College of Engineering, karmiel

Capstone Project Phase B

Project number

23-2-D-14

UrbanHive

Find your community!

Submitters:

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

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

Supervisor:

Prof. Zeev Brazily: zbarzily@braude.ac.il

Date: May 09,2024

Abstract

UrbanHive emerged from the innovative concept of utilizing digital technology to create and manage night watches within communities

This foundation inspired the expansion of UrbanHive into a comprehensive social platform aimed at fostering interconnectedness within residential areas.

The app now serves not only as a tool for enhancing community safety through night watch coordination but also as a hub for promoting social engagement.

Key functionalities include displaying public volunteer opportunities, organizing community events, enabling member-to-member chat, and managing community groups.

UrbanHive integrates features such as user authentication, personal account management, and region-specific content filtering, offering a tailored experience that encourages community involvement and support.

1 Introduction

Communities in the digital age encounter difficulties creating connected and safe spaces.

Among these, the absence of a specific platform for planning and supervising night watch shifts in locations with elevated security concerns is among the most serious problems.

Due to such a lack, people are now forced to depend on fragmented and inefficient methods like WhatsApp chats and Facebook groups, which are incapable of providing the necessary characteristics required for exact location, administration, and area requirements.

1.1 Problems

- The absence of an organized system for managing night watch operations in areas with significant security concerns creates a gap in community safety and coordination.

- New neighbors and families moving into unfamiliar communities' face challenges in connecting with local happenings, volunteer opportunities, and understanding neighborhood dynamics, which affects their sense of belonging and decision-making about the move.

- Traditional social media platforms offer limited solutions, lacking the specificity and efficiency needed for effective organization of night watch activities and for fostering community engagement. This results in safety vulnerabilities and decreased community involvement.

1.2 Solution

The UrbanHive project was started in response to the need for a comprehensive framework that could manage night watch shifts and promote community integration.
As development continued, it became clear that UrbanHive could have a much wider impact than just keeping residents safe.

UrbanHive has developed into a flexible tool that not only makes organizing night watch operations less difficult but also serves as a doorway for community involvement by providing features like volunteer opportunities, fundraising events, and a thorough rundown of nearby events and projects.
New residents can find opportunities to engage and contribute, learn about local events, and interact with other residents through UrbanHive, which not only improves safety but also enhances the community experience overall. With UrbanHive, immigrants can blend into society with ease.

1.3 Application Stakeholders

Community Members:

Residents of the communities UrbanHive serves, including new neighbors and families who have recently moved or are considering moving to a new area. They are the primary users of the app, engaging in night watch activities, participating in events, and utilizing the platform for social engagement and local information.

Community Managers:

Individuals who take active roles in organizing and managing community groups, events, and night watch schedules.

They use UrbanHive for coordinating these activities, communicating with members, and fostering a sense of community.

2 Solution Description

Using a digital platform, the UrbanHive responsibility utilizes an innovative, scalable technology stack to promote safety and community involvement.

Secure authentication, reliable data management, and client-server communication may all be integrated effortlessly with this architecture.

A comprehensive summary of the elements utilized in the UrbanHive project is provided here, with special attention paid to the client application, server environment, communication protocol, database system, and authentication method.

2.1 Software Architecture

2.1.1 User

Users' interaction with the application include creating and logging into their accounts securely, connecting by adding friends, and viewing community members.

The platform enables community leaders and members to create and participate in events, share updates and announcements through posts, and actively engage by committing to posts or volunteering activities.

Additionally, UrbanHive facilitates the organization and participation in night watch shifts, directly contributing to the community's safety and security.

2.1.2 Client App Expo React Native and JavaScript

Facebook's open source React Native framework is used in the development of UrbanHive's client side.
With React Native, developers can use a single JavaScript codebase to create natively rendered iOS and Android mobile applications.

This methodology enables the quick creation and implementation of cross-platform mobile applications by focusing on the extensive JavaScript ecosystem and the declarative user interface paradigm of the React package.

Expo, a framework and platform for universal React applications that simplifies the React Native development process by providing a set of tools and services that make it easy for app creation, testing, and deployment.
It eliminates the need for native code compilation, enabling developers to focus on writing JavaScript while Expo takes care of the complexity of building and managing the native part of the application.

The component-based architecture of React Native makes it easier to create dynamic and responsive user interfaces, which enhances the user experience.

2.1.3 Server: Python Flask

UrbanHive uses Flask, an efficient and flexible micro web framework built in Python, for the server-side logic. Flask offers simplicity, scalability, and smooth control over its components, enabling developers to build a solid web application backend with the required resources and capabilities. Flask was selected for the server environment due to its quick RESTful API development, filled documentation, and ease of use. Python is a great option for managing server-side operations and interfacing with other technologies like databases and authentication services because of its extensive library and Flask's simple design.

2.1.4 REST API as the communication protocol

REST (Representational State Transfer) APIs allow the UrbanHive project's client and server to communicate with each other.

The REST design supports GET, POST, PUT, and DELETE actions and uses HTTP requests to simplify communication between the client and server. These functions enable the client application to communicate with the server and make changes to the database. They are similar to fundamental CRUD (Create, Read, Update, Delete) functionalities.

The efficiency and reliability of the system improve overall when REST APIs are used because they provide a stateless, cacheable, and scalable system in which every request made by the client to the server has all the data needed for processing it.

2.1.5 MongoDB is the database system

The UrbanHive project uses MongoDB, a NoSQL database, to manage its data.

High performance, high availability, and simple scalability are the architectural objectives of MongoDB. Data is stored in flexible documents that resemble JSON, so the data format can change over time.

The UrbanHive project benefits from MongoDB's schema-less structure, which can handle the various and changing data requirements of community management, event planning,user interactions and night watch management.

2.1.6 Auth0 for authentication

UrbanHive uses Auth0, a flexible cloud-based identity management platform that offers secure access to services and apps, for authentication.

Extensive authentication and authorization capabilities, such as social login, multi-factor authentication, and user management, are available with Auth0.

Through the integration of Auth0, UrbanHive provides a safe and easy authentication process for its users, protecting private data and increasing platform trust.

Figure 01: A software architecture diagram of UrbanHive

2.2 Client Architecture

The UrbanHive project's React Native architecture follows a design pattern that combines the **MVC (Model-View-Controller)** pattern with **functional programming** and component-based paradigms. This structure supports a reliable and effective development process through:

1. **Modularity**: By encapsulating functionality, components simplify navigation, maintenance, and updateability inside the codebase. Every screen or function, like the ability to create events or list communities, is a stand-alone module.
2. **Reusability:** Common user interface elements are separated into reusable components, allowing for easier updates, simplified development, and consistent user interfaces across the application.
3. **State Management:** State management is achieved through a combination of React's stateful components and the Context API, ensuring UI elements and underlying data are correctly synchronized.
4. **Lifecycle Management**: Lifecycle methods and hooks give developers access to critical points in a component's life, providing full control over how the component acts and performs.
5. MVC Pattern: The application's structure demonstrates a clear separation of concerns:
    - The Model consists of API functions housed in apiUtils.js, managing interactions with the server.
    - The View is the JSX files in the screens folder for each screen, which directly renders UI components and leverages data from the Model.
    - The Controller role is embedded into the view (JSX files) , directly connecting the Model's data-fetching functions to the rendered UI, managing the application's flow.

