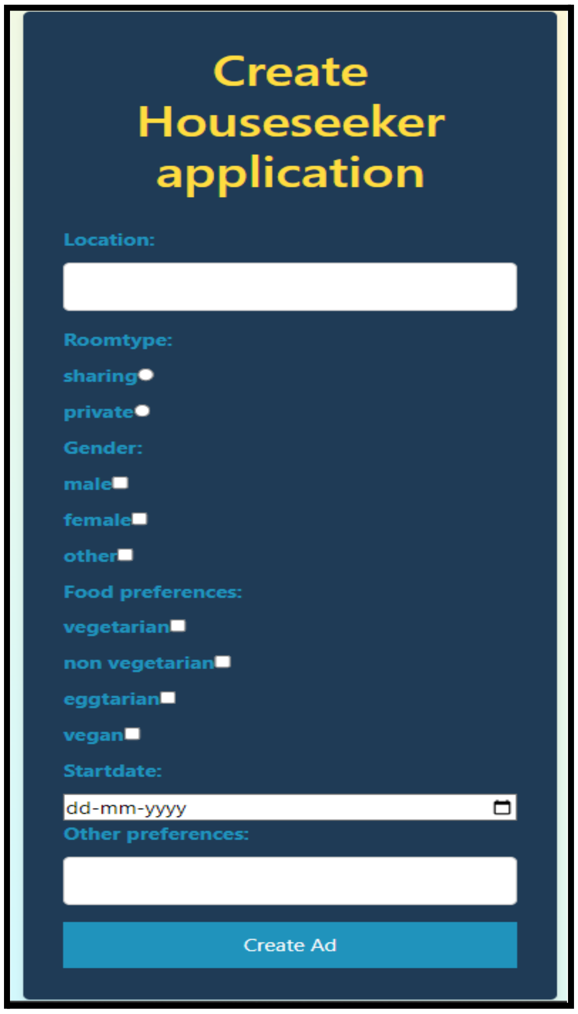


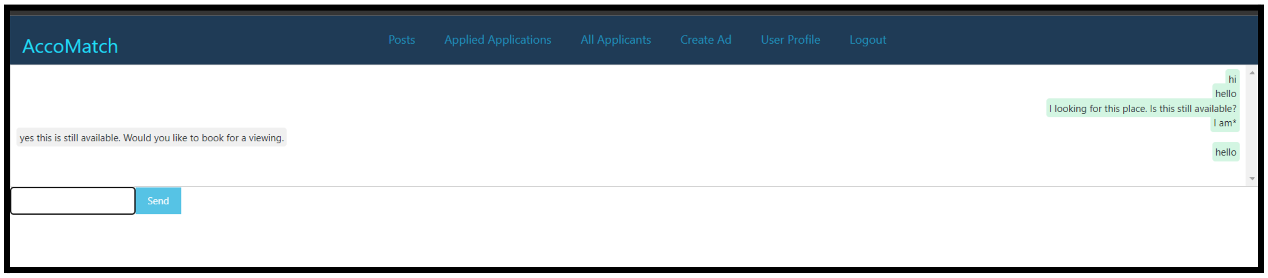
Fig 3.2.9: Create Ad Page

Chat Room Controller:

Purpose: Manages endpoints related to chat rooms, such as retrieving room IDs based on user and application IDs.
Inputs: JSON object with user and application IDs in the request body.
Outputs: Integer representing the room ID.
Interaction: The controller interacts with the ChatRoomService to handle chat room operations.

ChatController:

Purpose: Handles endpoints for sending and retrieving chat messages in a chat room.
Inputs: JSON object with user ID, room ID, and message in the request body for sending messages. Room ID in the path variable for fetching messages.
Outputs: String indicating the success or error of sending messages, and ArrayList of ChatMessageModel objects containing chat messages.
Interaction: The controller interacts with the ChatService to send and retrieve chat messages.

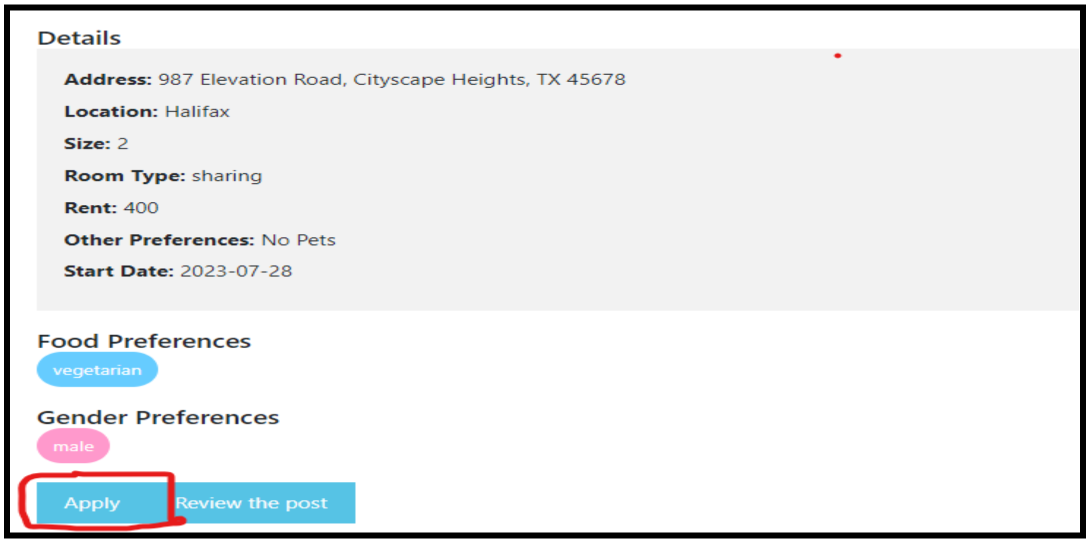


hi
hello
I looking for this place. Is this still available?
I am*
yes this is still available. Would you like to book for a viewing.
hello

Fig 3.2.10: Chat Page

ApplyonPostController:

Purpose: Manages endpoints for leaseholder applicants to apply on posts and check if they have already applied.
Inputs: LeaseHolderApplicantModel object in the request body for applying on posts. Map containing user ID and application ID for checking if the user has already applied.
Outputs: String indicating the status of the application for applying, and boolean for the application check.
Interaction: The controller interacts with the ApplyonPostService to apply on posts and check application status.


Details

Address: 987 Elevation Road, Cityscape Heights, TX 45678

Location: Halifax

Size: 2

Room Type: sharing

Rent: 400

Other Preferences: No Pets

Start Date: 2023-07-28

Food Preferences

vegetarian

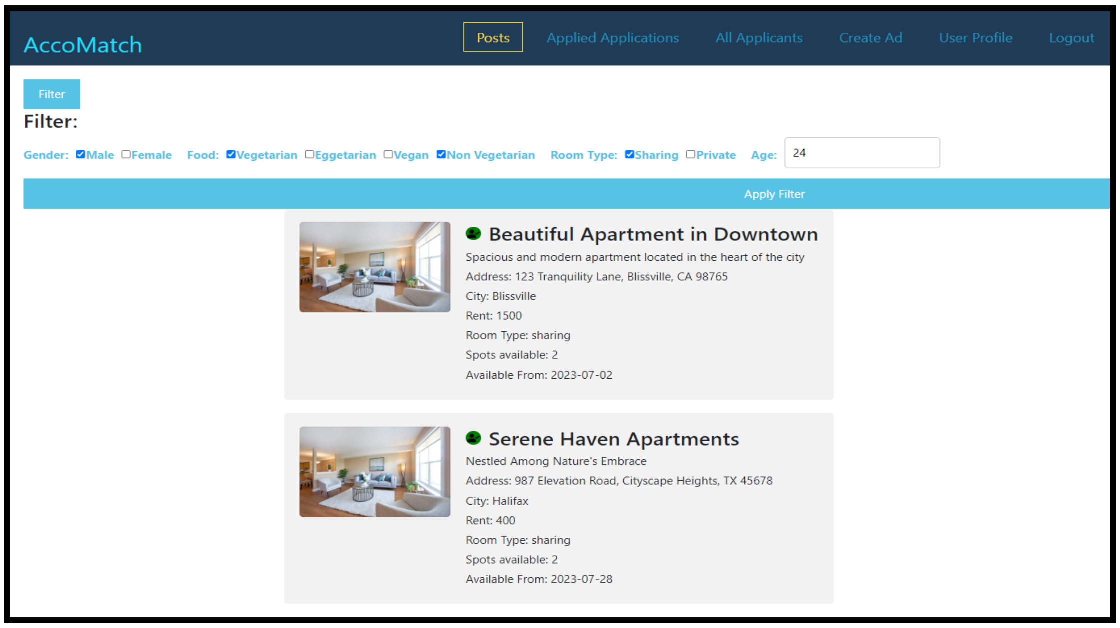
Gender Preferences

male

Fig 3.2.11: Apply to post

ApplicantPostFiltering:

Purpose: Provides an endpoint for filtering posts based on applicant preferences.
Inputs: Map containing applicant preferences, such as gender preference, food preference, age, and room type, in the request body.
Outputs: List of Posts objects representing filtered posts.
Interaction: The controller interacts with the ApplicantPostFilterService to filter posts based on applicant preferences.



