

Capstone Project Phase A

Project number: 23-2-D-14

UrbanHive – Find your community!

Danor Sinai: danor.sinai@e.braude.ac.il

Sahar Oz: sahar.oz@e.braude.ac.il

Supervisor: Zeev Brazily, zbarzily@braude.ac.il

Table of Contents

1. Abstract 2

1.1. Introduction 2

2. Background and Related Work..... 3

2.1. Ubiquity of Internet Connectivity..... 3

2.2. Community Interactions and Marketplaces: 3

2.2.1 Prevalent Platforms Today: 3

2.2.1.1 Facebook Groups: 3

2.2.1.2 Online Marketplaces: 3

2.2.1.3 Localized Apps: 3

2.2.1.4 Word of Mouth: 3

2.2.2. Aspiration: 3

3. Agile Development 4

4. Client-server 5

4.1. Understanding UrbanHive's Client-Server Model: 5

4.2. UrbanHive's Client-Server model uses: 5

4.3. UrbanHive's Client-Server Network: 6

4.4. Advantages of UrbanHive's Client-Server Computing: 6

5. Cloud computing in UrbanHive..... 7

6. Expected Achievements 8

6.1. Goals: 8

6.2. Success Criteria: 8

6.3. Unique Features: 9

7. Research / Engineering Process..... 9

8. Product 10

9. Software Engineering Documents..... 11

9.1. Application Architecture 11

9.1.1.1. React/React Native for the Frontend: 12

9.1.1.2. RESTapi: 12

9.1.1.3. Python Flask: 13

9.1.1.4. MongoDB: 14

9.1.1.5. Auth Users: 15

9.1.1.6. Push Notification: 16

10. Diagrams..... 17

10.1. Use-Case Diagram 17

10.2. Class Diagram..... 18

10.3. Activity Class (Login/signup ->Home Screen->Functions->Exit System)..... 19

11. GUI Design 20

12. Evaluation / Verification Plan 24

13. References..... 26

1. Abstract

This project proposes an innovative digital platform, “UrbanHive”, designed as a comprehensive social network to support and foster interactions within diverse residential communities. UrbanHive aims to enhance community engagement and support through numerous features, such as creating and joining local communities, displaying public volunteer opportunities, and managing community events.

The project encompasses several functional requirements, including user authentication, personal account management, community management, and region-specific content filtering.

Another prominent feature of UrbanHive is the creation and management of community groups. Users can create a community, invite members, establish community rules, and manage community events, among other activities. Additionally, users can choose to share their location, which enables the system to display relevant information about the selected area.

1.1. Introduction

With the constant evolution of global interconnectedness, the potential of online social networking has greatly expanded. However, it is equally apparent that this vast digital landscape has led to scattered and disjointed information for users. One example of how this application is utilized is through the exchange of tickets and the sale/delivery of products.

UrbanHive aims to address these pressing issues. This platform, a social network designed for diverse residential communities such as neighborhoods, districts, and cities, has been envisioned to facilitate users through a plethora of features. Our main goal is to centralize all the ticket information, with the intention to reduce instances of profiteering.

Besides being a ticket exchange platform, UrbanHive provides a space for communities to interact, share, participate, and grow together. From displaying linked communities, public volunteer opportunities, electronic markets for buying and selling, and managing a schedule of events for each user, UrbanHive offers a social networking experience.

Unlike existing platforms where information is dispersed, UrbanHive aims to the users to serve as the source of information. This user-centric approach not only try to streamline the ticket exchange process but also try to strengthen community bonds and facilitates shared experiences.

UrbanHive, at its core, is a digital solution designed to cater to the evolving needs of ticket holders and seekers, while fostering a nurturing space for community interaction and growth.

As we move forward with UrbanHive, we invite stakeholders to participate in shaping this platform to ensure it is as user-friendly, efficient, and beneficial as possible. Our hope is that UrbanHive will not only try to solve the existing challenges in online ticket exchange but also transform the way residential communities interact and engage with each other online.

2. Background and Related Work

2.1. Ubiquity of Internet Connectivity

In contemporary society, internet access is nearly universal, particularly in developed countries. From smartphones and tablets to computers and other digital devices, the internet has become an integral part of everyday life. This reality has resulted in a significant growth in the use of social networks as platforms for communication, networking, and engagement. Moreover, individuals are becoming increasingly reliant on the internet to access products and services, favoring convenience and speed over traditional, physical means of acquisition.

2.2. Community Interactions and Marketplaces:

With the increased usage of the internet and social networks, several platforms have emerged offering users opportunities to interact and trade within specific communities or neighborhoods. The necessity for such localized online spaces can be attributed to the human desire for easy, quick, and secure transactions, be they for services or goods.

2.2.1 Prevalent Platforms Today:

2.2.1.1 Facebook Groups:

The leading social network globally, Facebook allows users to create groups tailored to their interests or geographical locations. Such groups often serve as local marketplaces or forums for communication within specific communities or neighborhoods.

2.2.1.2 Online Marketplaces:

Online platforms, such as Craigslist or eBay, offer users the ability to buy and sell a wide range of goods and services, including within specific geographical areas. These platforms are popular due to their wide reach and variety of listings.

2.2.1.3 Localized Apps:

Several localized apps have been developed to cater to specific neighborhoods or communities. These apps often include features such as community news updates, marketplace listings, event calendars, and more.

2.2.1.4 Word of Mouth:

Despite the surge in online platforms, word of mouth remains a valuable tool for local transactions. This is particularly true in tightly knit communities where trust plays a significant role in buyer-seller relationships.

2.2.2. Aspiration:

Against this backdrop, our project, UrbanHive, aims to address the limitations of existing platforms and enhance community interactions by developing a social network specifically tailored for diverse residential groups. This platform aims to streamline and secure the process of buying and selling goods, facilitating community interactions, scheduling events, and more. By integrating features such as user-friendly interfaces, secure transaction methods, and efficient community management, UrbanHive seeks to bring communities closer together in the digital age. This platform's objective is not just to provide convenience but to enhance the sense of community and neighborhood connectivity, thus redefining the social networking landscape.

3. Agile Development

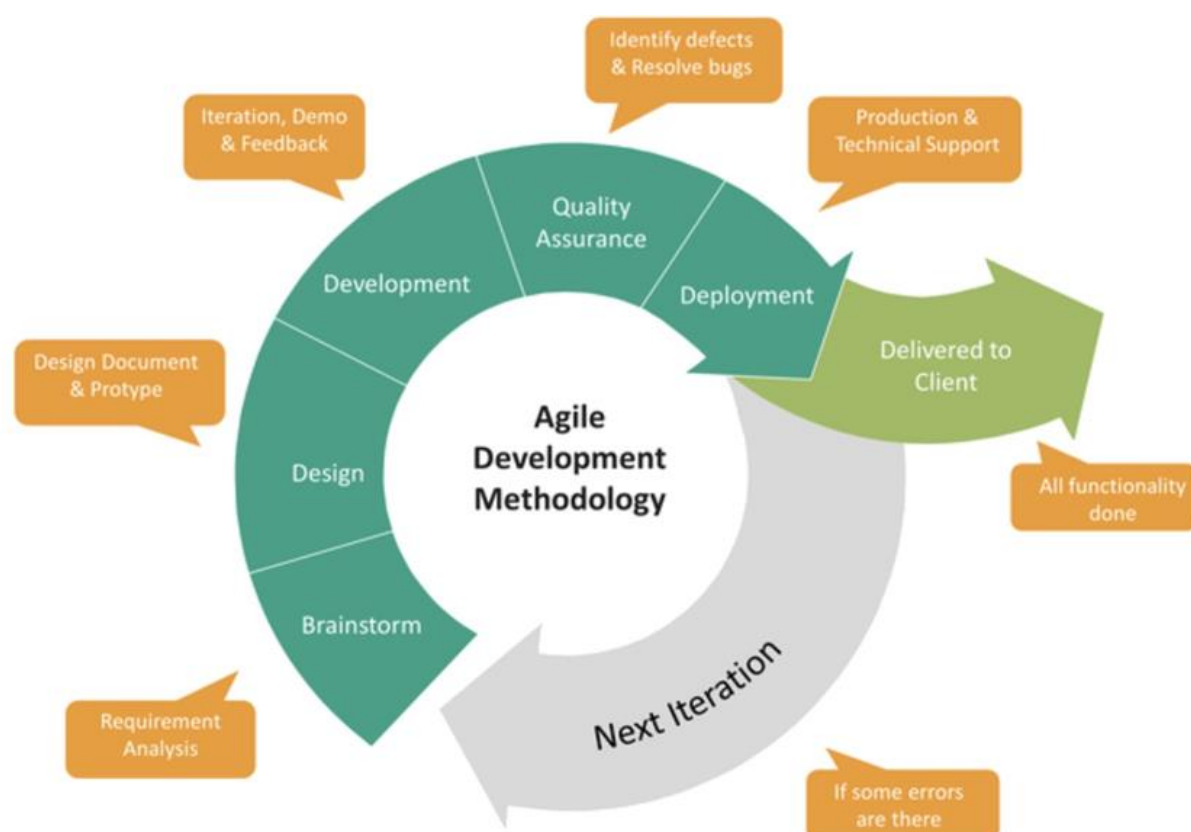
Agile development revolves around the convergence of people, processes, connectivity, technology, time, and place to devise the most efficient and suitable methodology for task completion. It functions within the task's guidelines, yet the approach to achieving it is boundary-free.