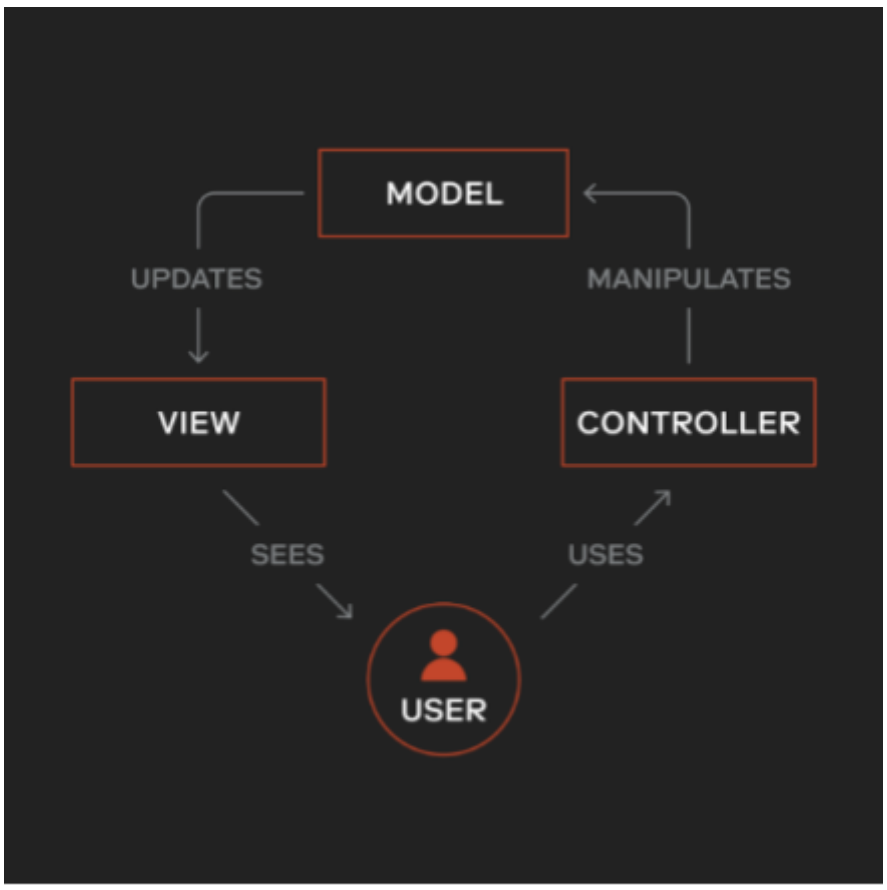

Figure 02: MVC Design Pattern diagram

React Native Rendering:

The React Native rendering architecture consists of three primary phases[3]:

1. **Render:** JavaScript-based React components are converted into C++ React Shadow Tree structures, which run on both Android and iOS.
2. **Commit:** During this step, components are laid out and prepared for rendering.
3. **Mount:** Components are rendered to the screen as native user interface elements.

This architecture, with Expo's tools, optimizes performance through asynchronous execution and reduces direct manipulation of the UI thread. Expo's development framework also provides an efficient way to build and deploy React Native applications by:

- Offering a comprehensive set of modules and libraries for common app functionalities, reducing the need for external dependencies.
- Allowing seamless deployment to both Android and iOS through Expo Go and EAS (Expo Application Services), simplifying the development process.
- Ensuring that React Native's rendering and execution flow remain smooth and uninterrupted.

In summary, the UrbanHive project's architecture integrates various design patterns and development tools to create a modular, reusable, and efficient client application, maximizing productivity and performance across all phases.

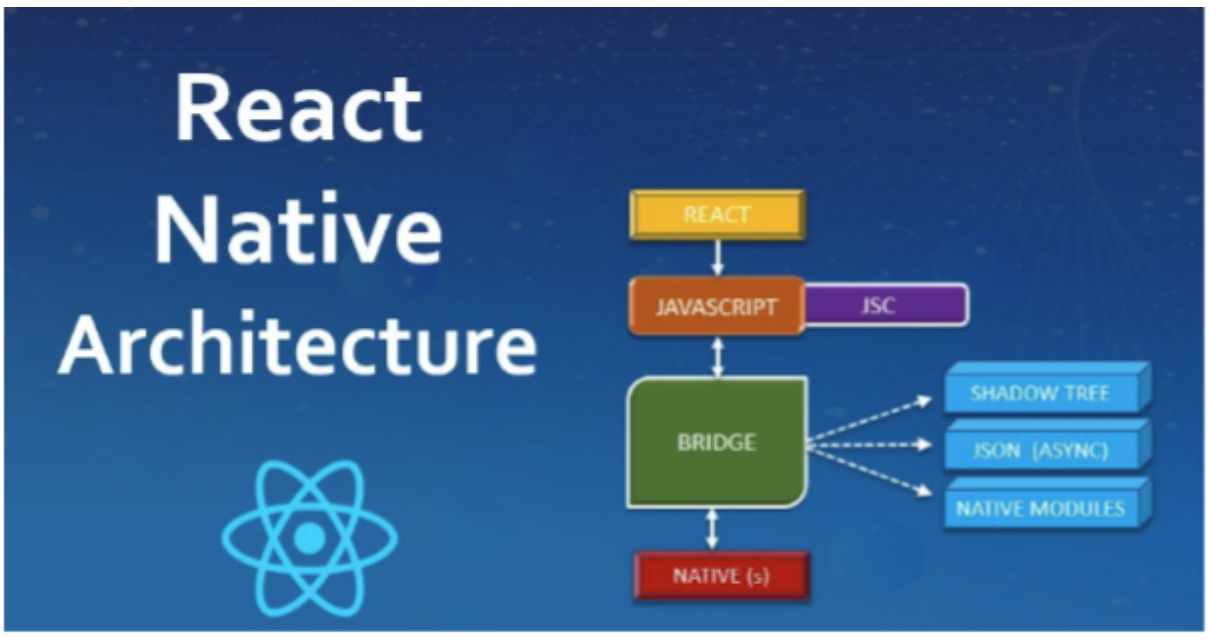

Figure 03 : React-Native rendering process diagram

2.2.1 Client structure

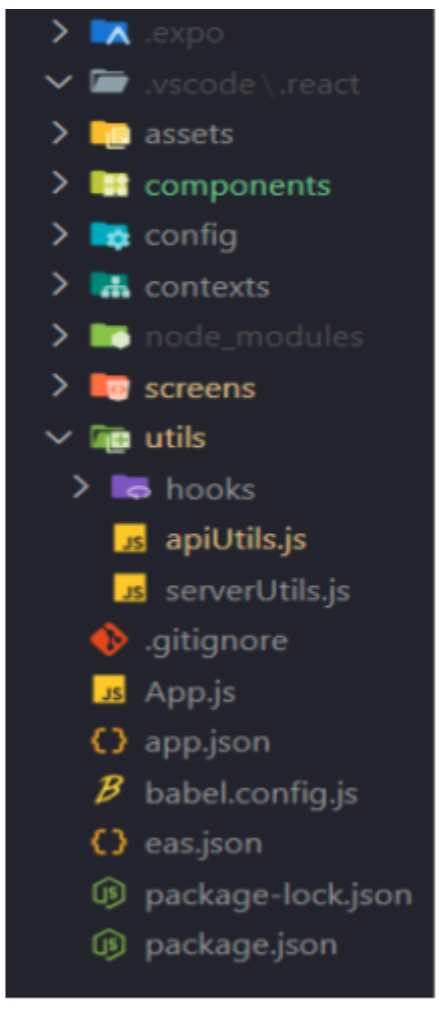

Figure 04 : Client app root folder structure

- **assets**: Stores static files such as images, fonts, and any other assets required by the application.
- **components:** A directory that includes reusable UI components for the project, such as modals, buttons, and input fields. This allows for modular and maintainable code.