Filter:

Gender: ☑Male ☐Female Food: ☑Vegetarian ☐Eggetarian ☐Vegan ☑Non Vegetarian Room Type: ☑Sharing ☐Private Age: 24

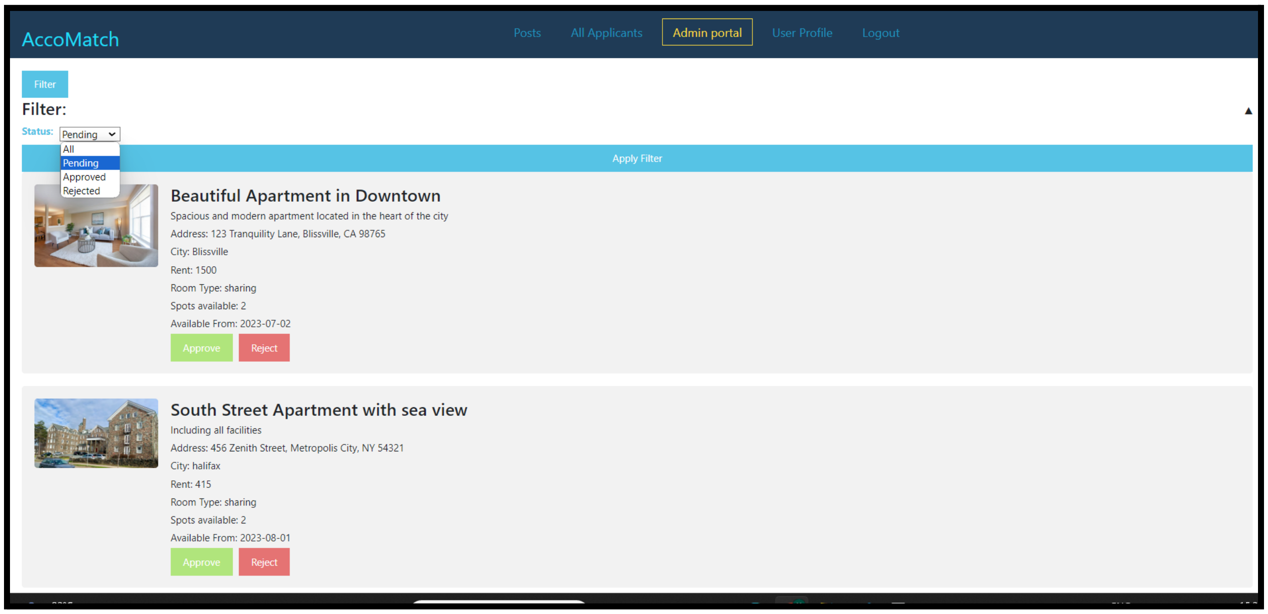
Beautiful Apartment in Downtown
Spacious and modern apartment located in the heart of the city
Address: 123 Tranquility Lane, Blissville, CA 98765
City: Blissville
Rent: 1500
Room Type: sharing
Spots available: 2
Available From: 2023-07-02

Serene Haven Apartments
Nestled Among Nature's Embrace
Address: 987 Elevation Road, Cityscape Heights, TX 45678
City: Halifax
Rent: 400
Room Type: sharing
Spots available: 2
Available From: 2023-07-28

Fig: 3.2.12: Application Filtering

AdminController:

Purpose: Handles endpoints for administrative actions, such as verifying individual ads and fetching the list of posts for admin based on status.
Inputs: Posts objects in the request body for verifying ads. Map containing the status for fetching posts.
Outputs: String indicating the success or error of verifying ads, and List of Posts objects representing the list of posts.
Interaction: The controller interacts with the AdminInterface and LeaseHolderDashboardInterface to handle administrative actions and fetch the list of posts.



Beautiful Apartment in Downtown
Spacious and modern apartment located in the heart of the city
Address: 123 Tranquility Lane, Blissville, CA 98765
City: Blissville
Rent: 1500
Room Type: sharing
Spots available: 2
Available From: 2023-07-02

South Street Apartment with sea view
Including all facilities
Address: 456 Zenith Street, Metropolis City, NY 54321
City: halifax
Rent: 415
Room Type: sharing
Spots available: 2
Available From: 2023-08-01