UrbanHive, a new social network designed for diverse residential groups, operates under the Agile development philosophy. It understands that the development of such a network is a fluid, empirical issue, not a matter of forecast or design. The development of UrbanHive is essentially treated as a goal-oriented collaboration game, responding flexibly to changes and feedback as they occur.

1. User Account Management - Agile approach emphasizes iterative and incremental progress. Hence, the system's user registration and login functionalities, including password recovery, will be developed, and tested in iterations, with feedback incorporated at each stage for continuous improvement.
2. User Operations - Post-login operations such as community interactions, calendar management, visibility settings, etc., will be developed following an Agile methodology, with incremental feature addition and consistent feedback incorporation.
3. Electronic Market - Agile methodology will be employed to handle the unique challenges of building an electronic market. Features like ad publication, chat request, and service scheduling will be developed iteratively, improving with every feedback loop.
4. Friend List Management - Friends' list functionalities will be implemented progressively, with every feature and change evaluated, tested, and integrated seamlessly into the system.
5. Activity Radius Determination - Agile principles will guide the development of user-specific activity area determination and its related features, with continuous iterations and improvements.
6. Community Management - Agile methodology encourages self-organizing teams with cross-functional skills. Thus, every aspect of community creation, management, and interaction will be designed to promote collaborative effort and collective ownership.

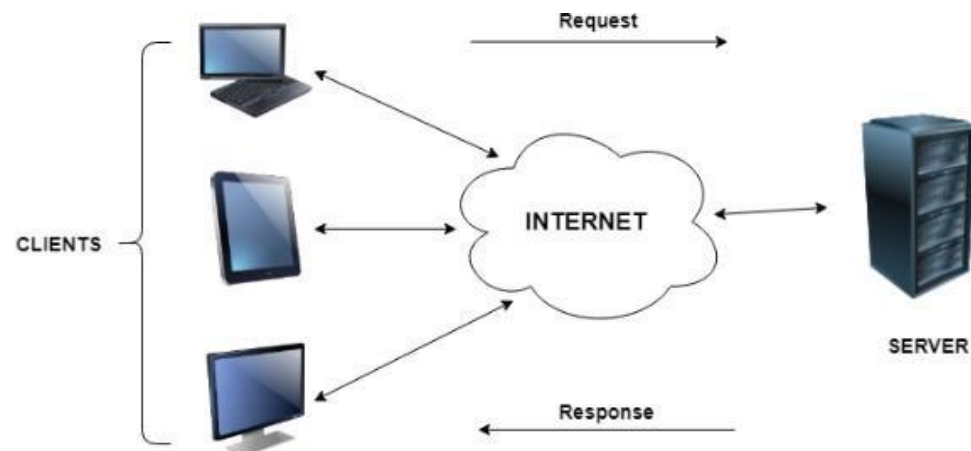
In essence, UrbanHive's development process aims to be 'agile' - swift, adaptable, and efficient. It treats software development as a collaborative game with the goal of building a comprehensive social network that caters to diverse residential communities. Through agile

methodology, we'll respond to changes swiftly, making UrbanHive a truly user-centric platform.



4. Client-server

UrbanHive utilizes a client-server model, an architecture that delineates the relationship within collaborative software in distributed computing. This model allocates tasks or workload between the server - the service or resource provider, and the client - the service requester. This prevalent model is fundamental in computer networks, with the server being the passive component that listens to the network, waiting to fulfill requests.



4.1. Understanding UrbanHive's Client-Server Model:

UrbanHive's architecture is based on a client-server model, a distributed framework that segregates duties between servers and clients. These entities may either exist within the same system or communicate over a computer network or the Internet.

In the context of UrbanHive, this model facilitates the transfer of information from the server to the user's device. It delineates the method by which devices access the vast amount of data stored on UrbanHive's servers. This system permits multiple users to launch applications or retrieve files from a single server, maintaining a uniform experience across all devices.

In essence, UrbanHive's client-server model operates as a data storage system. Much of the user-specific data and application configurations are stored on a remote server. When a user needs to access a file or utilize a feature within UrbanHive, the request is made to the server. The server then validates the request to ensure it originated from an authenticated device. After confirming the user's credentials, the requested information is downloaded to the user's device.

User devices can either share a network with their host server or connect through the internet. UrbanHive's client-server model primarily operates on a request/response pattern, utilizing a messaging system to make server requests. Communications between a client and a server follow specific protocols like TCP/IP.

The TCP protocol is chosen for distributing application data into packets, which are then transported to and received from the network. It also manages flow control and the retransmission of dropped or garbled packets. The IP protocol, being connectionless, treats each packet transferred as an independent data unit. Client requests are organized and prioritized in a scheduling system, enabling the server to respond to multiple distinct clients swiftly.

4.2. UrbanHive's Client-Server model uses:

The client-server model in UrbanHive enables the server to handle numerous requests concurrently using a scheduling system to prioritize messages from clients. This model enhances the overall functionality of UrbanHive by harnessing resources from various devices through a server.

4.3. UrbanHive's Client-Server Network:

UrbanHive's client-server network interacts with two types of networks:

1. LAN - Local Area Network
2. WAN - Wide Area Network

4.4. Advantages of UrbanHive's Client-Server Computing:

UrbanHive's server consolidates all essential data in a single place, enhancing data protection and user authorization management. Data can be accessed efficiently, regardless of the physical proximity between clients and the server. The data transferred through UrbanHive's client-server protocols is platform-agnostic, offering a seamless user experience across all devices.

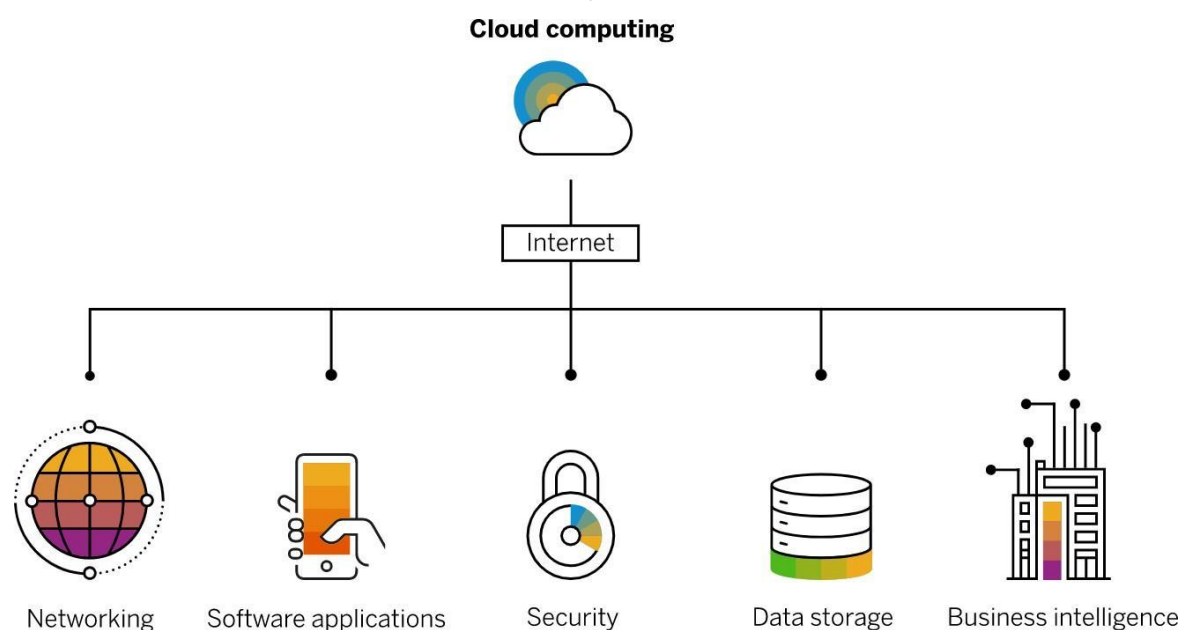
5. Cloud computing in UrbanHive

UrbanHive utilizes cloud computing, which is the provision of various services over the Internet. These resources encompass tools and applications such as data storage, servers, databases, networking, and software, which are integral to UrbanHive's operations.