- **config**: includes various configuration files that used to define global settings and environment-specific variables.
- **contexts**: Contains React context files which are used for managing state in a more global scope, enabling different components to access shared data.
- **node_modules**: Stores third-party libraries and dependencies for the project.
- **App.js**: The main entry point of the application, where the root component is defined.
- **app.json**: Includes metadata about the application such as the name, version, and entry point.

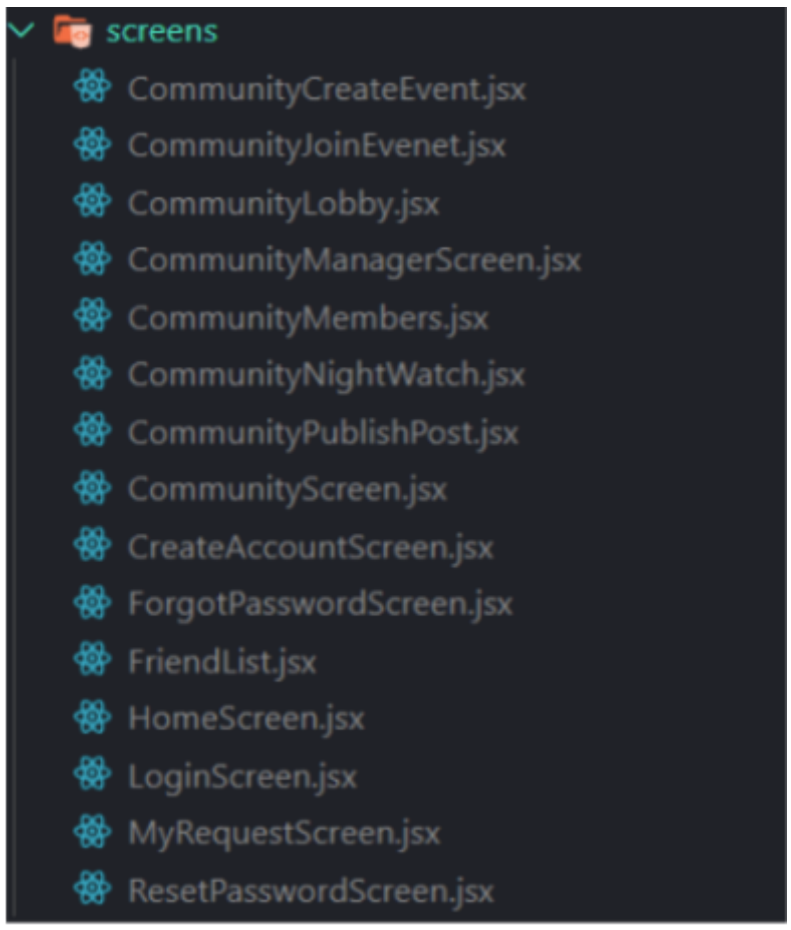

Figure 05: Client Screen folder structure

- **screens**: serve as the View pattern and Includes files for each of the screens or views of the application. Each screen file, such as CommunityCreateEvent.jsx or LoginScreen.jsx, represents a different page within the app, encapsulating the UI and logic for that particular part of the user experience.

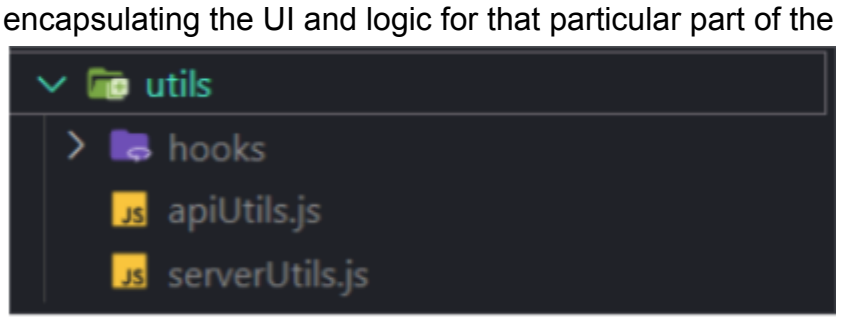

Figure 06: Client Utils folder structure

- **utils**: Contains utility scripts and functions that are used across the application to perform common tasks. Within this folder, there's a hooks subfolder in which custom React hooks are used for logic reuse, with hooks like apiUtils.js handling API (server as the view pattern) calls and serverUtils.js managing server interactions.

2.3 Server Architecture

The Python Flask server for the UrbanHive project utilizes a RESTful architecture to manage entities including users, communities, posts, events, and night watch. This server also employs a **Service Layer** design pattern, facilitating a structured approach to managing its various functionalities:

- **Service Layer:** For each entity (users, communities, posts, events, and night watch), a distinct set of endpoints is defined, which provides an interface to various actions:
  - Retrieval: via GET
  - Creation: via POST
  - Updating: via PUT or PATCH
  - Deletion: via DELETE
- The service layer handles the business logic for these actions and communicates with the database.
- Database Integration: Every entity is stored in its collection in the MongoDB database, with which the server interacts through the service layer. This allows for seamless storage and querying of data, making MongoDB a good fit for the structured and dynamic data that the application's various features handle.

- Logging and Testing: The service layer is supported by a logging system that tracks each function's activities, aiding in debugging and monitoring. Additionally, a comprehensive testing suite ensures the reliability and functionality of the RESTful APIs.
- Flask Framework: Flask's lightweight and modular design makes it an excellent choice for developing RESTful APIs, providing a clear interface for interacting with and manipulating data in this complex application.

2.3.1 Server Structure

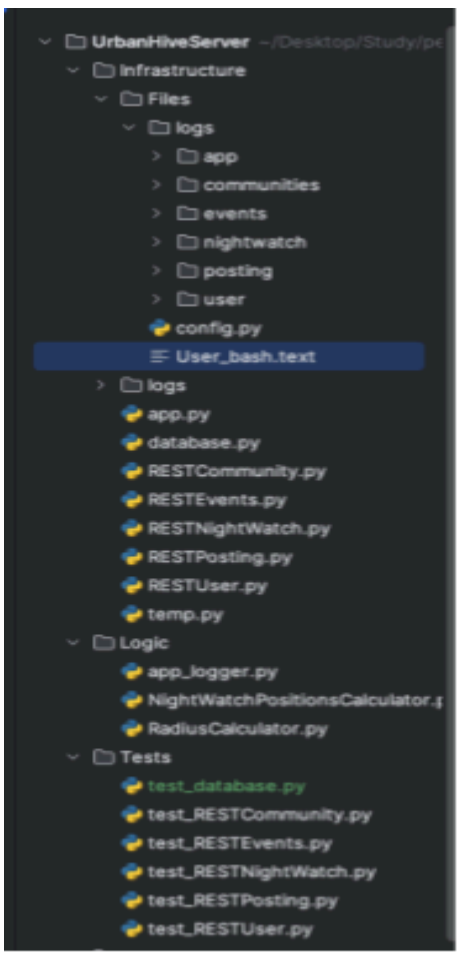

Figure 07: Server root folder structure

Infrastructure:

- **Files:** Contains various files and directories:
  - **logs**: Stores log files for each RESTful API route, aiding in monitoring and debugging:
    - **app:** Logs for the main application logic.
    - **communities:** Logs for community-related operations.
    - **events:** Logs for event-related functionalities.
    - **nightwatch:** Logs for night watch sessions.
    - **posting:** Logs for post-related operations.
    - **user:** Logs for user-related operations such as registration and authentication.
  - **User_bash.txt:** Stores examples of cURL requests for all the endpoints.
- **app.py:** The main entry point for the Flask application.
- **database.py:** Handles database connections and operations, interfacing with MongoDB.
- **RESTCommunity.py:** Defines RESTful API routes for community-related operations.
- **RESTEvents.py:** Manages API endpoints for event-related functionalities.
- **RESTNightWatch.py**: Contains the API routes for managing night watch sessions.
- **RESTPosting.py:** Deals with endpoints for creating, reading, updating, and deleting posts.
- **RESTUser.py:** Provides API routes related to user management such as registration and authentication.