Fig 3.2.13: Admin portal

3.3 Flow Diagram:

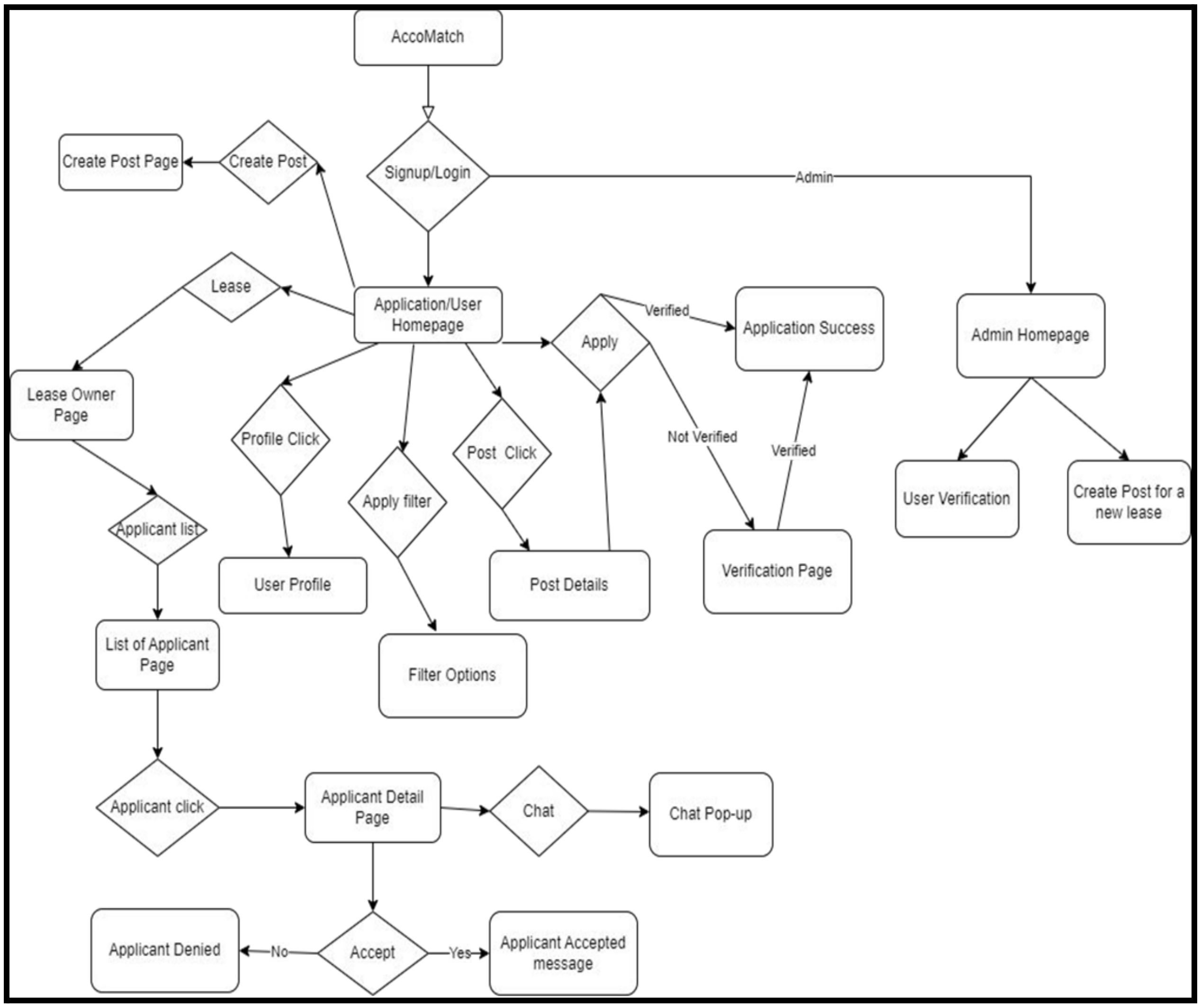


Fig 3.3.1: Flowchart of the Application

4. Data Design

4.1 Data Models or ER Diagrams:

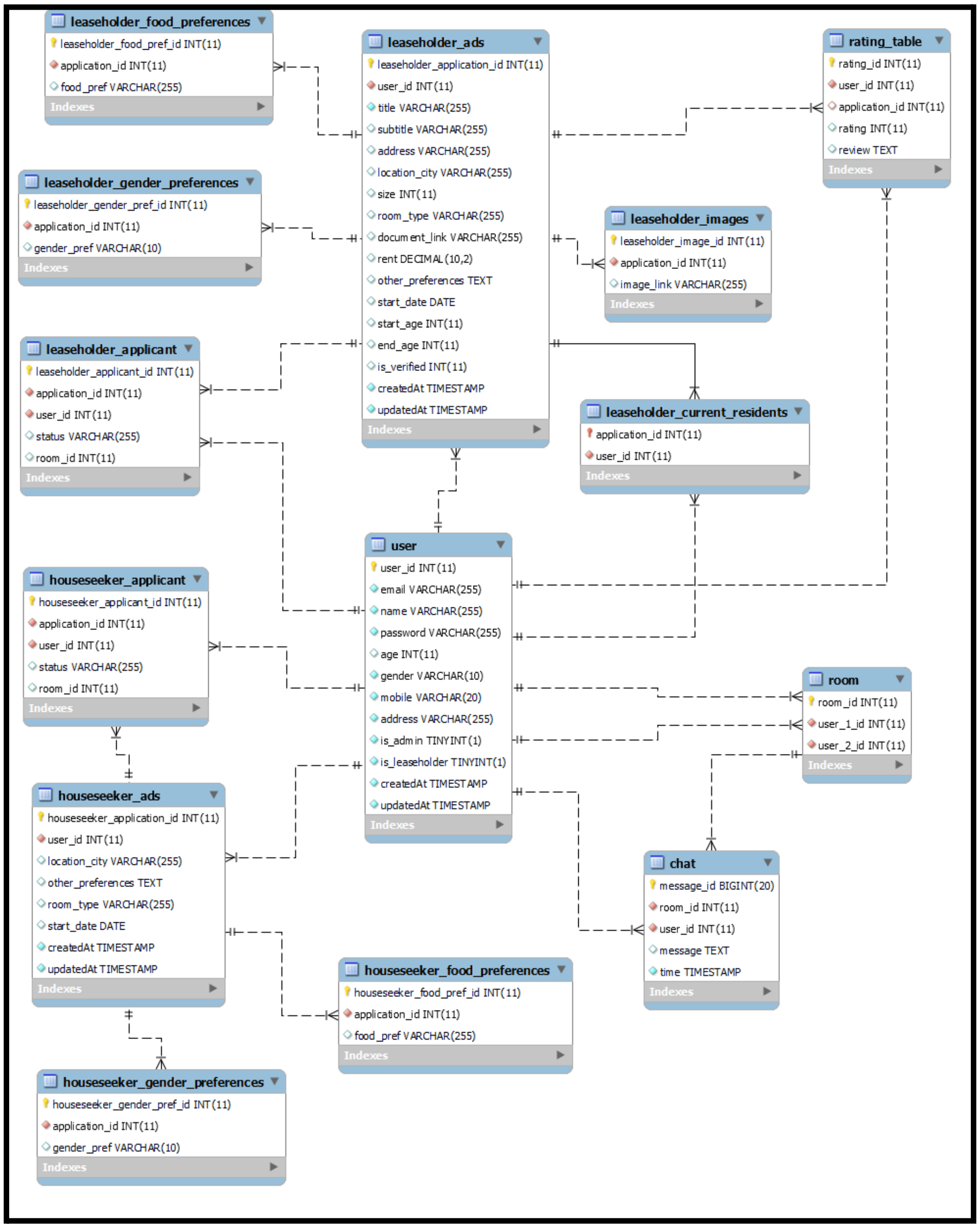


Fig 4.1.1: Entity Relation Diagram

4.2 Database Schema:

The database system contains several tables designed to store information related to users, ads for leaseholders and house seekers, preferences, applicants, chat messages, ratings, and current residents. Below is the organization of data in the system's database:

User Table:
Stores information about users who have registered on the platform.
Fields: user_id (Primary Key), email, name, password, age, gender, mobile, address, is_admin, is_leaseholder, createdAt, updatedAt.

Leaseholder Ads Table:
Contains ads posted by leaseholders who are looking for house seekers or roommates.
Fields: leaseholder_application_id (Primary Key), user_id (Foreign Key to User Table), title, subtitle, address, location_city, size, room_type, document, rent, other_preferences, start_date, start_age, end_age, is_verified, createdAt, updatedAt.

HouseSeeker Ads Table:
Stores ads posted by house seekers who are looking for accommodation.
Fields: houseseeker_application_id (Primary Key), user_id (Foreign Key to User Table), location_city, other_preferences, room_type, start_date, createdAt, updatedAt.

Leaseholder Food Preferences Table:
Records food preferences of leaseholders for potential house seekers.
Fields: leaseholder_food_pref_id (Primary Key), application_id (Foreign Key to Leaseholder Ads Table), food_pref.

Leaseholder Images Table:
Stores images related to leaseholder ads.
Fields: leaseholder_image_id (Primary Key), application_id (Foreign Key to Leaseholder Ads Table), image_link.

Leaseholder Gender Preferences Table:
Captures gender preferences of leaseholders for potential house seekers.
Fields: leaseholder_gender_pref_id (Primary Key), application_id (Foreign Key to Leaseholder Ads Table), gender_pref.

HouseSeeker Food Preferences Table:
Records food preferences of house seekers for potential leaseholders.
Fields: houseseeker_food_pref_id (Primary Key), application_id (Foreign Key to HouseSeeker Ads Table), food_pref.

HouseSeeker Gender Preferences Table:
Captures gender preferences of house seekers for potential leaseholders.
Fields: houseseeker_gender_pref_id (Primary Key), application_id (Foreign Key to HouseSeeker Ads Table), gender_pref.

Leaseholder Applicant Table:
Tracks applications made by users for leaseholder ads.
Fields: leaseholder_applicant_id (Primary Key), application_id (Foreign Key to Leaseholder Ads Table), user_id (Foreign Key to User Table), status, room_id.

HouseSeeker Applicant Table:
Tracks applications made by users for house seeker ads.
Fields: houseseeker_applicant_id (Primary Key), application_id (Foreign Key to HouseSeeker Ads Table), user_id (Foreign Key to User Table), status, room_id.

Chat Table:
Stores chat messages exchanged between users.
Fields: message_id (Primary Key), room_id (Foreign Key to Room Table), user_id (Foreign Key to User Table), message, time.

Room Table:
Records rooms in which users engage in chat conversations.
Fields: room_id (Primary Key), user_1_id (Foreign Key to User Table), user_2_id (Foreign Key to User Table).

Rating Table:
Stores ratings and reviews given by users to leaseholder ads.
Fields: rating_id (Primary Key), user_id (Foreign Key to User Table), application_id (Foreign Key to Leaseholder Ads Table), rating, review.

Leaseholder Current Residents Table:
Keeps track of current residents for leaseholder ads.
Fields: application_id (Primary Key, Foreign Key to Leaseholder Ads Table), user_id (Foreign Key to User Table).

4.3 Data Dictionary:

Defining each data element in the database and explain its purpose:

User Table:
user_id: An auto-incrementing unique identifier for each user in the system.
email: The email address of the user, used as a unique key and for communication.
name: The name of the user.
password: The password of the user for authentication (Note: Storing passwords directly in a database is not secure. Best practices involve hashing and salting passwords before storage).
age: The age of the user.
gender: The gender of the user.
mobile: The mobile number of the user.
address: The address of the user.
is_admin: A flag (1 or 0) indicating if the user is an admin.
is_leaseholder: A flag (1 or 0) indicating if the user is a leaseholder (ad poster).
createdAt: Timestamp for when the user record was created.
updatedAt: Timestamp for when the user record was last updated.