UrbanHive doesn't rely on proprietary hard drives or local storage devices for storing files. Instead, it harnesses the power of cloud-based storage, which allows saving data to a remote database. This ensures that as long as a device has web access, it can retrieve the data and run the necessary software programs.

Opting for cloud computing has made UrbanHive a popular choice among its users. This can be attributed to several reasons, including cost-effectiveness, enhanced productivity, speed and efficiency, robust performance, and heightened security.

UrbanHive's implementation of cloud computing facilitates easy and on-demand access to a shared pool of computing resources via a communication network. Here, computing resources refer primarily to computing power, storage, and connectivity. UrbanHive ensures that the allocation and release of resources to users are executed swiftly and simply, requiring minimal administrative effort and an interface with the service provider. This streamlined process accentuates UrbanHive's commitment to user convenience and efficiency.



6. Expected Achievements

6.1. Goals:

1. Develop a comprehensive, user-friendly social networking platform for urban communities, focusing on local collaboration and engagement.
2. Provide a feature-rich environment that includes public volunteer opportunities, creation of communities, buying and selling of items or services, and event scheduling.
3. Enable users to create, join, and manage communities efficiently, with the ability to publish posts, schedule events, and manage regional guards.
4. Offer a robust mechanism for creating and managing ads in an electronic marketplace, with features like filtering ads and automated deletion of ads after a month.
5. Enable safe and secure collection of payments for community activities, with proper notification system for event cancellations.

6.2. Success Criteria:

1. UrbanHive is launched successfully and receives positive user feedback regarding ease of use, design, and functionality.
2. Active engagement within the communities, as indicated by the frequency and diversity of posts, comments, and participation in volunteer opportunities.
3. High utilization of the electronic marketplace, measured by the number of ads published and transactions made.
4. Successful collection of payments for community activities, with minimal issues or disputes.
5. Continuous growth in the user base, demonstrating the platform's appeal and usefulness.

6.3. Unique Features:

1. **Location-based Social Networking:** UrbanHive allows users to connect with their local community on various fronts - volunteering, events, or marketplace. The locality-based aspect of the platform is unique and brings people closer in a virtual environment.
2. **Electronic Marketplace:** UrbanHive aims to provide platform services within their community. This feature facilitates the circulation of resources within the community and promotes local economic growth.
3. **community management:** UrbanHive gives community creators extensive management powers, including the appointment of managers, post approval, and community rules enforcement.
4. **Regional Guard Management:** A unique feature that encourages community safety and security by managing local guard units within the community.
5. **Community Funding:** UrbanHive facilitates the collection of funds for community activities via external applications, thus enhancing community collaboration and project development.
6. **Meeting Scheduling and Interest-based Matching:** UrbanHive allows community members to schedule meetings based on shared interests, both online and offline, promoting deeper social connections within the community.

7. Research / Engineering Process

- 7.1. Requirement Gathering and Analysis: The first step is to thoroughly understand the project's requirements. All the details are extracted from the provided requirements document and analyzed thoroughly to comprehend the scope of the project.
- 7.2. Designing System Architecture: After a deep understanding of the requirements, the next step is to design the system's architecture. This includes deciding the software, hardware, system, and network requirements.
- 7.3. Development and Coding: In this phase, developers will start coding to implement the system as per the requirements and design. The system will be developed in modules and each module will be assigned to the respective teams.
- 7.4. Testing: After the development phase, the product will be tested. This includes Unit Testing, Integration Testing, System Testing, and Acceptance Testing. Any defects found during the testing phase will be logged and fixed.
- 7.5. Deployment: Once the product passes the testing phase, it will be deployed to the production environment.
- 7.6. Maintenance and Monitoring: After deployment, the product will be monitored for any issues. Regular maintenance will be done to keep the system up-to-date and to fix any issues if they occur.

8. Product

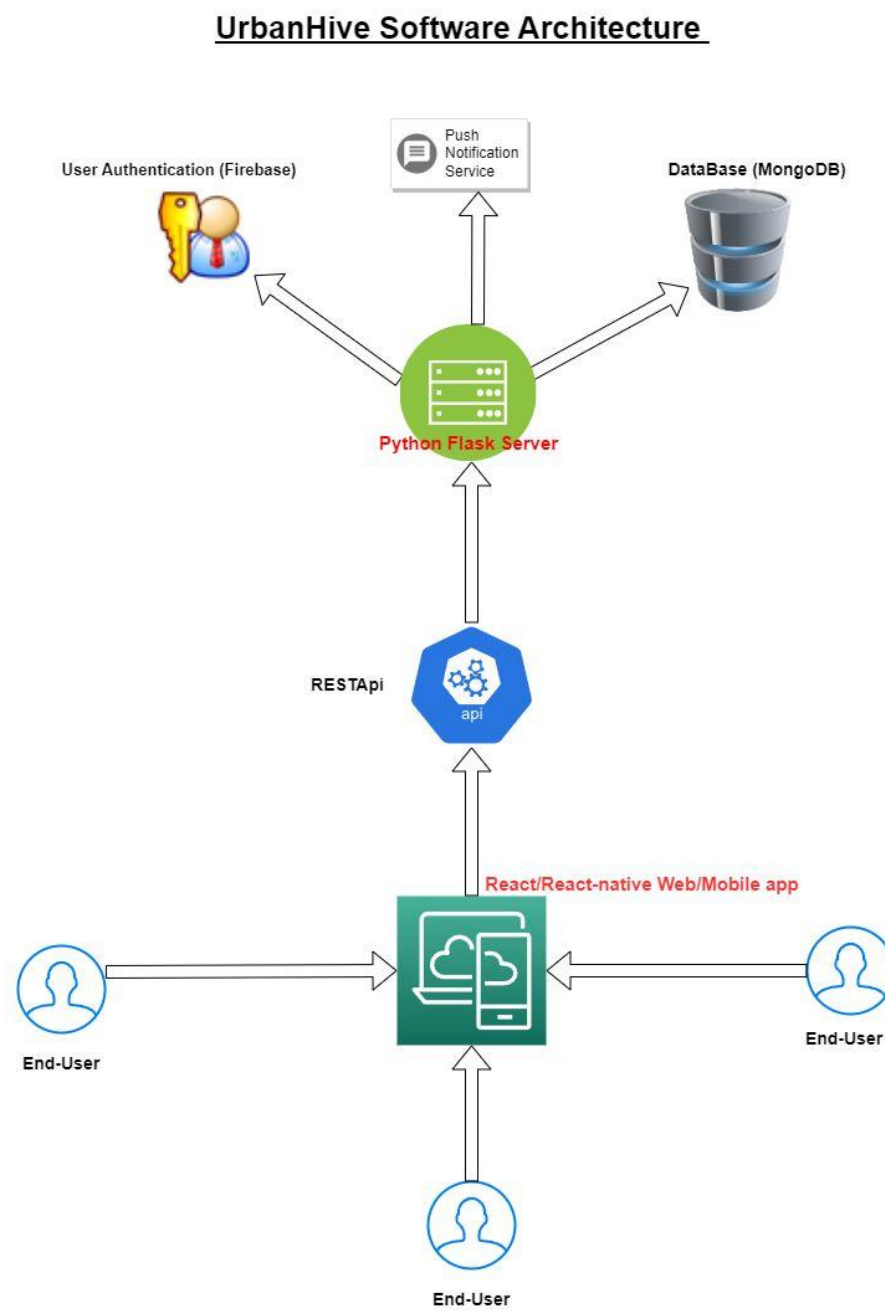
UrbanHive, a social network for residential communities, is designed to offer several features that enable interaction, communication, and community management. The product will be divided into several components, each implementing a specific use-case as mentioned in the requirements document:

- 8.1. User Authentication: This includes the functionalities for user registration, login, and password recovery.
- 8.2. User Operations: This component will handle various operations a user can perform after logging in. These operations include displaying linked communities, showing volunteer opportunities, managing user pages, managing friends list, event scheduling, and payment collections.
- 8.3. Advertisement Management: This component will handle the creation, publication, and management of ads in the electronic marketplace. It will also provide functionality for chat requests and transaction approvals.
- 8.4. Friend List Management: This component will deal with managing a user's friend list. It will handle friend requests, member availability status, and information sharing among friends.
- 8.5. Activity Radius Determination: This component will enable users to set a radius for their activity area. This will influence the information they receive.
- 8.6. Community Management: This comprehensive component will handle all the functionalities related to the creation, management, and participation in communities. This includes community creation, membership management, event scheduling, group chat management, regional guard management, etc.

These components will interact with each other and the central database to create a cohesive product. The product will be user-friendly and efficient, making community management and participation easier and more streamlined

9. Software Engineering Documents

9.1. Application Architecture



9.1.1.1. Technologies:

9.1.1.1.1. React/React Native for the Frontend:

React and React Native are popular frameworks for building user interfaces (UIs) using JavaScript or TypeScript. While react is primarily used for web development, React Native extends React's capabilities to build native mobile applications for iOS and Android platforms.

React:

- React is a JavaScript library developed by Facebook that allows developers to build reusable UI components. It follows a component-based architecture, where each component represents a part of the user interface.
- React uses a virtual DOM (Document Object Model) to efficiently update and render UI elements. It intelligently updates only the necessary parts of the UI when there are changes, resulting in improved performance.
- React is typically used for single-page applications (SPAs), where the UI is dynamic and interactive. It enables developers to create complex UIs by composing smaller components and managing state efficiently.

React Native:

- React Native is a framework built on top of React that enables developers to build mobile applications using JavaScript or TypeScript.
- With React Native, you can write code once and deploy it on both iOS and Android platforms. It achieves this by translating JavaScript code into native components, which allows for a truly native user experience.
- React Native provides a set of pre-built components that closely resemble native UI elements. These components can be combined to create a UI that feels native to the platform it runs on.
- In React Native, you can access platform-specific APIs and functionalities using JavaScript or TypeScript. This makes it possible to leverage device capabilities such as camera, geolocation, and push notifications.

JavaScript/TypeScript:

- React and React Native can be used with either JavaScript or TypeScript. JavaScript is a widely used programming language for web development, while TypeScript is a typed superset of JavaScript that adds static type checking and other features to improve code quality and maintainability.
- TypeScript brings type safety and allows for better code organization, especially in large-scale projects. It helps catch potential errors during development and provides better tooling support, including autocompletion and refactoring capabilities.

Overall, React and React Native, with the flexibility of JavaScript or TypeScript, offer powerful tools for developing web and mobile applications, respectively. They provide a declarative and efficient approach to building UIs, enabling developers to create interactive and native-like experiences with ease.

9.1.1.1.2. RESTApi:

A RESTful API (Representational State Transfer) is a widely used architectural style for designing networked applications. It serves as a communication interface between the frontend and backend of an application, allowing them to exchange data and perform operations.

Here's an elaboration on RESTful APIs and their role in connecting the frontend and backend:

1. RESTful Principles:

- Stateless: The server does not maintain any client state between requests. Each request contains all the necessary information for the server to process it.
- Client-Server Architecture: The frontend and backend are separate entities that communicate over HTTP/HTTPS using a standard set of operations.
- Uniform Interface: The API follows a standardized set of methods (HTTP verbs) such as GET, POST, PUT, DELETE, etc., for performing actions on resources.
- Resource-Based: Resources are identified by unique URIs (Uniform Resource Identifiers) and can be manipulated using the API's methods.
- Representation-Oriented: The server transfers representations of resources (e.g., JSON or XML) to the client, which can understand and process them.

2. API Endpoints:

- An API endpoint is a specific URL or URI that represents a resource or a collection of resources. It defines the location where clients can interact with the API.
- For example, `/users` can represent a collection of user resources, while `/users/123` represents a specific user with the ID 123.
- Endpoints are associated with HTTP methods to perform actions on the corresponding resources. For instance, a GET request to `/users` retrieves a list of users, while a POST request to `/users` creates a new user.

3. Request and Response:

- When the frontend needs to communicate with the backend, it sends HTTP requests to the appropriate API endpoint, specifying the desired method and resource.
- The request can include additional data in the form of query parameters, request headers, or a request body (for methods like POST or PUT).
- Upon receiving the request, the backend processes it, performs the necessary operations (e.g., fetching data from a database, updating records), and generates an HTTP response.
- The response typically includes a status code indicating the success or failure of the request (e.g., 200 for success, 404 for not found) and a response body containing data or additional information.
- The response body is often in a structured format like JSON or XML, which the frontend can parse and utilize to update the user interface.

4. Authentication and Authorization:

- RESTful APIs often require authentication to ensure that only authorized users or applications can access certain resources or perform specific actions.
- Authentication mechanisms like API keys, tokens, or OAuth can be used to validate the identity of the client making the request.
- Authorization mechanisms control what a particular authenticated user or client can do. This can involve roles, permissions, or access control lists (ACLs) associated with each resource.

By implementing a RESTful API, the frontend and backend can establish a standardized and flexible communication channel. The frontend can consume the backend's services by making HTTP requests to the API endpoints, and the backend can process those requests, retrieve or update data, and send back the necessary information. This decoupling between frontend and backend enables separate development and allows for scalability and interoperability.

9.1.1.3. Python Flask:

Python Flask is a lightweight and flexible web framework that can be used to build backend applications. Here's an elaboration on how Flask can be used to handle user authentication using Firebase, connect to a MongoDB database, and send push notifications to the frontend:

1. User Authentication with Firebase:

- Flask can integrate with Firebase Authentication to handle user authentication and authorization.
- You can use Firebase SDKs or Firebase Admin SDK in your Flask application to authenticate users using various methods such as email/password, Google, Facebook, etc.
- Flask routes can be protected with authentication middleware to ensure only authenticated users can access specific resources.
- Firebase provides features like user management, password resets, and token-based authentication that can be utilized in your Flask application.

2. Connecting to MongoDB:

- Flask can utilize various libraries like PyMongo or MongoEngine to connect to a MongoDB database.
- You need to install the required MongoDB driver and configure the connection settings in your Flask application.
- Once connected, you can define models or schemas to represent the data stored in MongoDB collections.
- Flask routes can handle database operations such as querying, inserting, updating, or deleting data using the appropriate MongoDB

library functions.

- These routes can fetch data from the database and send it back as a response or modify the database based on frontend requests.

3. Sending Push Notifications to the Frontend:

- To send push notifications to the frontend, you typically need to use a push notification service or a messaging platform such as Firebase Cloud Messaging (FCM) or OneSignal.
- You can integrate FCM or OneSignal into your Flask application to send push notifications to registered devices or users.
- When a certain event occurs in your backend, such as new data being added to the database or a specific action performed, you can trigger a push notification to be sent to the relevant devices or users.
- The Flask routes handling these events can utilize the appropriate push notification library or service to send the notifications with the required payload and target audience.

Flask's flexibility allows you to incorporate various third-party libraries and services like Firebase Authentication, MongoDB, and push notification platforms to build a backend that meets your specific requirements. By leveraging these tools within Flask, you can authenticate users, interact with databases, and send push notifications seamlessly to your frontend application.

9.1.1.4. MongoDB:

MongoDB is a popular NoSQL document-oriented database that can be easily integrated with a Python Flask server.

1. MongoDB Basics:

- MongoDB stores data in flexible, JSON-like documents known as BSON (Binary JSON).
- Each document can have a different structure, allowing for dynamic and schema-less data storage.
- MongoDB organizes documents into collections, which are analogous to tables in traditional relational databases.
- MongoDB supports rich query capabilities, indexing, and aggregation pipelines for efficient data retrieval and manipulation.