Logic:

- **app_logger.py:** Responsible for logging and tracking events within the application, aiding in debugging and monitoring.
- **NightWatchPositionsCalculator.py:** Contains logic for calculating and managing positions for the night watch system.
- **RadiusCalculator.py**: A utility to calculate distances or geographical radius, for defining the scope of community events or night watches.

Tests:

- **test_RESTCommunity.py:** Contains unit tests for community-related REST endpoints.
- **test_RESTEvents.py:** Contains unit tests for event-related REST endpoints.
- **test_RESTNightWatch.py:** Contains unit tests for night watch-related REST endpoints.
- **test_RESTPosting.py:** Contains unit tests for post-related REST endpoints.
- **test_RESTUser.py:** Contains unit tests for user-related REST endpoints.

2.4 Database Structure

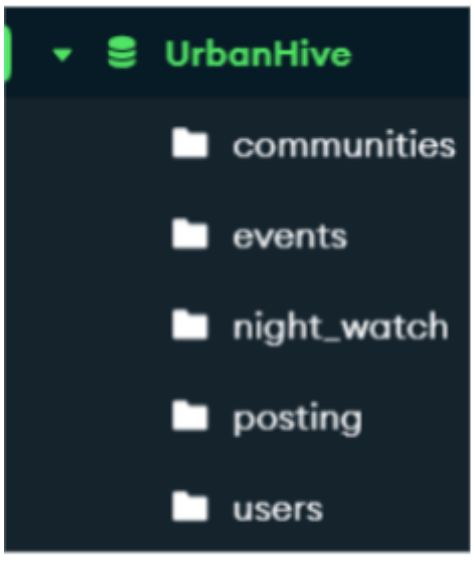

Figure 08: MongoDB collections structure

2.4.1 Communities Collection

- **Purpose**:Stores information about different community groups within the app, including location, members, and activities.
- Key Fields:
  - **id**: Unique identifier for each document in MongoDB.
  - **community_id**: A unique UUID to distinctly identify each community.
  - **area, location**: Geographical data for community operations.
  - **rules**: Regulations governing the community; initially an empty array, can be populated as needed.
  - **communityMembers, communityManagers**: Arrays that list the members and managers with their respective details.
  - **events, night_watches**: Schedules and details of community events and night watches.
  - **posts**: Community announcements or updates.

2.4.2 Night Watch Collection

- **Purpose:**Manages details about the security or night watch events organized within communities.
- **Key Fields:**
  - **watch_id**: Identifier for the watch event, linked with community events.
  - **initiator_id, initiator_name**: Information about who initiated the night watch.
  - **community_area**: Specifies where the night watch is held.
  - **watch_date, watch_radius, positions_amount:** Details when the watch happens, the area covered, and positions available.
  - **watch_members**: List of members participating in the night watch.

2.4.3 Posting Collection

Purpose:The Posting Collection stores user-generated content within the community, such as announcements, event information, and calls for volunteers. It facilitates communication and engagement among community members.
Key Fields:

- **id**: MongoDB's built-in unique identifier for each document.
- **post_id**: A unique identifier for each post, used for tracking and referencing posts.
- **user_id, user_name**: Information identifying the user who created the post, linking the post to the specific user profile in the Users Collection.
- **community_area**: Specifies the target community or area of the post, helping to filter posts relevant to specific user groups.
- **post_content**: An object containing:
  - **header**: The title of the post, summarizing its purpose or topic.
  - **body**: The detailed message of the post, which may include formatted text to enhance readability and provide comprehensive information.
- **post_date**: The timestamp of when the post was created, crucial for sorting and displaying posts chronologically.

2.4.4 Events Collection

Purpose: An event document in MongoDB provides a structured representation of an event within the application. It includes various fields that capture essential details for management, tracking, and integration of the event.

- **Key Fields**:
  - **ID**: An ObjectID or unique identifier assigned by MongoDB, serving as the document's primary key.
  - **Event ID**: A unique identifier for the event, allowing for precise referencing and distinction from other events.
  - **Initiator:** Represents the ID of the user or entity who created or initiated the event.
  - **Community:**Indicates the community associated with the event, tying it to a specific geographical or social context.
  - **Location:** An embedded document containing details such as:
    - **address:** The textual address of the event's location.
    - **latitude:** The geographical latitude coordinate.
    - **longitude**: The geographical longitude coordinate.
  - **Event Details:**
    - **event_name:** Specifies the event's name, providing context for its nature and activities.
    - **event_type:** Categorizes the event, indicating its general type or classification.
  - **Timing:**
    - **start_time:** Defines the event's start time.
    - **end_time:** Defines the event's end time.
  - **Participants:**
    - **guests:** A list indicating invited guests.
    - **attending:** A list showing confirmed attendees.

2.4.5 Users Collection

- **Purpose**:Manages user-specific data, including personal information and their involvement in community activities.
- **Key Fields**:
  - **id, name, email, password**: Basic user identification and authentication data.
  - **phoneNumber, location**: Contact and geographical data.
  - **friends**: Array of objects detailing friends of the user, potentially for social features within the app.
  - **requests, radius, communities, night_watches**: User's engagement in various community-related functionalities.

3 Analysis And Conclusions

3.1 Challenges and Solutions

Challenge 1: Resource Limitations

Challenge: Significant obstacles to our research were related to the lack of suitable development environments. A distinct development server where updates could be regularly established and tested would have been the ideal situation.

Solution: Due to limited resources, each team member worked independently on their own machines—Danor focused on the frontend, and Sahar handled the backend development. We utilized Git as our version control system to manage and integrate our work. Although this method required frequent pulling of updates to Danor's computer for combined testing, it enabled us to progress despite the lack of a unified development environment. This approach was not without its flaws, primarily the increased risk of integration issues, but it was a necessary compromise given our time and resource limitations.

Challenge 2: Implementing Location Services

Challenge: A technical challenge we encountered was calculating the geographical radius for localizing community activities. We needed a reliable method to determine the nearest activities within a specified circular area around a user's location.

Solution: After researching various mathematical approaches, we discovered the Haversine formula, an equation that calculates the distance between two points on the Earth using their latitudes and longitudes. Implementing this formula allowed us to accurately and efficiently find activities within the desired radius. This solution proved to be an essential component of our location-based services, enhancing the app's functionality by enabling users to discover local community events and initiatives.

Challenge 3: UI/UX Design

Challenge: The lack of UI/UX design knowledge or skill among the team members presented a significant obstacle to creating an app interface that was both intuitive and user-friendly.

Solution: To overcome our lack of design experience, we decided to leverage existing resources. We explored various UI/UX templates and designs available on Figma, a platform known for its high-quality design resources. By adapting a design that closely matched our vision and functional requirements, we were able to accelerate the design process and focus on adapting it to fit our specific needs. This approach not only saved time but also ensured that we could provide a visually appealing and functional user interface without having to create one from scratch.

3.2 Results and Conclusions

The journey to develop "UrbanHive" was marked by several challenges, each providing a unique learning opportunity and paving the way for innovative solutions. Despite initial hurdles related to resources, location services, and UI/UX design, the project culminated successfully with the creation of a robust application built on Expo React Native and a solid backend service using Python Flask.