Leaseholder Ads Table:
leaseholder_application_id: An auto-incrementing unique identifier for each leaseholder ad.
user_id: Foreign key referencing the user who posted the leaseholder ad.
title: The title of the ad.
subtitle: A subtitle or additional information for the ad.
address: The address of the property being advertised.
location_city: The city of the property being advertised.
size: The size of the property (e.g., square feet).
room_type: The type of room being offered (e.g., single, shared).
document: A document related to the property (e.g., lease agreement).
rent: The rental amount for the property.
other_preferences: Additional preferences or details about the property.
start_date: The date from which the lease begins.
start_age: The minimum age preference for applicants.
end_age: The maximum age preference for applicants.
is_verified: A flag (1 or 0) indicating if the ad is verified.
createdAt: Timestamp for when the ad record was created.
updatedAt: Timestamp for when the ad record was last updated.

HouseSeeker Ads Table:
houseseeker_application_id: An auto-incrementing unique identifier for each house seeker ad.
user_id: Foreign key referencing the user who posted the house seeker ad.
location_city: The city preferred by the house seeker for accommodation.
other_preferences: Additional preferences or details about the accommodation sought.
room_type: The type of room sought (e.g., single, shared).
start_date: The date from which the house seeker needs the accommodation.

createdAt: Timestamp for when the ad record was created.
updatedAt: Timestamp for when the ad record was last updated.

Leaseholder Food Preferences Table:
leaseholder_food_pref_id: An auto-incrementing unique identifier for each leaseholder's food preference entry.
application_id: Foreign key referencing the leaseholder ad to which the food preference belongs.
food_pref: The food preferences of the leaseholder (e.g., vegetarian, non-vegetarian).

Leaseholder Images Table:
leaseholder_image_id: An auto-incrementing unique identifier for each leaseholder image entry.
application_id: Foreign key referencing the leaseholder ad to which the image belongs.
image_link: The link or path to the image related to the leaseholder ad.

Leaseholder Gender Preferences Table:
leaseholder_gender_pref_id: An auto-incrementing unique identifier for each leaseholder's gender preference entry.
application_id: Foreign key referencing the leaseholder ad to which the gender preference belongs.
gender_pref: The gender preferences of the leaseholder for potential house seekers (e.g., male, female).

HouseSeeker Food Preferences Table:
houseseeker_food_pref_id: An auto-incrementing unique identifier for each house seeker's food preference entry.
application_id: Foreign key referencing the house seeker ad to which the food preference belongs.
food_pref: The food preferences of the house seeker (e.g., vegetarian, non-vegetarian).

HouseSeeker Gender Preferences Table:
houseseeker_gender_pref_id: An auto-incrementing unique identifier for each house seeker's gender preference entry.
application_id: Foreign key referencing the house seeker ad to which the gender preference belongs.
gender_pref: The gender preferences of the house seeker for potential leaseholders (e.g., male, female).

Leaseholder Applicant Table:
leaseholder_applicant_id: An auto-incrementing unique identifier for each leaseholder applicant entry.
application_id: Foreign key referencing the leaseholder ad to which the applicant applies.
user_id: Foreign key referencing the user who applied.
status: The status of the application (e.g., pending, accepted, rejected).
room_id: Foreign key referencing the room (if assigned) for the applicant.

HouseSeeker Applicant Table:
houseseeker_applicant_id: An auto-incrementing unique identifier for each house seeker applicant entry.
application_id: Foreign key referencing the house seeker ad to which the applicant applies.
user_id: Foreign key referencing the user who applied.
status: The status of the application (e.g., pending, accepted, rejected).
room_id: Foreign key referencing the room (if assigned) for the applicant.

Chat Table:
message_id: An auto-incrementing unique identifier for each chat message entry.
room_id: Foreign key referencing the room in which the message was sent.
user_id: Foreign key referencing the user who sent the message.
message: The content of the chat message.
time: The timestamp of when the message was sent.

Room Table:
room_id: An auto-incrementing unique identifier for each chat room entry.
user_1_id: Foreign key referencing one user in the chat room.
user_2_id: Foreign key referencing the other user in the chat room.

Rating Table:
rating_id: An auto-incrementing unique identifier for each rating entry.
user_id: Foreign key referencing the user who provided the rating.
application_id: Foreign key referencing the leaseholder ad for which the rating is given.
rating: The numerical rating given by the user for the leaseholder ad.
review: Textual review or feedback provided by the user for the leaseholder ad.

Leaseholder Current Residents Table:
application_id: Primary key and Foreign key referencing the leaseholder ad for which current residents are recorded.
user_id: Foreign key referencing the user who is a current resident for the leaseholder ad.

5. Interface Design

5.1 Internal Interfaces:

User Registration and Authentication:
When a user registers on the website, their information, such as email, name, password, age, gender, etc., is stored in the "User" table in the database using an SQL INSERT statement.
During the authentication process, the backend server checks the provided credentials against the "User" table to verify the user's identity and grant access to specific features based on flags like "is_admin" or "is_leaseholder."

Leaseholder and HouseSeeker Ads Creation:
Leaseholders and house seekers create advertisements on the website. When they submit an ad form, the backend server processes the information and inserts the ad details into the "Leaseholder Ads" or "HouseSeeker Ads" table, respectively. The user_id associated with the ad is fetched from the user's session, which was established during the authentication process, to link the ad with the corresponding user.

Preference and Image Uploading:
Leaseholders and house seekers can upload preferences, images, and other details related to their ads. This information is stored in tables like "Leaseholder Food Preferences," "Leaseholder Images," "Leaseholder Gender Preferences," "HouseSeeker Food Preferences," and "HouseSeeker Gender Preferences." The backend server receives the uploaded data, validates it, and inserts the information into the appropriate tables, linking them to the corresponding leaseholder or house seeker ad using application_id.

Applicant Management:
When a user applies for a leaseholder or house seeker ad, the backend server creates a new entry in either the "Leaseholder Applicant" or "HouseSeeker Applicant" table, associating the user_id and application_id accordingly. The status of the application (e.g., pending, approved, rejected) is updated in the respective table entry based on the leaseholder or house seeker's response.

Chat Room and Messaging:
Users can communicate with each other through the website's chat feature. When a user sends a message, the backend server stores the message in the "Chat" table along with the user_id and the corresponding room_id. The backend server manages the creation and retrieval of chat rooms, associating them with user_1_id and user_2_id in the "Room" table.

Rating and Reviews:
Users can rate and review leaseholder ads. When a user submits a rating and review, the backend server inserts the data into the "Rating" table, associating the user_id and application_id accordingly.

Leaseholder Current Residents:
The "Leaseholder Current Residents" table maintains the list of current residents for each leaseholder property. When a user becomes a resident of a leaseholder property, the backend server adds a new entry in the table, linking the user_id with the application_id.

5.2 External Interfaces:

User Interactions with the Website

User Registration and Authentication: Users interact with the website by registering an account and logging in. The website provides forms for user registration and login. When a user submits the registration form, the backend server processes the data, and if successful, the user is granted access to the website's features. During login, the system verifies the user's credentials against the data stored in the "User" table to authenticate them.

Creating and Managing Ads: Leaseholders and house seekers interact with the website to create ads for their respective requirements. They fill out forms with ad details, preferences, and images. Upon submission, the backend server processes the data and inserts it into the appropriate tables. Users can also manage their existing ads, update details, or remove ads.

Applying for Ads: House seekers can apply for leaseholder ads through the website. When a user submits an application, the backend server creates an entry in the "HouseSeeker Applicant" table, linking the user_id and the application_id.

Chat and Messaging: Users can communicate with each other through the website's chat feature. When a user sends a message, the frontend sends the message data to the backend server via APIs. The server processes the message and stores it in the "Chat" table, associating it with the user_id and the room_id.

Rating and Reviews: Users can provide ratings and reviews for leaseholder ads. When a user submits a rating and review through the frontend, the backend server processes the data and inserts it into the "Rating" table, associating the user_id and the application_id.

Viewing Current Residents: Leaseholders can view the current residents of their property through the website. The backend retrieves the data from the "Leaseholder Current Residents" table and displays it to the leaseholder.

Interactions with Other Systems:

User Authentication: The website is integrated with external authentication systems, such as OAuth or Single Sign-On (SSO), to provide users with various options for authentication.

Notifications: The system can interact with external notification services (e.g., email) to send users notifications about important events, such as application approvals and changes in ad status.