2. Integrating MongoDB with Flask:

- To connect your Flask server to MongoDB, you need to install the PyMongo library, which provides a Python driver for MongoDB.
- PyMongo allows your Flask application to interact with MongoDB using a straightforward API.
- First, you import the PyMongo library and create a MongoDB client instance, specifying the connection details (e.g., host, port).
- You can access a specific MongoDB database using the client's `get_database()` method, providing the database name.
- Once you have a reference to the database, you can work with its collections to perform CRUD operations (create, read, update, delete).

3. CRUD Operations:

- Creating: To insert data into a collection, you use the `insert_one()` or `insert_many()` methods, passing in the data in the form of Python dictionaries.
- Reading: You can query the database using the `find()` method, which retrieves documents based on specified criteria.
- Updating: The `update_one()` or `update_many()` methods allow you to modify existing documents by specifying filters and update operations.
- Deleting: You can remove documents from a collection using the `delete_one()` or `delete_many()` methods, specifying the filters to identify the documents to be deleted.

4. Mapping Data to Python Objects:

- PyMongo provides flexibility in how you handle data retrieved from MongoDB. You can map documents to Python objects using data models or schemas.
- You can define Python classes or use libraries like MongoEngine or Pydantic to define models that represent the structure and behavior of MongoDB documents.
- These models allow you to work with data in an object-oriented manner, simplifying data manipulation and validation.

5. Handling Relationships and Indexing:

- MongoDB supports various strategies for handling relationships between documents, such as embedding related documents within

each other or using references.

- You can create indexes on specific fields to improve query performance by using the `create_index()` method provided by PyMongo.

By integrating MongoDB with a Python Flask server, we can leverage the flexibility and scalability offered by a NoSQL document database. PyMongo provides an intuitive API for performing CRUD operations, allowing your Flask application to interact with MongoDB efficiently. With MongoDB's schema-less nature, you can adapt your data structures as your application evolves.

9.1.1.5. Auth Users:

Authentication is the process of verifying the identity of a user or entity to grant access to a system, application, or specific resources. It plays a crucial role in app security and user privacy. Here's an explanation of how authentication for users in an app typically works:

1. User Registration:

- The first step is to allow users to create an account or register in your app. This usually involves collecting necessary information such as username, email address, and password.
- The registration process may also include additional steps, like verifying email addresses through confirmation links or sending SMS verification codes.

2. User Authentication:

- Once registered, users need to authenticate themselves to access restricted areas or perform specific actions within the app.
- The most common method of authentication is through a username/email and password combination. The app compares the provided credentials with the stored values to validate the user's identity.
- Other authentication methods include social login (e.g., using Google or Facebook accounts) or biometric authentication (e.g., fingerprint or face recognition).

3. Security Measures:

- To ensure the security of user authentication, it is essential to implement several measures:
 - Password Storage: Store user passwords securely by using techniques like hashing and salting. Never store passwords in plain text.
 - Password Complexity: Encourage users to create strong passwords with a combination of uppercase and lowercase letters, numbers, and special characters.
 - Two-Factor Authentication (2FA): Provide an additional layer of security by implementing 2FA, where users need to provide a second form of verification, such as a one-time password (OTP) sent to their registered email or phone.
 - Account Lockouts: Implement mechanisms to lock or limit login attempts after a certain number of failed authentication attempts to prevent brute force attacks.
 - Session Management: Assign a session or token to authenticated users to maintain their logged-in state. This helps validate subsequent requests and ensures users stay authenticated during their session.

4. Authorization and Access Control:

- Authentication is distinct from authorization, which determines what resources and actions a user is allowed to access once authenticated.
- Implement authorization mechanisms to define user roles, permissions, and access control rules. This helps enforce fine-grained control over user actions and resource accessibility.
- Assign appropriate roles and permissions to users based on their privileges and responsibilities within the app.

5. Token-Based Authentication:

- Token-based authentication is a common approach where, upon successful authentication, the server generates a token (such as a JSON Web Token or JWT) and sends it back to the client.
- The client includes the token in subsequent requests as a means of authentication, typically in the request headers.
- The server verifies the token to ensure its authenticity and validity before granting access to protected resources.

6. Session-Based Authentication:

- Session-based authentication involves storing session information on the server after successful user authentication.

- The server issues a session ID or token, which is stored on the client-side as a cookie or in local storage.
- The client sends the session ID with each subsequent request, and the server verifies it against the stored session data.

Implementing robust and secure user authentication in your app is crucial for protecting user accounts and sensitive data. By following best practices and using secure authentication methods, you can provide a safe and trustworthy user experience while mitigating the risk of unauthorized access.

9.1.1.6. Push Notification:

Push notifications are a powerful feature in mobile app development that allow you to send messages or alerts to users even when they are not actively using the app. Here's an explanation of push notifications and how they are implemented in app development:

1. What are Push Notifications?

- Push notifications are messages sent from a server or backend to a user's device through a mobile operating system's push notification service (e.g., Apple Push Notification Service - APNs for iOS, Firebase Cloud Messaging - FCM for Android).
- These notifications appear as text-based banners, badges, or alert dialogs on the user's device, depending on the operating system and notification settings.
- Push notifications can be used to provide timely information, engage users, promote new features, deliver personalized updates, or send important reminders.

2. Implementing Push Notifications:

- To enable push notifications in your app, you typically need to follow these steps:
 - Register the app with the respective push notification service (e.g., APNs or FCM) to obtain the necessary credentials or keys required for sending notifications.
 - Integrate the push notification service's SDK or library into your app's codebase.
 - Request user permission to receive push notifications. This involves displaying an opt-in prompt to the user, explaining the benefits of notifications and how they will be used.
 - Handle the registration process to obtain a unique device token for each user's device. This token is used to identify the device and deliver notifications to it.
 - Store the device token on the server or backend associated with the respective user to send targeted notifications.

3. Sending Push Notifications:

- Once the user has granted permission and the device token is obtained, you can send push notifications from your server or backend using the push notification service's API or SDK.
- When sending a push notification, you specify the device token(s), the message content, and any additional data or actions associated with the notification.
- Push notifications can be sent immediately or scheduled for a specific time.
- Notifications can be customized with various options, including the notification title, body, badge count, sound, and actions.
- For personalized notifications, you can include user-specific data or attributes to tailor the content to each user.

4. Handling Push Notifications:

- On the client-side (the app), you need to handle received push notifications appropriately:
 - Implement notification handling logic in your app's code to handle different types of notifications and perform appropriate actions when a notification is received.
 - Display the received notification to the user by showing an alert, updating the app badge count, or playing a sound.
 - Handle user interactions with notifications, such as tapping on a notification to open a specific screen or perform a predefined action.
 - Manage the app's behavior when a notification is received while the app is in the foreground or background.