3.2.1 Achievements

1. Comprehensive Community Engagement Tools:"UrbanHive" carefully integrates a number of elements required for involvement in the community. Among these is the night watch function, which enables users to effectively plan and take part in neighborhood night watches. Furthermore, the program facilitates the creation, posting, and commenting on events, thereby promoting a dynamic platform for social engagement within the community.

2. Effective Communication Channels: Better communication and teamwork among users is made possible by the app's chat tools for friends and community members. Ensuring participation in local events and projects and creating a sense of community are made possible by this feature.

3. Event and Activity Integration: Users can easily view and engage with community activities, enhancing their involvement and presence within their local areas. This integration not only supports the app's primary function but also adds significant value by keeping users informed and engaged with their community's happenings.

4. From Night Watch to Social Networking: Initially targeted to satisfy the need for a specialized night watch application, "UrbanHive" evolved into a more comprehensive platform that encompasses features typical of social networking sites. This transition marks a significant expansion in the app's scope, and tries to make it a versatile tool for community engagement and personal interaction.

3.2.2 Future Enhancements

We intend to add a number of significant features to "UrbanHive" as it develops further to improve its usability and functionality:

- **Push Notifications**: Push notifications will be used to notify users in real time. In order to keep users informed and involved, this tool will notify them of new postings, planned events, and updates within their communities.
- **Map View**: A map view is going to be added to improve the app's geographical features. Users will be able to visually explore local events, night activities, and more thanks to this feature, which will enhance their ui experience by offering a spatial context.
- **Cloud Infrastructure on AWS**: We are going to a cloud infrastructure utilizing Amazon Web Services (AWS) in order to meet our application's expanding needs and ensure scalability and reliability. This change will improve data security, enhance overall performance, and offer solid backend support for our services.

3.3 Lessons Learned

As we reflect back on the development of "UrbanHive," a number of important lessons come to mind that show both our advantages and our potential for growth. While our method was adequate to overcome major technical and resource limitations, it also offered insightful teaching points.
Effective Use of Existing Resources and Tools: In order to overcome our lack of a unified development environment and design experience, we discovered how important it is to make use of already-existing resources and tools, such as Git for version control and Figma for UI/UX design. These resources were crucial to organizing our group work and accelerating the design phase.
adaptation and Problem Solving: We learned the importance of adapting and careful research from overcoming obstacles such as deploying location services. Our ability to perform geographical computations using the Haversine formula showed that we could find effective solutions for challenging issues.

3.3.1 Areas for Improvement

- **Early and Continuous Testing**: One of the main areas for improvement would be incorporating user testing earlier in the development process. Early feedback could have significantly influenced design adjustments and functionality enhancements, leading to a more user-friendly application.
- **Planning and Anticipation of Technical Needs**: More thorough preparation was needed for the design of our program and the fusion of frontend and backend components. Better speed optimizations and more seamless transitions would have been possible if these needs had been anticipated.
- **Structured Code Reviews**: Maintaining improved code quality and consistency would have been made possible by implementing regular and consistent code reviews. Having formalized review procedures in place could have improved communication of data and identified any problems early in the development process.

3.4 Project Metrics

To measure the success and guide the development of "UrbanHive," we established several project metrics centered around functionality, user engagement, and team learning:
Functional and Scalable Application: Our primary objective was to deliver a robust and scalable application capable of supporting community engagement through features like night watches, social interactions, and event management.
This goal was achieved through the integration of Expo React Native and Python Flask, which provided a solid foundation for our application.

Team Collaboration and Skill Enhancement: We aimed to use this project as an opportunity to enhance our team collaboration and individual technical skills, particularly in areas where we lacked prior experience, such as UI/UX design and advanced location services. The project served as a practical learning environment, and we made significant progress in these areas.

User Engagement and Feedback: We sought to create an application that was not only functional but also engaging for the users. By adapting UI/UX designs from professional templates and implementing intuitive features, we aimed to maximize user satisfaction and engagement. Initial user feedback has been positive, indicating success in this metric.

Enjoyment and Satisfaction: Lastly, we wanted to ensure that the project process was enjoyable and fulfilling for the team. Overcoming the challenges and seeing the application come to life provided a sense of accomplishment and satisfaction, reinforcing our passion for software development.
We felt responsible for the app because we made everything by ourselves in a team of 2 people.

4 Testing Process

4.1 Introduction

The testing process is an essential part of software development that ensures the application meets its specifications and is free of defects. For our application, we employ a comprehensive testing strategy that covers various levels of the system, ensuring each feature functions as expected under different scenarios. This section outlines the specific test cases designed for each module of our application. By conducting rigorous unit tests, we aim to identify and resolve any issues before they impact the end-users. Each test case described below is crafted to verify the functionality, reliability, and security of the application components, thereby facilitating a robust and user-friendly experience.

4.2 Unit Testing

Each unit test focuses on a small, isolated part of the codebase to ensure it performs as intended. Our unit tests are categorized under different modules of the application such as User management, Community interactions, Event handling, Posting functionalities, and Night Watch operations. Below are the specific test cases for each module.

4.2.1 User

Table Name	Test-case ID	Description
RESTUser.py	Test-case 1: Get Users	Test fetching all users from the database successfully and handling exceptions.
RESTUser.py	Test-case 2: Add User	Test adding a new user with all required fields and handling duplicate user data.
RESTUser.py	Test-case 3: Get User by ID	Test retrieving a single user by ID and handling user not found cases.
RESTUser.py	Test-case 4: Update User	Test updating user data by ID and handling user not found and invalid update data cases.
RESTUser.py	Test-case 5: Delete User	Test deleting a user by ID and handling a user not found case.
RESTUser.py	Test-case 6: Check User Password	Test validating user password with ID and handling invalid ID or password cases.
RESTUser.py	Test-case 7: Change Password	Test changing user password and validating new password fields.
RESTUser.py	Test-case 8: Update User Radius	Test updating the user's location radius and handling invalid radius inputs.
RESTUser.py	Test-case 9: Add Friend	Test sending a friend request, verifying users' existence, and adding to requests.
RESTUser.py	Test-case 10: Respond to Request	Test responding to a friend request, either approving or declining, and handling invalid response.
RESTUser.py	Test-case 11: Delete Friend	Test deleting a friend from a user's friend list and handling non-friendship cases.

4.2.2 Community

Table Name	Test-case ID	Description
Community.py	Test-case 1: Add Community	Test adding a new community with a unique area and location, and handling duplicate entries.
Community.py	Test-case 2: Add User to Community	Test updating community requests for users to join a community and handling database errors.
Community.py	Test-case 3: Respond to Community Request	Test responding to community join requests with accept or decline and update records accordingly.
Community.py	Test-case 4: Delete User from Community	Test removing a user from a community's member list and the user's community list.
Community.py	Test-case 5: Get Communities by Radius and Location	Test retrieving communities within a certain radius and location.
Community.py	Test-case 6: Get Community Details by Area	Test fetching details of a community by its area name, handling missing or non-existent communities.
Community.py	Test-case 7: Get All Communities	Test retrieving all communities from the database, and handling any database errors.
Community.py	Test-case 8: Request to Join Community	Test sending a join request to a community by a user and handling invalid requests.
Community.py	Test-case 9: Respond to Join Request	Test community manager's response to a join request and update community membership.

4.2.3 Events

Table Name	Test-case ID	Description
Events.py	Test-case 1: Add Event	Test creating a new event, pushing event data to users and communities, and handling errors during the process.
Events.py	Test-case 2: Get All Events	Test retrieving all events from the database, ensuring they are formatted correctly for the JSON response.
Events.py	Test-case 3: Respond to Event Request	Test user response to an event invitation, updating event and user records based on acceptance or decline.
Events.py	Test-case 4: Delete Event	Test deletion of an event, removing references from the communities and users, and handling event not found errors.