Analytics and Monitoring: The system may be integrated with analytics and monitoring tools to track website usage, user behavior, and performance metrics. This data can help improve the website's functionality and user experience.

5.3 API Documentation:

Accomatch has following API that implements separate functionalities:

1. API Name: verifySingleAd
Endpoint: /api/admin/verify/one
Request: HTTP POST
RequestBody: A JSON object containing two key-value pairs: "isVerified" (integer) and "leaseholderApplicationId" (integer).
Functionality: This API is used to verify a single ad. It takes the isVerified flag (1 or 0) and the leaseholderApplicationId (ID of the ad to be verified) as input. The method attempts to verify the specified ad and returns a success message if the verification is successful. If any error occurs, it returns an error message.

2. API Name: verifyAllAd
Endpoint: /api/admin/verify/all
Request: HTTP POST
RequestBody: A JSON object representing a Posts object.
Functionality: This API is used to verify all unverified ads. It takes a Posts object as input. The method attempts to verify all unverified ads based on the properties set in the Posts object. It returns a success message if all ads are successfully verified. If any error occurs, it returns an error message.

3. API Name: getListOfPosts
Endpoint: /api/admin/get/list/post
Request: HTTP GET
Functionality: This API is used to retrieve the list of posts for the admin. It does not require any request body or parameters. The method simply fetches and returns the list of posts.

4. API Name: getListOfPostsByStatus
Endpoint: /api/admin/get/list/postbystatus
Request: HTTP POST
RequestBody: A JSON object containing a key-value pair: "status" (integer).
Functionality: This API is used to retrieve the list of posts for the admin based on their status. It takes the status (integer) as input to filter posts by their status. The method attempts to retrieve the list of posts having the specified status and returns the filtered list.

5. API Name: filter
Endpoint: /api/applicant/posts/filter
Request: HTTP POST
RequestBody: A JSON object (Map) containing applicant preferences as key-value pairs. The keys in the map are "gender_pref", "food_pref", "age", and "room_type", and their corresponding values are strings.
Response: A list of Posts that match the given applicant preferences.
Functionality: This API is used to filter posts based on applicant preferences. It takes a JSON object containing applicant preferences as input. The preferences include the applicant's gender preference ("gender_pref"), food preference ("food_pref"), age ("age"), and room type ("room_type"). The method splits the comma-separated values for "gender_pref" and "food_pref" and passes all preferences to the applicantPostFilterService for filtering. The service method filterPost processes the preferences and returns a list of Posts that match the provided criteria.

6. API Name: apply
Endpoint: /api/applicant/apply
Request: HTTP POST
RequestBody: A JSON object representing the LeaseHolderApplicantModel.
Response: A string representing the status of the application.
Functionality: This API is used for an applicant to apply on any house application. It takes a JSON object containing the details of the LeaseHolderApplicantModel as input. The LeaseHolderApplicantModel represents the model details of the leaseholder applicant. Additionally, a ChatRoomModel object is created, representing a chat room where the user can communicate with the property owner or administrator. The method then calls the apply method from the ApplyonPostService service to handle the application process and returns the status of the application.

7. API Name: isUserApplied
Endpoint: /api/applicant/isApplied
Request: HTTP POST
RequestBody: A JSON object containing user_id and application_id as key-value pairs.
Response: A boolean value indicating whether the user has already applied for the post.
Functionality: This API is used to verify if a user has already applied for a specific house post. It takes a JSON object containing user_id and application_id as input. The method extracts these values from the request body, creates a LeaseHolderApplicantModel object with the provided user_id and application_id, and then calls the isApplied method from the ApplyonPostService

service to check if the user has already applied for the post. The method returns a boolean value indicating the result of the check.

8. API Name: sendMessage
Endpoint: /api/chat/send
Request: HTTP POST
RequestBody: A JSON object containing three key-value pairs: "user_id" (integer), "room_id" (integer), and "message" (string).
Response: A string representing a success message or an error message.
Functionality: This API is used to send a chat message. It takes a JSON object containing the user_id of the sender, the room_id of the chat room where the message should be sent, and the message to be sent. The method then creates a ChatMessageModel object with the provided details, including the current timestamp, and passes it to the chatService to handle the message sending process. If the message is sent successfully, it returns a success message; otherwise, it throws a ChatMessageException with an error message.

9. API Name: getMessages
Endpoint: /api/chat/get/{room_id}
Request: HTTP GET
PathVariable: room_id (integer)
Response: An ArrayList of ChatMessageModel containing chat messages for the specified room.
Functionality: This API is used to get all chat messages for a specific chat room. It takes a room_id as a path variable, which indicates the chat room from which the messages should be retrieved. The method then calls the chatService.getMessages(room_id) to fetch all chat messages for the specified room. If the messages are fetched successfully, it returns an ArrayList of ChatMessageModel containing the chat messages; otherwise, it throws a ChatMessageException with an error message.

10. API Name: getRoomId
Endpoint: /api/room/getRoomId
Request: HTTP POST
RequestBody: A JSON object containing two key-value pairs: "user_id" (integer) and "application_id" (integer).
Response: An integer representing the roomId based on the given arguments.
Functionality: This API is used to get the roomId on the basis of userId and applicationId. It takes a JSON object containing the user_id and application_id as input. The method then checks if the application_id and user_id are valid (greater than 0) using an InvalidInputException. If the provided inputs are valid, it calls the getRoomId method from the ChatRoomService to retrieve the roomId based on the given application_id and user_id, and returns the roomId as an integer

11. API Name: createAD
Endpoint: /api/application/create
Request: HTTP POST
RequestBody: A JSON object containing parameters to create an application. The specific structure and keys of the JSON object are not explicitly defined in the code.
Response: A string representing the status of the creation process.
Functionality: This API is used to create applications. It takes a JSON object containing the necessary parameters as input. The createApplicationService is responsible for handling the application creation process. The method calls the createAD method from the CreateApplicationFactory service (implementation) and passes the requestBody to it. The implementation of createAD method inside CreateApplicationFactory handles the application creation logic. If the creation is successful, it returns a success message; otherwise, it returns an error message.

12. API Name: getListOfAllApplicantPosts
Endpoint: /api/houseSeeker/getListOfAllApplicantPosts
Request: HTTP GET
Response: A list of HouseSeekerModel objects representing the applicant posts.
Functionality: This API is used to retrieve a list of all applicant posts. It does not require any specific request parameters. The method calls the getListOfAllApplicantPosts method from the HouseSeekerService to fetch the list of HouseSeekerModel objects representing the applicant posts. The service is responsible for fetching data from the database or any other data source and processing it to return the list of applicant posts. The method then returns the list of HouseSeekerModel objects as the response.

13 .API Name: getListofApplicants
Endpoint: /api/leaseholder/applicant/get/list/applicant/{application_id}
Request: HTTP GET
PathVariable: application_id (integer)
Response: A list of Applicant objects representing the applicants for the given application_id.
Functionality: This API is used to retrieve a list of applicants for a specific leaseholder application. It takes application_id as a path variable, which indicates the ID of the application for which the applicants should be fetched. The method then checks if the provided application_id is valid (greater than 0) using an InvalidInputException. If the application_id is valid, it calls the getListOfApplicants method from the LeaseApplicationService to fetch the list of Applicant objects representing the applicants for the specified leaseholder application. The method then returns the list of Applicant objects as the response.

14. API Name: chanegStatusofApplication
Endpoint: /api/leaseholder/applicant/changeStatus
Request: HTTP POST
RequestBody: A JSON object containing four key-value pairs: "application_id" (integer), "user_id" (integer), "status" (string), and "email" (string).
Response: A boolean value indicating whether the status has been changed successfully.

Functionality: This API is used to change the status of an applicant on any leaseholder application. It takes a JSON object containing the necessary parameters as input, including the application_id, user_id, status, and email. The method then checks if the provided application_id and user_id are valid (greater than 0) using an InvalidInputException. If the provided inputs are valid, it calls the changeStatusofApplicant method from the LeaseApplicationService to change the status of the applicant for the specified application. Additionally, it sends an email to the provided email address to notify the applicant about the status change using the mailSenderClass. The method returns a boolean value indicating whether the status change was successful.

15 . API Name: getListOfPosts
Endpoint: /api/leaseholder/dashboard/get/list/post
Request: HTTP GET
Response: A list of Posts objects representing the posts.
Functionality: This API is used to retrieve a list of posts for the leaseholder dashboard. It takes an Authentication object as input to get the currently authenticated user details. The method then extracts the username from the Authentication object, logs it, and calls the getListOfPosts method from the DashboardInterface (or its implementation) to fetch the list of Posts objects representing the posts. The method returns the list of Posts objects as the response.