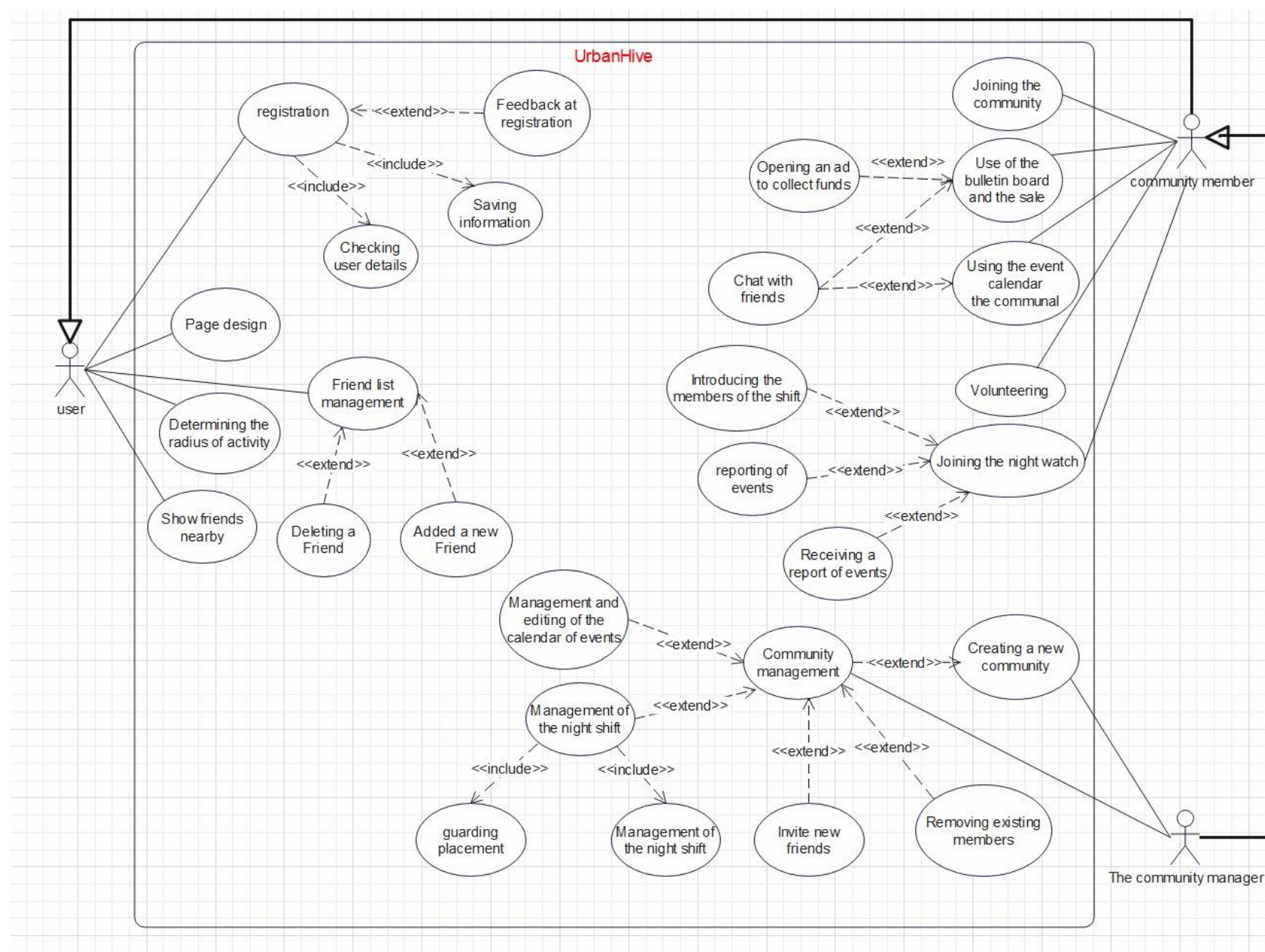
Push notifications are an effective way to engage users and provide real-time updates, ensuring your app remains relevant and connected with users even when they are not actively using it. Proper implementation and thoughtful use of push notifications can enhance user experience and drive user engagement in your app.

10. Diagrams

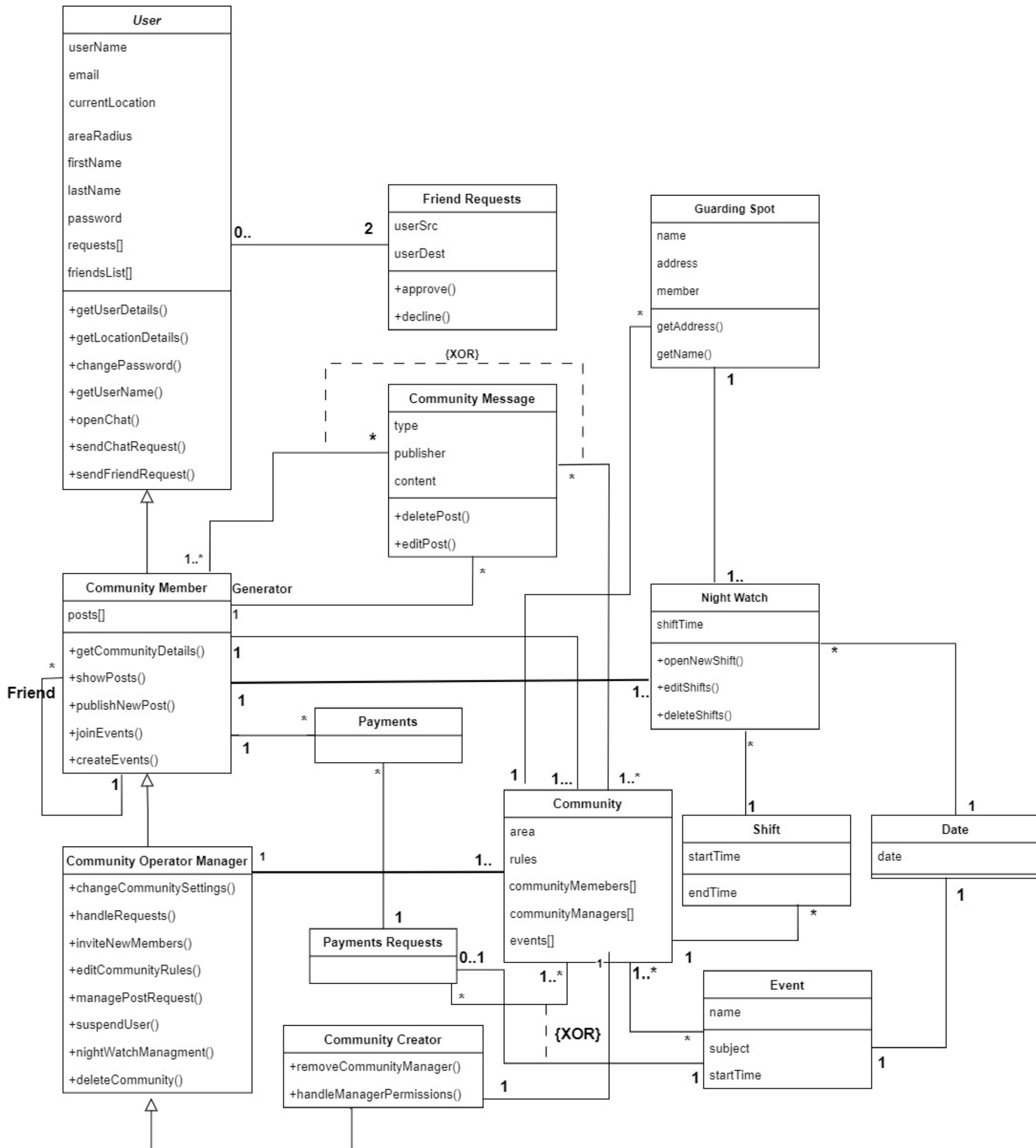
10.1. Use-Case Diagram

Actor Descriptions:

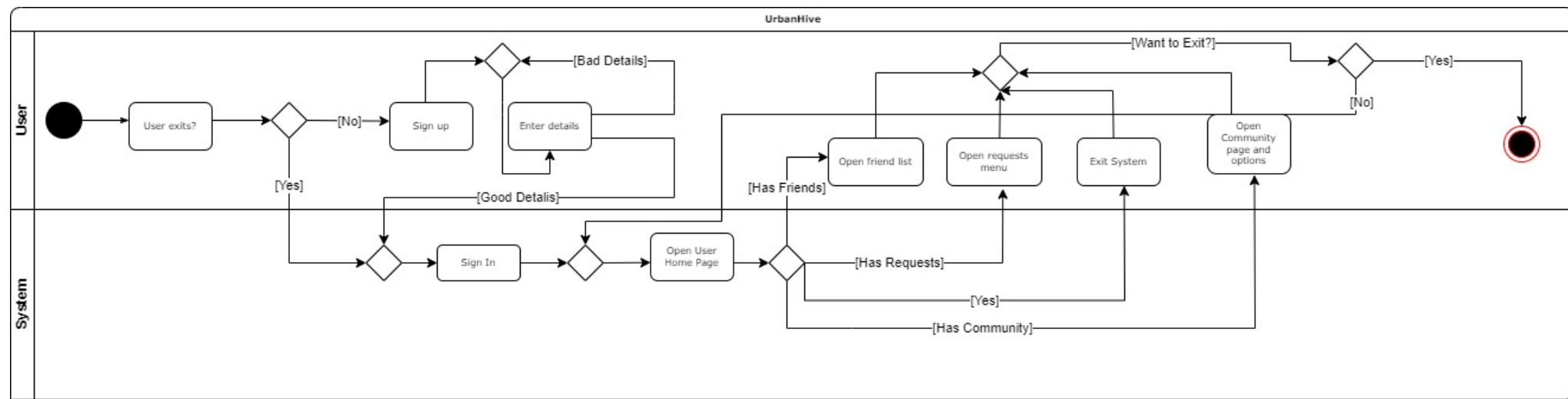
1. **User:** A User is a registered individual on UrbanHive who can access their unique account to perform various actions including creating or joining communities, managing their profile, adding friends, and determining their activity radius. Users can also contribute to fundraising initiatives, post on community boards, and participate in community events or activities. Users can also request to join a community and once approved, they can view and comment on posts, publish posts (according to community rules), and manage the community's event calendar.
2. **Community Member:** A Community Member is a User who is part of a specific community within UrbanHive. As a member, they have been approved to join a community by the Community Manager or the creator of the community. They can view and comment on posts within their community, publish posts according to community rules, participate in community events, access the community events calendar, and contribute to the community's fund collections. They can also participate in regional guard management activities and link with other community members for social or gaming activities.
3. **Community Manager:** A community Manager is a User who has been appointed by the creator of a community to help manage and moderate that community. They have the authority to change group settings, approve join requests, invite people to the community, modify community rules, and remove members who violate these rules. They can also manage community events and group chats, approve or disapprove posts before they are published, and contribute to managing regional guards. They are responsible for defining the "community house" and have the power to remove administrators from their management positions if necessary. They are also involved in fundraising efforts and can make decisions regarding post publishing frequency and the visibility of posts.