4.2.4 Posting

Table Name	Test-case ID	Description
Posting.py	Test-case 1: Add Post	Test adding a new post to a community by a user, ensuring the user is a member, and handling missing fields and database errors.
Posting.py	Test-case 2: Delete Post	Test deleting a post by post_id, ensuring the post is removed from both posting and communities collections, and handling errors when the post is not found.
Posting.py	Test-case 3: Add Comment to Post	Test adding a comment to a post by post_id, ensuring all required fields are present, and handling errors when the post is not found.
Posting.py	Test-case 4: Delete Comment from Post	Test deleting a comment from a post using post_id and comment_id, ensuring removal from both posting and communities, and handling errors when the post or comment is not found.

4.2.5 Night Watch

Table Name	Test-case ID	Description
NightWatch.py	Test-case 1: Add New Night Watch	Test the creation of a new night watch, ensuring all required fields are present and that no duplicate watch exists for the same area and date.
NightWatch.py	Test-case 2: Join Night Watch	Test the functionality that allows a user to join an existing night watch, ensuring the user is not already part of the watch and that there are available positions.
NightWatch.py	Test-case 3: Close Night Watch	Test the closure of a night watch by watch_id, verifying that it is properly removed from the night_watch collection, users' night_watches list, and the community's night_watches list.
NightWatch.py	Test-case 4: Calculate Position for Watch	Test the calculation and assignment of positions for a night watch, ensuring the operation adheres to specified parameters like positions_amount and watch_radius.
NightWatch.py	Test-case 5: Get Night Watches by Community	Test the retrieval of night watches for a given community, focusing on future dates and verifying the existence of the community.

4.3 Functional Testing

Functional testing is a critical phase in the development of the UrbanHive application, aimed at verifying that each feature of the app functions according to the specified requirements. This type of testing focuses on user requirements and is crucial for ensuring the reliability and user-friendly nature of the application. The goal of this testing phase is to simulate typical usage scenarios to identify any functional issues before the application is deployed to real users. This plan outlines the key functional tests that will be conducted based on the application's core functionalities as outlined in the project documentation.

Test Environment

- **Platform**: iOS and Android devices
- **Network Conditions**: Tested under localhost network type (Wi-Fi)
- **Data Set**: Use realistic community data, user profiles, and event details to simulate daily usage

Test Area	Test Case Description	Criteria for Success
User Authentication and Management		
Login	Test login functionality with correct and incorrect credentials.	Successful login with valid credentials; error on invalid ones.
User Registration	Ensure that the user can register with all required fields and handle registration with missing or duplicate information.	Successful registration; proper error handling.
Password Recovery	Check the password recovery process for existing emails and handle non-existing emails gracefully.	Correct recovery steps; proper error messages.
Update Profile	Verify that users can update their personal information and handle validation errors.	Successful update; validation errors are caught.
Community Management		
Create Community	Test creating a new community with valid and invalid data.	Community is created successfully; errors are handled.
Join Community	Simulate a user request to join a community and test both acceptance and rejection scenarios.	Requests are processed correctly; appropriate feedback given.
Post to Community	Ensure users can post updates and handle errors related to permissions and data validation.	Posts are successful; errors are appropriately managed.
Community Search	Test searching for communities by different criteria such as location and interests.	Search returns correct results; handles no results found.
Event Management		
Create Event	Test creating events with all required information and handling cases with missing or invalid data.	Event is created successfully; errors are handled.
Attend Event	Simulate user responses to event invitations (accept/decline) and verify the update in the event's participant list.	Responses are registered correctly; list updates.
Event Notifications	Ensure that notifications about upcoming events are sent out correctly and received by intended users.	Notifications are accurate and timely.
Night Watch Operations		
Schedule Night Watch	Verify that night watches can be scheduled with appropriate details and handle scheduling conflicts.	Night watches are scheduled correctly; conflicts are managed.
Join Night Watch	Test the functionality for a user to join an existing night watch and verify the validation for overcapacity and eligibility.	User joins successfully; capacity and eligibility checked.
Night Watch Reminders	Ensure that participants receive timely reminders for upcoming night watches.	Reminders are sent and received as expected.
Posting and Comments		
Add/Delete Post	Test adding and deleting posts by authorized users, and handle unauthorized attempts.	Posts are managed correctly; unauthorized attempts blocked.
Comment on Post	Allow users to add comments to a post and handle cases where the post does not exist.	Comments are added appropriately; errors are managed.

In the course of our project development, we adopted a hybrid approach to testing to ensure comprehensive evaluation of our system's functionality. This involved a combination of automated tests and manual testing procedures. Automated tests were implemented using Python, which enabled us to systematically verify specific functionalities and performance benchmarks. These tests are scripted and repeatable, offering consistent results across different test runs.

Conversely, certain aspects of our system required nuanced human observation to assess user interaction, graphical interfaces, and complex scenarios that automated scripts could not effectively capture. For these components, manual testing was employed.

This approach allowed for dynamic, scenario-based evaluation, which is essential for identifying issues that occur under less predictable conditions.

By integrating both automated and manual testing methods, we aimed to leverage the strengths of each approach to achieve a more thorough understanding of the system’s capabilities and limitations.

5 User Guide

5.1 Login Screen (LoginScreen.jsx)

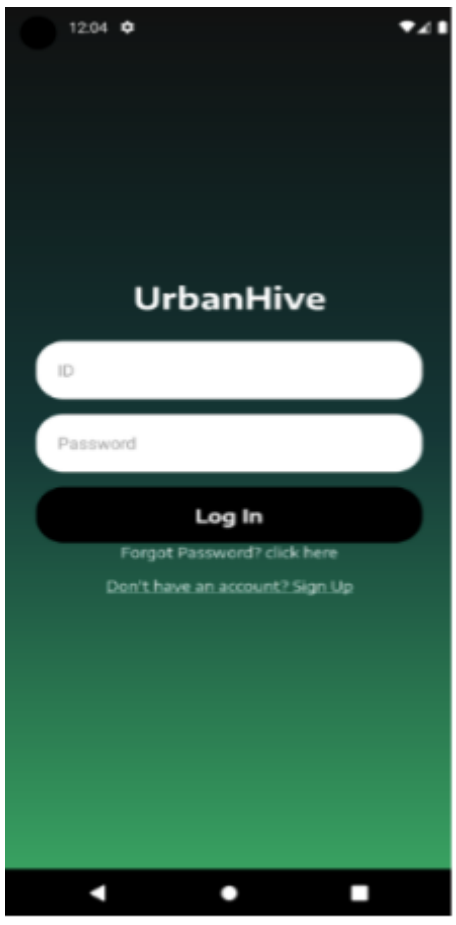

Figure 09 : Login Screen

Description:
The login screen is designed to be simple and user-friendly, allowing existing users to access their accounts quickly.
It features the app's name prominently at the top, followed by two input fields where users can enter their ID and password.
Below the input fields, there is a "Login" button to submit the credentials.
For added user convenience, there are two clickable text links: one for users who have forgotten their password, titled "Forgot Password? click here," and another for new users to create an account, titled "Don't have an account? Sign Up."

Flow:
1.The user opens the app and is greeted with the login screen.
2.They enter their ID and password into the respective fields.
3.To log in, the user taps the "LogIn" button.
4.If the user forgets their password, they can tap on the "Forgot Password?" link to initiate the recovery process.
5.If the user does not have an account, they can tap on the "Sign Up" link to be redirected to the account creation screen.