16. API Name: getPostDetails
Endpoint: /api/leaseholder/dashboard/get/post/details/{applicationId}
Request: HTTP GET
PathVariable: applicationId (integer)
Response: A Posts object representing the details of the post with the given applicationId.
Functionality: This API is used to retrieve the details of a post based on the application ID. It takes applicationId as a path variable, which indicates the ID of the application for which the post details should be fetched. The method then checks if the provided applicationId is valid (greater than 0) using an InvalidInputException. If the applicationId is valid, it calls the getPostByApplicationId method from the DashboardInterface (or its implementation) to fetch the details of the post with the specified applicationId. If the post is found, it returns the Posts object representing the post details as the response; otherwise, it throws a PostNotFoundException.

17. API Name: getListOfImages
Endpoint: /api/leaseholder/dashboard/get/list/images/{applicationId}
Request: HTTP GET
PathVariable: applicationId (integer)
Response: A list of strings representing the URLs of images associated with the post with the given applicationId.
Functionality: This API is used to retrieve the list of images for a post based on the application ID. It takes applicationId as a path variable, which indicates the ID of the application for which the images should be fetched. The method then checks if the provided applicationId is valid (greater than 0) using an InvalidInputException. If the applicationId is valid, it calls the getListOfImagesByApplicationId method from the DashboardInterface (or its implementation) to

fetch the list of image URLs associated with the post. If the post is found, it returns the list of strings representing the image URLs as the response; otherwise, it throws a PostNotFoundException.

18. API Name: getListOfFoodPreferences
Endpoint: /api/leaseholder/dashboard/get/list/food/{applicationId}
Request: HTTP GET
PathVariable: applicationId (integer)
Response: A list of strings representing the food preferences associated with the post with the given applicationId.
Functionality: This API is used to retrieve the list of food preferences for a post based on the application ID. It takes applicationId as a path variable, which indicates the ID of the application for which the food preferences should be fetched. The method then checks if the provided applicationId is valid (greater than 0) using an InvalidInputException. If the applicationId is valid, it calls the getListOfFoodPreferencesByApplicationId method from the DashboardInterface (or its implementation) to fetch the list of food preferences associated with the post. If the post is found, it returns the list of strings representing the food preferences as the response; otherwise, it throws a PostNotFoundException.

19. API Name: getListOfGenderPreferences
Endpoint: /api/leaseholder/dashboard/get/list/gender/{applicationId}
Request: HTTP GET
PathVariable: applicationId (integer)
Response: A list of strings representing the gender preferences associated with the post with the given applicationId.
Functionality: This API is used to retrieve the list of gender preferences for a post based on the application ID. It takes applicationId as a path variable, which indicates the ID of the application for which the gender preferences should be fetched. The method then checks if the provided applicationId is valid (greater than 0) using an InvalidInputException. If the applicationId is valid, it calls the getListOfGenderPreferencesByApplicationId method from the DashboardInterface (or its implementation) to fetch the list of gender preferences associated with the post. If the post is found, it returns the list of strings representing the gender preferences as the response; otherwise, it throws a PostNotFoundException.

20. API Name: getListOfPersonalPosts
Endpoint: /api/leaseholder/dashboard/get/list/getListOfPersonalPosts/{user_Id}
Request: HTTP GET
PathVariable: user_Id (integer)
Response: A list of Posts objects representing the personal posts of the user with the given user_Id.
Functionality: This API is used to retrieve a list of personal posts for a specific user based on their user ID. It takes user_Id as a path variable, which indicates the ID of the user for whom the personal posts should be fetched. The method then checks if the provided user_Id is valid (greater than 0) using an InvalidInputException. If the user_Id is valid, it calls the

getListOfPersonalPosts method from the DashboardInterface (or its implementation) to fetch the list of Posts objects representing the personal posts of the user. If the user is found, it returns the list of Posts objects as the response; otherwise, it throws a PostNotFoundException.

21. API Name: getListofAppliedPosts
Endpoint: /api/leaseholder/loggedinapplicant/get/list/applicant/{user_id}
Request: HTTP GET
PathVariable: user_id (integer)
Response: A list of Posts objects representing the posts on which the user with the given user_id has applied.
Functionality: This API is used to provide the user with all the posts they have applied to. It takes user_id as a path variable, which indicates the ID of the user for whom the applied posts should be fetched. The method then checks if the provided user_id is valid (greater than 0) using an InvalidInputException. If the user_id is valid, it calls the getListOfLoggedinApplicants method from the LeasePostsLoggedinService (or its implementation) to fetch the list of Posts objects representing the posts on which the user has applied. If the user is found, it returns the list of Posts objects as the response; otherwise, it throws an InvalidInputException.
Note: The exact implementation of LeasePostsLoggedinService is not provided in the code snippet, but it is assumed to be responsible for handling the functionality to retrieve the applied posts for the user based on the provided user_id.

22. API Name: getListOfAllRatings
Endpoint: /api/reviews/getListOfAllRatings/{application_id}
Request: HTTP GET
PathVariable: application_id (integer)
Response: A list of Review objects representing all the ratings people have given to the post with the specified application_id.
Functionality: This API is used to return all the ratings that people have given to a post with the given application_id. It takes application_id as a path variable, which indicates the ID of the post for which the ratings should be fetched. The method then checks if the provided application_id is valid (greater than 0) using an InvalidInputException. If the application_id is valid, it calls the getAllReviews method from the ReviewServiceImplementation (or its interface) to fetch the list of Review objects representing all the ratings for the specified post. If the post is found, it returns the list of Review objects as the response; otherwise, it throws an InvalidInputException.

23. API Name: getAverageRatings
Endpoint: /api/reviews/getAverageRatings/{application_id}
Request: HTTP GET
PathVariable: application_id (integer)
Response: A list of Ratings objects representing the average ratings people have given to the post with the specified application_id.
Functionality: This API is used to return the average rating that people have given to a post with the given application_id. It takes application_id as a path variable, which indicates the ID of the post for which the average rating should be fetched. The method then checks if the provided

application_id is valid (greater than 0) using an InvalidInputException. If the application_id is valid, it calls the getRatingsAverage method from the ReviewServiceImplementation (or its interface) to fetch the list of Ratings objects representing the average ratings for the specified post. If the post is found, it returns the list of Ratings objects as the response; otherwise, it throws an InvalidInputException.

24. API Name: getAllPostReviews
Endpoint: /api/reviews/getAllPostReviews
Request: HTTP GET
Response: A list of Review objects representing all the reviews of the posts.
Functionality: This API is used to get all the reviews of the posts. It calls the getAllPostReviews method from the ReviewServiceImplementation (or its interface) to fetch the list of Review objects representing all the reviews of the posts. It then returns the list of Review objects as the response.

25. API Name: getAllAverageRatings
Endpoint: /api/reviews/getAllAverageRatings
Request: HTTP GET
Response: A list of Ratings objects representing all the average ratings of the posts.
Functionality: This API is used to get all the average ratings of the posts. It calls the getAllRatingsAverage method from the ReviewServiceImplementation (or its interface) to fetch the list of Ratings objects representing all the average ratings of the posts. It then returns the list of Ratings objects as the response.

26. API Name: createReview
Endpoint: /api/reviews/createReview
Request: HTTP POST
RequestBody: A Review object containing the details of the review to be created. The Review object has properties such as userId, applicationId, and other details related to the review.
Response: Void (No specific response is returned for this API)
Functionality: This API is used to handle a POST request to create a new review for a user. It takes a Review object as the request body, which contains the details of the review to be created. The method then extracts the userId and applicationId from the Review object and logs that the review controller is active. After that, it calls the createReview method from the ReviewService (or its implementation) and passes the Review object to it for further processing. The actual functionality of creating a review is expected to be implemented in the ReviewService.

27. API Name: signUp
Endpoint: /api/users/signup
Request: HTTP POST
RequestBody: A UserModel object containing user details for signup. The UserModel object represents the user's information such as username, password, email, etc.
Response: A string message indicating the status of the signup process.

Functionality: This API is used for user registration (signup). It takes a UserModel object as the request body, which contains the user's details for signup. The method then calls the SignUp method from the UserService (or its implementation) and passes the UserModel object to it for further processing. The SignUp method is responsible for registering the user with the provided details and returns a string message indicating the status of the signup process.