10.2. Class Diagram

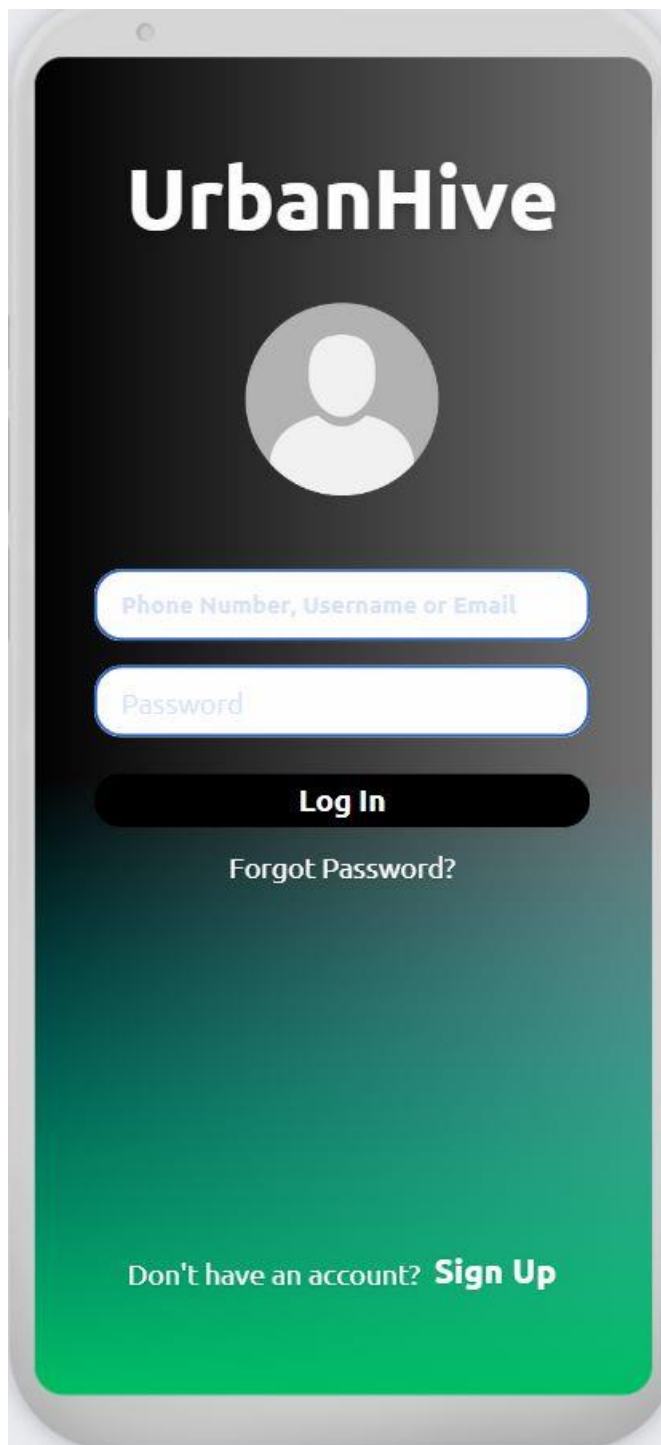


10.3. Activity Class (Login/signup ->Home Screen->Functions->Exit System)

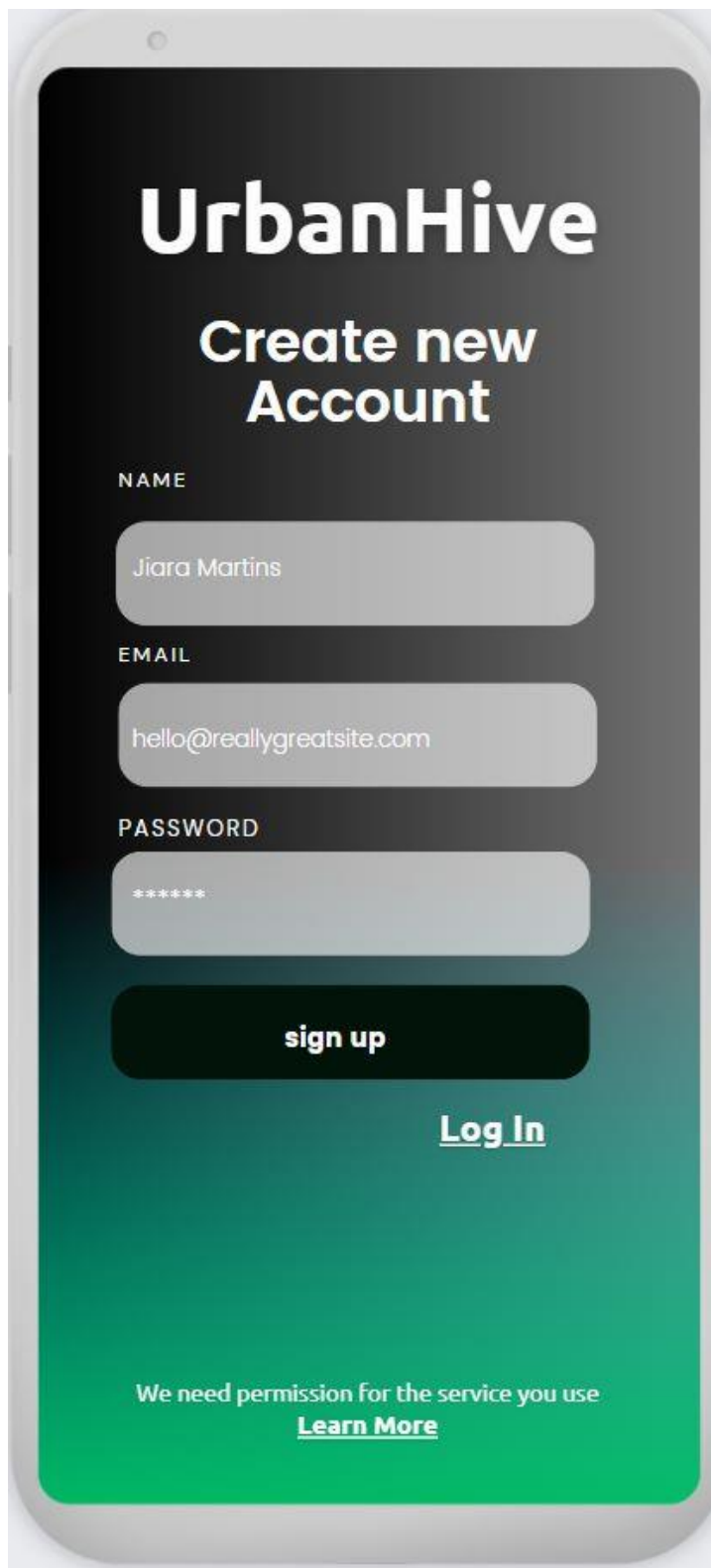


11. GUI Design

Login:



Create account:



The image shows a mobile app interface for UrbanHive. The screen has a dark grey header with the 'UrbanHive' logo in white. Below the logo, the text 'Create new Account' is displayed in white. The form consists of three input fields: 'NAME' with the value 'Jiara Martins', 'EMAIL' with the value 'hello@reallygreatsite.com', and 'PASSWORD' with the value '*****'. A black 'sign up' button is positioned below the password field. At the bottom of the form, there is a 'Log In' link. The background of the app is a green-to-white gradient. At the very bottom, a small text line reads 'We need permission for the service you use' followed by a 'Learn More' link.