5.2 Create Account Screen (CreateAccount.jsx)

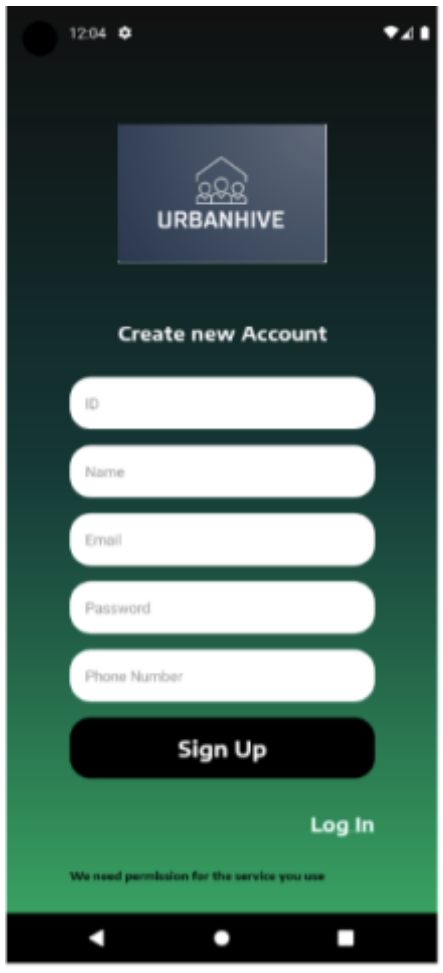

Figure 10: Create Account Screen

Description:

The create account screen follows the app's color scheme and contains input fields for ID, Name, Email, Password, and Phone Number, allowing new users to register for "UrbanHive." The top of the screen showcases the app's logo and name. At the bottom of the screen, there is a "Sign Up" button to submit the new account details.

There's also a text reminder that the app requires permission for the services used, ensuring users are aware of the app's requirements. A "LogIn" link is provided at the bottom for users who already have an account and may have navigated to this screen by mistake.

Flow:

1.From the login screen, new users select "Sign Up" to reach this screen.
2.Users fill in their personal information in the provided fields.
3.After entering all details, the user taps the "Sign Up" button to create their new account.
4.The app requests for a location permission, and takes the current longitude and latitude of the user..
5.If a user already has an account, they can return to the login screen by tapping the "LogIn" link.

5.3 Home Screen (HomeScreen.jsx)

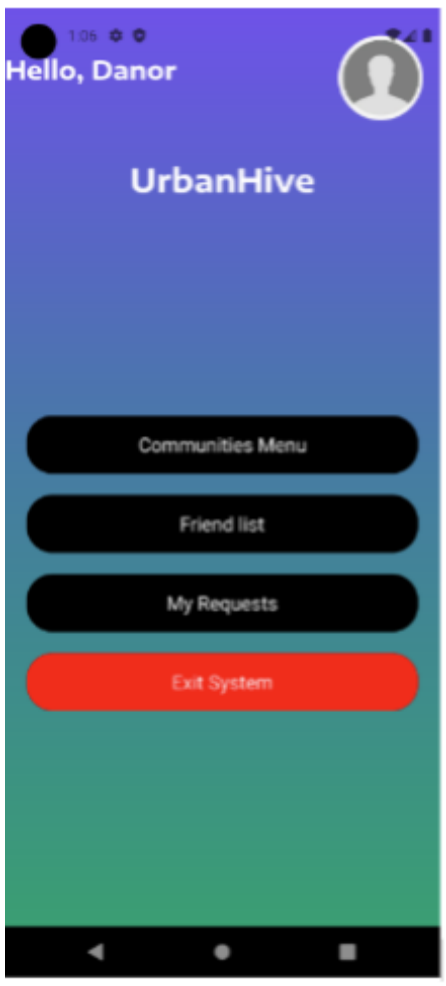

Figure 11: Home Screen

Description:
Upon successful login, the user arrives at the Home Screen, the central navigation point of the "UrbanHive" app. The user is greeted by name at the top of the screen, making for a personalized and engaging experience.
The screen features a straightforward layout with a series of buttons for the main functions of the app. These are:

- **Communities Menu:** This button likely leads to a section where the user can view various communities they are a part of or explore new ones to join.
- **Friend list:** Tapping this allows the user to view and manage their list of friends within the app, facilitating social connections.
- **My Requests**: This is where the user can manage incoming friend or community requests, keeping them engaged and connected.
- **Exit System:** A prominent red button designed for users to log out of the application safely.

Flow:
1.The user logs in and is directed to the Home Screen.
2.They are presented with four options to navigate through the app's main features.
3.Selecting the Communities Menu would allow the user to participate in community-specific activities or explore other communities.
4.The Friend list provides a social element, where the user can interact with friends, initiate chats, or plan community activities.
5.Through My Requests, users can accept or decline friend and community invitations, helping them curate their social network within "UrbanHive."
6.When finished, the user can exit the application securely by pressing the Exit System button.