28. API Name: login
Endpoint: /api/users/login
Request: HTTP POST
RequestBody: A UserModel object containing user credentials for login. The UserModel object represents the user's login credentials such as username/email and password.
Response: A map containing authentication status and user-related data on successful login.
Functionality: This API is used for user login. It takes a UserModel object as the request body, which contains the user's login credentials. The method then calls the Login method from the UserService (or its implementation) and passes the UserModel object to it for further processing. The Login method is responsible for authenticating the user with the provided credentials and returns a map containing the authentication status and user-related data on successful login.

29. API Name: login (Dummy Endpoint)
Endpoint: /api/users/login
Request: HTTP GET
Response: A string message indicating successful login.
Functionality: This is a dummy endpoint used for testing purposes. It simply returns a string message indicating successful login.

30. API Name: getUserInformation
Endpoint: /api/users/get/{id}
Request: HTTP GET
PathVariable: id (integer) - The user ID whose information is to be retrieved.
Response: The UserModel object containing user information.
Functionality: This API is used to retrieve user information by user ID. It takes the id as a path variable, which represents the user ID. The method then calls the getUserInfo method from the UserService (or its implementation) and passes the id to it for further processing. The getUserInfo method retrieves the user information based on the provided user ID and returns the UserModel object containing user information.

31. API Name: forgotPassword
Endpoint: /api/users/forgot/password
Request: HTTP POST
RequestBody: A UserModel object containing the user's email for password reset.
Response: A string message indicating the status of the password reset process or an error message.
Functionality: This API is used to handle the forgot password feature. It takes a UserModel object as the request body, which contains the user's email for password reset. The method then calls

the CheckEmailID method from the UserService (or its implementation) to check if the provided email exists in the system. If the email exists, it sends a password reset email to the user's email address using the MailSenderClass. The email contains a link to reset the password. If the email does not exist, it returns an appropriate error message.

32. API Name: updatePassword
Endpoint: /api/users/update/password
Request: HTTP POST
RequestBody: A UserModel object containing the user's email and new password for password update.
Response: A string message indicating the status of the password update process or an error message.
Functionality: This API is used to handle the password update process after the user clicks the reset password link from the email. It takes a UserModel object as the request body, which contains the user's email and new password. The method then calls the ForgotPassword method from the UserService (or its implementation) to update the password. The ForgotPassword method processes the password update and returns a string message indicating the status of the password update process or an error message.

6. Security Design

6.1 User Authentication and Access Control:

- The system will implement user authentication using JSON Web Tokens (JWT) for stateless authentication.
- Upon registration, users will provide their email address, password, and other required details.
- Passwords will be securely hashed using a strong cryptographic hashing algorithm (bcrypt) before storing them in the database.
- User login will be performed through the application's login endpoint, where users will submit their email and password.
- The CustomUserDetailsService will be responsible for loading user details from the database based on the provided email during authentication.
- If the email and password match, the JwtService will generate a JWT token containing user details and roles, which will be sent back to the client for subsequent requests.
- Access to different parts of the system will be controlled based on user roles and permissions carried within the JWT token.
- Role-based access control will be used to assign roles to users, such as "AP" (Apartment), "AD" (Admin), or "LH" (Leaseholder).
- The JwtFilter will intercept incoming requests and validate the JWT token, setting the authenticated user's details in the SecurityContextHolder.

- The application's controllers will check the user's roles and permissions stored in the security context to authorize access to certain endpoints and actions.

6.2 Data Encryption and Protection:.

- User passwords are securely hashed before storing in the database using BCryptPasswordEncoder, making them irreversibly encrypted.
- Sensitive data, such as user email, will be stored in the JWT token, which is digitally signed but not encrypted. JWT is secure due to its digital signature, which prevents tampering. The strong hashing algorithm (SHA-256) ensures the integrity of the token, and the secret key keeps it confidential. These features make JWT a reliable and widely used authentication mechanism.
- The database access is managed by the application's authentication mechanism and the roles/permissions assigned to users, reducing the risk of unauthorized data access

6.3 Compliance Measures:

- Security events and incidents are logged and can be reviewed regularly.
- The application maintains access logs and monitors user activities to detect any suspicious or unauthorized behavior.
- We carefully log user login attempts and keep a record of successful logins, providing a detailed account of user access and authentication events. Additionally, our logs are designed to track user interactions with various parts of the system, including API endpoints accessed and actions performed by specific users. By closely monitoring user activities through these logs, we gain valuable insights into user behavior, which helps us better understand their actions and aids in identifying any suspicious or unauthorized activities. This proactive approach to monitoring user behavior enhances our application's security measures and ensures we maintain a safe and trusted platform for our users.

7. Performance Considerations

7.1 Scalability:

In our project, we have focused on scalability by adopting a modular architecture. Different layers like controller, service, and repository are organized into separate components, each with its specific responsibility. For instance, we have dedicated services for user authentication, data processing, and messaging. This approach enables us to scale these components independently as per their demand, ensuring efficient resource utilization and overall improved performance. By following the Single Responsibility Principle (SRP), each service is designed to handle a specific task, promoting maintainability and flexibility in our application.

Additionally, to manage logging efficiently and consistently throughout the project, we have implemented the Singleton design pattern for the LoggerClass. To achieve this, we have provided a static method getLogger() within the LoggerClass, which returns the single instance of the LoggerClass. The constructor of the LoggerClass is made private to prevent other parts of the code from creating multiple instances of the logger.

By using the code Logger logger = LoggerClass.getLogger(); in every file where logging is required, we ensure that all parts of our application use the same logger instance. This centralized approach to logging enables us to control the logging behavior from a single point and avoids the overhead of creating multiple logger instances. Moreover, the use of Singleton in the logger aligns with our modular architecture and SRP, allowing each service to focus on its designated responsibility while sharing a common logging utility.

By using the builder pattern, code can create a HouseSeekerModel instance by chaining the builder methods as needed, setting only the desired attributes, while leaving others with default values or leaving them unset. Once all the necessary attributes are specified, the build() method is called, which returns the final, fully constructed HouseSeekerModel object.

This approach simplifies the object creation process, makes the code more maintainable, and enhances code readability by clearly defining the steps to create a HouseSeekerModel. It provides an elegant way to create complex objects without the need for numerous constructor overloads, resulting in a clean and flexible solution for creating instances of HouseSeekerModel.

7.2 Load Handling:

In the user authentication module, during peak periods, we utilize caching to store authentication tokens, avoiding repetitive database queries and enhancing response times. This aligns with SOLID principles, promoting single responsibility and clean code practices by improving efficiency and readability. Additionally, we ensure data consistency by refreshing the cache at intervals to maintain up-to-date information. Caching plays a crucial role in load handling, optimizing performance during peak times.

7.3 Fault Tolerance and Recovery Measures:

We have applied the Dependency Inversion Principle (DIP) by abstracting our database interactions and using interfaces. This allows us to easily switch between different database implementations ensuring fault tolerance. In cases of service failures, we implement exception handling patterns to gracefully handle errors and prevent cascading failures. This aligns with clean code practices, as we handle errors effectively with proper error messages, logging, and monitoring. The logger helps us to log and monitor errors in real-time, facilitating quick recovery measures.

8. Technology Stack

8.1 Programming Languages: Identify the languages that will be used in development.

Languages: Java, JavaScript, HTML, CSS

8.2 Frameworks and Libraries: List the frameworks and libraries that will be used.

The following frameworks and libraries used in the application:

Front-end:

React: A popular JavaScript library for building user interfaces. It allows you to create reusable UI components and manage the application's state efficiently.

Back-end:

Spring Boot: A framework for building Java-based web applications quickly and with minimal configuration. It provides a set of features, including dependency injection, security, and web services, making it easier to develop robust back-end applications.

Database:

MySQL: An open-source relational database management system (RDBMS) widely used for storing and managing structured data.

8.3 Tools and Environments: Describe the tools and environments used for development, testing, deployment, etc.

Tools & Technology:

The following technologies and tools in our application:

- UI/UX: User Interface and User Experience design, which focuses on creating visually appealing and user-friendly interfaces for your application.