UrbanHive

Create new Account

NAME

Jiara Martins

EMAIL

hello@reallygreatsite.com

PASSWORD

sign up

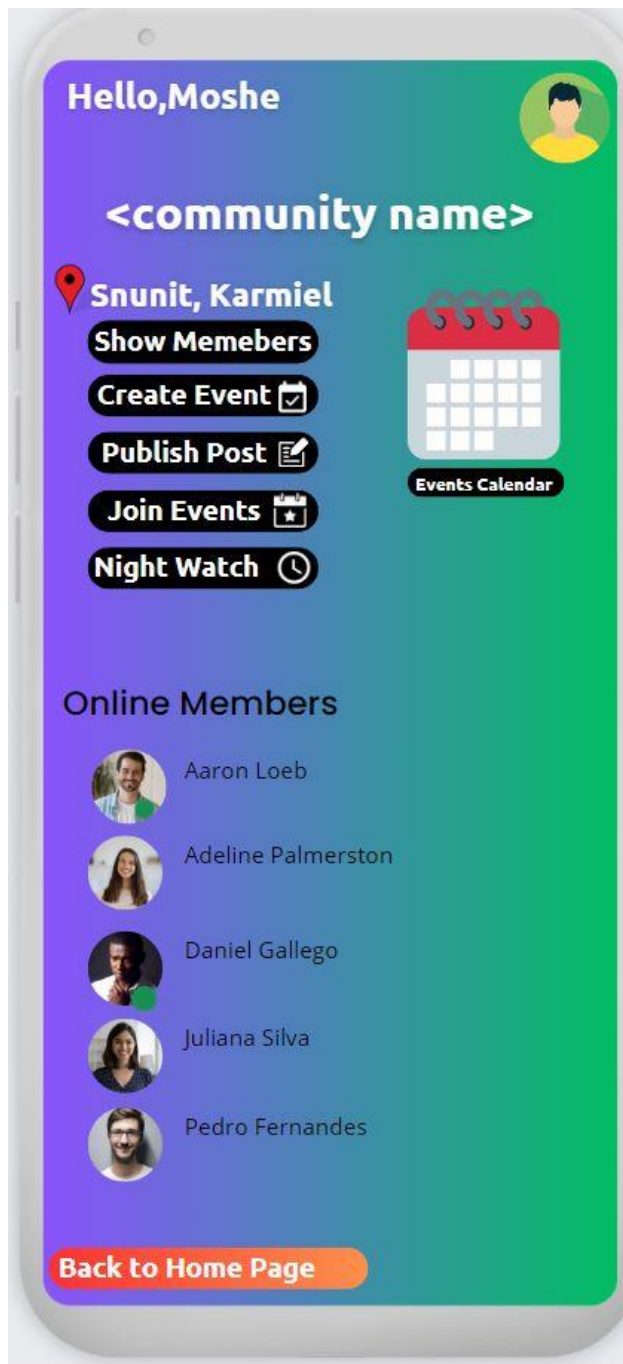
[Log In](#)

We need permission for the service you use
[Learn More](#)

Home Screen:



Community menu:



12. Evaluation / Verification Plan

- 12.1. **Unit testing** - a method of testing individual components or units of a software application to ensure they function correctly and independently of other components.

Table Name	Test-case ID	Description
users.test.js	Test-case 1: User Registration	Test user registration with valid inputs. Test user registration with invalid inputs. Test user registration with already used email.
users.test.js	Test-case 2: User Login	Test user login with valid credentials. Test user login with invalid credentials. Test user login with email and password.
users.test.js	Test-case 3: User Password Recovery	Test password recovery with valid email or phone number. Test password recovery with invalid email or phone number. Test password recovery with unregistered email or phone number.
users.test.js	Test-case 4: User Profile Management	Test editing of user details.

Table Name	Test-case ID	Description
communities.test.js	Test-case 1: Community Creation and Management	Test community creation with valid details. Test community creation with invalid details. Test assignment of community managers.
communities.test.js	Test-case 2: Community Membership	Test sending of membership request. Test receipt of membership request. Test cancellation of membership request.
communities.test.js	Test-case 3: Community Activities	Test management of community events calendar. Test creation and participation in joint events.

Table Name	Test-case ID	Description
friends.test.js	Test-case 1: Friend List Management	Test addition of friends. Test sending of membership request. Test receipt and cancellation of membership requests.

Table Name	Test-case ID	Description
radius.test.js	Test-case 1: Activity Area Radius	Test setting of activity area radius. Test display of user activity areas.

Table Name	Test-case ID	Description
payments.test.js	Test-case 1: Payments and Collections	Test creation of posts for fund collection. Test payment as part of fund collection. Test refunding or retaining of funds in case of event cancellation.

Table Name	Test-case ID	Description
UrbanHive	Community Creation & Data Types Check	Create a new user -> Create a new community -> Add user to the community -> Check the community's member list -> Remove user from the community -> Verify the user is no longer in the community's member list.
UrbanHive	Event Management	Login as a community member -> Create new event -> Add participants to the event -> Find event by filter -> Find participant by filter -> End the event -> Check event history.
UrbanHive	User Role Management	Create a new user -> Promote the user to be community manager -> Promote another user to community manager -> Demote that user from community manager to standard user -> Verify user roles are updated correctly.
UrbanHive	UI Check	Create a new user -> Create a new community -> Add user to the community -> Verify that all changes are accurately reflected in the UI -> Remove user from the community -> Verify that the UI updates correctly.
UrbanHive	User Profile Updates	Login as a user -> Update user profile details (photo, bio, interests, area) -> Logout -> Login as the same user -> Verify that the updated details are accurately displayed.

Table Name	Test-case ID	Description
UrbanHive	Performance Test	Load Test: Simulate high traffic load on the system and verify its response times and stability. For example, simulate 1000 simultaneous user logins and ensure the system maintains acceptable performance.
UrbanHive	Scalability Test	Verify the system's ability to scale up and handle an increasing number of users or communities. For example, add 1000 new communities and ensure the system can handle this increase.
UrbanHive	Compatibility Test	Verify that the system works across various platforms (Windows, MacOS, Linux) and browsers (Chrome, Firefox, Safari, Edge). Test the system's mobile compatibility on different devices (iOS, Android) and screen sizes.
UrbanHive	Usability Test	Evaluate the system's user interface for intuitiveness, efficiency, and ease of use. Test the accessibility of the system for users with disabilities.
UrbanHive	Localization Test	If the system is expected to support multiple regions, verify that the localization features work correctly.
UrbanHive	Resilience Test	Test how the system handles errors or unexpected inputs. For example, what happens if a user tries to create an account with an existing email address.

13. References

Haroon S. (Feb. 2014). Client-server model. *IOSR Journal of Computer Engineering (IOSR-JCE)*

https://www.researchgate.net/profile/Shakirat-Sulyman/publication/271295146_Client-Server_Model/links/5864e11308ae8fce490c1b01/Client-Server-Model.pdf

Client Server

<https://www.omnisci.com/technical-glossary/client-server>

Agile Development

<https://www.guru99.com/agile-scrum-extreme-testing.html>

React Native

https://en.wikipedia.org/wiki/React_Native

React:

[https://en.wikipedia.org/wiki/React_\(software\)](https://en.wikipedia.org/wiki/React_(software))

Flask Python

[https://en.wikipedia.org/wiki/Flask_\(web_framework\)](https://en.wikipedia.org/wiki/Flask_(web_framework))

MongoDB

<https://en.wikipedia.org/wiki/MongoDB>

Authentication

https://en.wikipedia.org/wiki/Authentication_server

Push Notification

https://en.wikipedia.org/wiki/Push_technology