- Testing: The process of validating and verifying that your application functions as expected. It includes unit testing, integration testing, and end-to-end testing.
- Bootstrap: A front-end CSS framework that provides pre-designed components and styles to enhance the appearance and responsiveness of your application.
- CSS: Cascading Style Sheets, used to style and format the presentation of your web pages.
- JavaScript: A programming language used for client-side scripting to add interactivity and dynamic elements to your web pages.
- JUnit5: A popular testing framework for Java applications, used for writing and executing unit tests to ensure the correctness of individual code units (e.g., methods, classes).
- Mockito: A Java-based mocking framework used for creating mock objects to facilitate unit testing by isolating dependencies.
- Postman: A popular API testing tool used to test and interact with APIs, allowing developers to send requests, view responses, and perform automated testing.

9. Testing Strategy

9.1 Unit Testing: Describe how individual components will be tested.

Understanding Requirements: We begin by thoroughly understanding the requirements and specifications for each component we are going to develop. Having a clear understanding of what the component should do helps us in writing effective test cases.

Writing Test Cases: With the requirements in hand, we write test cases for each component. We consider different scenarios, edge cases, and potential error conditions. These test cases serve as our safety net to catch any issues during development and future changes.

Test-Driven Development (TDD): Sometimes, we adopt the Test-Driven Development approach, where we write the test cases first and then develop the component to make those tests pass. This approach ensures that we focus on writing code that fulfills the requirements.

Mocking and Stubbing Dependencies: During testing, we isolate the component from its dependencies using mocking and stubbing techniques. This allows us to simulate the behavior of external components, ensuring the tests are consistent and repeatable.

Running Tests Frequently: We regularly run the test suite while developing the component. This helps us catch issues early and provides immediate feedback on whether the component is functioning correctly or not.

Debugging and Fixing Issues: When a test case fails, we immediately investigate the cause of the failure. We use debugging tools and logs to identify the problem and then fix the issue in the component code.

Regression Testing: Whenever we make changes to the component or introduce new features, we rerun the entire test suite, including all previously passing test cases. This ensures that our changes do not break existing functionality.

Continuous Integration (CI): In a CI environment, our test suite is automatically run whenever code changes are committed. This practice helps catch issues early and prevents integration problems.

Maintaining Test Suite: As the application evolves, we update the test suite to accommodate new features and changes. It's essential to keep the test suite up to date to reflect the current behavior of the component.

9.2 Integration Testing: Explain how the interaction between components will be tested.

Integration testing ensures that different components of a software application work together correctly. For example, in our web application, integration testing would verify that the frontend user interface interacts smoothly with the backend server and that data is exchanged accurately between the two. Let's say in our application LeaseHolder creates posts and view their applicants. During integration testing, we would simulate user interactions by creating a post through the frontend interface and checking if the backend correctly stores and retrieves the messages. Additionally, we would test if the frontend properly displays the posts fetched from the backend. This testing would cover various scenarios, such as posting messages with different character limits, handling large numbers of posts, and checking how the application handles concurrent users accessing the system.

The test environment for integration testing would mimic the production setup, including databases and services. We would also use mock objects or stubs to simulate external services if needed. Each test case would validate the interaction between the frontend and backend, including data flow, error handling, and communication protocols. Issues identified during testing would be logged and resolved, and regression testing would be performed to ensure that previous interactions remain intact after fixes. Continuous integration and continuous testing would help maintain a seamless integration process as the application evolves. By conducting thorough integration testing, we can ensure that the different components of the social media application work harmoniously together, providing users with a smooth and reliable experience.

9.3 System Testing: Outline how the system as a whole will be tested.

System testing involves evaluating the entire software application to ensure that all individual components work together as expected and meet the overall requirements. System testing aims to validate the application's functionality, performance, security, and compatibility with the intended environment.

For example, on our website. In system testing, we would assess the entire application flow, from user registration,creating a post, and chatting between applicant and Leaseholder, to accepting the applicant and Logout process. We would verify if the data flows correctly between the frontend, backend, and the database. Additionally, we would examine how the system handles multiple users simultaneously, whether it gracefully recovers from errors, and if it provides the expected responses under different user scenarios.

9.4 User Acceptance Testing: Discuss how users will test the system to ensure it meets their needs.

During user testing, the system's complete application flow would be evaluated, encompassing actions such as user registration, post creation, and chatting between applicants and leaseholders, as well as accepting applicants and the logout process. The focus would be on ensuring smooth data flow between the frontend, backend, and database components. Moreover, the system's performance in handling multiple users simultaneously would be examined to gauge its scalability. Users would deliberately trigger error scenarios to assess how well the system recovers from unexpected issues and if it provides appropriate error messages. Various user scenarios would be simulated to validate that the system delivers expected responses under different usage conditions. For example, users would register on the site, create new posts, interact through chat, and accept or reject applicants. They would also perform stress testing by accessing the application with multiple users simultaneously, testing its ability to handle peak loads effectively. The collected feedback from users would be used to improve the application's usability, performance, and overall alignment with the needs of its intended users.

10. Conclusion and Future Work

10.1. Summary:

The document provides a comprehensive overview of the "Accomatch" application, which serves as a platform for users to search for accommodations worldwide. It emphasizes the application's user-centric approach, aiming to offer a seamless experience for both lease owners and applicants.

1. Application Overview: The document introduces the Accomatch application, explaining its purpose and the benefits it brings to users seeking accommodations. It positions the app as a valuable resource for those in need of housing solutions, emphasizing its global reach.

2. User Guidance: A significant portion of the document is dedicated to providing step-by-step instructions on how to use the application effectively. It covers the entire process, from account creation to searching for accommodations and establishing communication with lease owners. This guidance ensures that first-time users can navigate the app with ease.

3. Accommodation Selection: The document highlights the flexibility in accommodation selection. Users can customize their preferences for food, gender, and room type, ensuring they find accommodations that align with their specific needs. This tailored approach enhances user satisfaction.

4. Communication and Feedback: Communication features play a crucial role in the application, and the document elaborates on this aspect. It explores the chat functionality, enabling direct communication between users and lease owners. Additionally, it covers the review and rating system, encouraging users to provide feedback on their experiences, fostering transparency and accountability.

5. Collaborative Environment: The application's collaborative environment is emphasized, emphasizing how both lease owners and applicants can specify their criteria and preferences. This fosters effective communication and negotiation, creating a harmonious interaction between the parties involved.

6. Customization Options: The document underlines the application's ability to cater to individual preferences and requirements. Users can define specific criteria, allowing for a personalized and tailored accommodation search process. This focus on customization enhances the user experience.

In conclusion, the document offers a detailed understanding of Accomatch, encompassing various aspects to guide users effectively. From providing a clear application overview to explaining accommodation selection, communication features, and customization options, the document showcases the application's user-centric approach, positioning it as a valuable solution for global accommodation seekers.

10.2. Future Improvements or Enhancements:

1. Mobile Application Development: Extend the reach and accessibility of the Accomatch platform by developing dedicated mobile applications for both Android and iOS. Mobile apps can provide a more seamless user experience, leveraging device-specific features like GPS, push notifications, and offline access. The mobile apps should offer the same functionalities as the web version, ensuring a consistent user experience across different platforms.

2. UI/UX Design Overhaul: Conduct a comprehensive UI/UX design overhaul to create an intuitive and visually appealing interface for the application. Implement modern design principles, use attractive visuals, and focus on responsive and user-friendly layouts. Enhancing the user experience can lead to increased user engagement and retention.

3. Advanced Search and Filters: Implement more advanced search and filtering options to allow users to find accommodations based on specific criteria like location, price range, amenities, and proximity to landmarks or institutions.

4. Real-Time Availability: Integrate a real-time availability feature, allowing users to see the current availability status of accommodations and make immediate bookings.

5. Social Media Integration: Enable social media integration, allowing users to sign up or log in using their social media accounts. Additionally, users can share their accommodation experiences on their social media platforms, promoting the app and increasing its user base.

6. In-App Messaging System: Enhance the in-app messaging system to facilitate seamless communication between lease owners and applicants. Add features like message history, message notifications, and message status indicators.

7. Multi-Language Support: Add multi-language support to cater to users from different regions and enhance accessibility for a global audience.

8. Payment Integration: Integrate secure payment gateways to facilitate smooth and secure transactions for bookings and rental payments.

9. Analytics and Insights: Implement analytics and reporting features to track user behavior, popular accommodation choices, and user engagement. This data can provide valuable insights to improve the application further.

10. User Support and Help Center: Create a comprehensive user support system and a help center with frequently asked questions (FAQs) to assist users with any issues they may encounter.

By considering these improvements and enhancements, the "Accomatch" application can offer a more seamless, feature-rich, and secure experience to its users, attracting a wider audience and fostering user loyalty.