5.4 Community Lobby Screen (CommunityLobby.jsx)

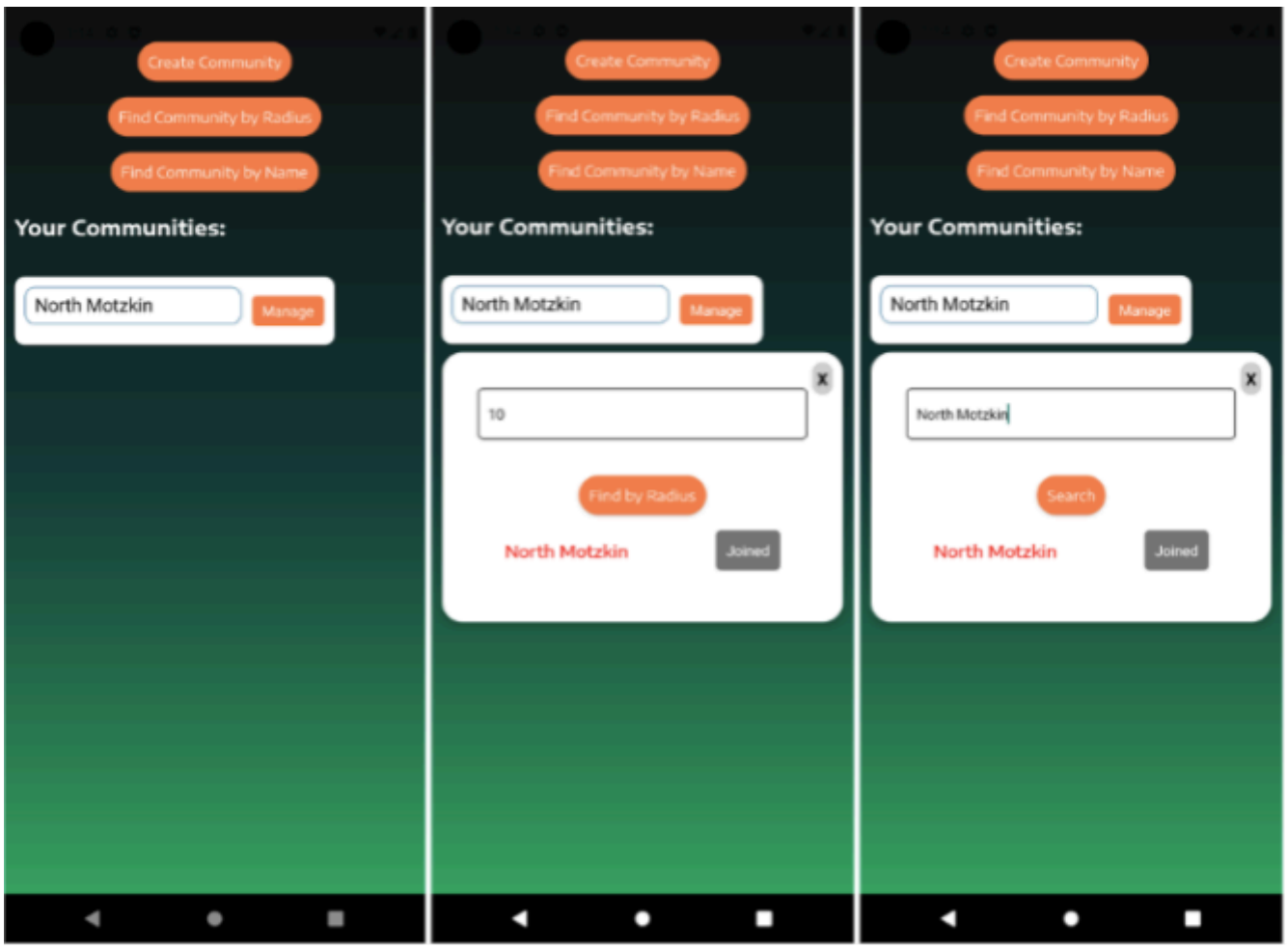

Figure 12: Community Lobby Screen

Description:

The Community Lobby is a dedicated space where users can manage their engagement with different communities within the "UrbanHive" app. This screen is accessible after selecting the "Communities Menu" button from the Home Screen. It provides several options for users to expand their community interactions:

- Create Community: A button for users to establish a new community. This feature is intended for users who want to start their own groups based on location, interests etc..
- Find Community by Radius: Allows users to search for communities within a certain distance from their current location, using geolocation services.
- Find Community by Name: Enables users to look up communities directly by their name, which is useful for those who know exactly what they're searching for.

Below these options is a list titled "Your Communities," which displays the communities the user is currently a part of. If the user is a manager of any community, a "Manage" button appears next to the community's name, leading to administrative functions for that particular group.

Flow:

1.Upon entering the Community Lobby, users can opt to create a new community, find communities nearby, or search for a specific one by name.
2.To create a new community, users will click the "Create Community" button, likely prompting them to fill in details about the new community they wish to establish.
3.For finding communities, users can choose between searching by radius, which may present a field to enter the desired search radius, and searching by name, which would open a search bar.
4.The list of "Your Communities" reflects the user's current memberships. Clicking on a community name takes the user to that community's dedicated screen, where they can participate in discussions, events, and other community-specific activities.
5.For community managers, the "Manage" button allows them to access administrative options where they can oversee membership, manage posts, events, and other community-related settings.

5.5 Community Manager Screen (CommunityManagerScreen.jsx)

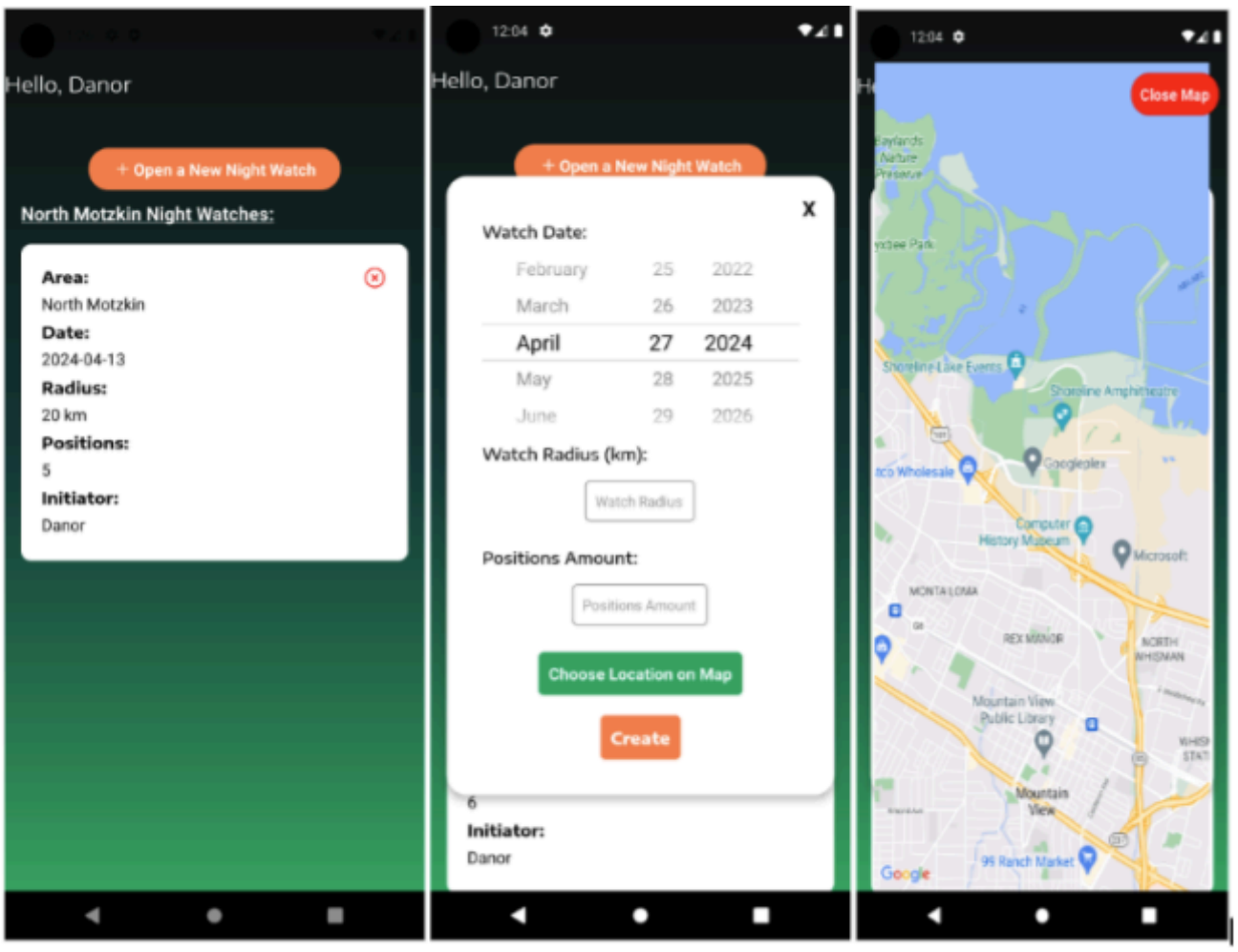

Figure 13: Community Manager Screen

Description:

The Community Manager Screen is a functionality-specific interface that allows a community manager to oversee and organize night watch events for their community. This specialized dashboard provides the administrative role, providing tools necessary for effective community management.

The screen greets the manager by name, offering a personalized touch. The most prominent feature is the button to "+ Open a New Night Watch," suggesting an action to create a new night watch event .

Below the button, there is a list of "{Community_name} Night Watches," which includes details such as area, date, radius, number of positions available, and the initiator's name. There is also a red 'X' button, used to close the pop-up modal.

Flow:

1.A community manager taps on the "Manage" button next to a community's name in the Community Lobby to access the Community Manager Screen.

2.On this screen, the manager can view existing night watch events under the section "North Motzkin Night Watches."

3.To organize a new night watch, the manager would click the "+ Open a New Night Watch" button.

4.Upon clicking, a modal or new screen appears, prompting the manager to enter details for the new night watch, such as date, radius, and the number of positions required.

5.After entering the information, the manager can create the event by tapping the "Create" button.

6.Managers also have the option to manage requests for joining the community from other users by accepting or declining the user to join the community.

5.6 Community Screen (CommunityHubScreen.jsx)

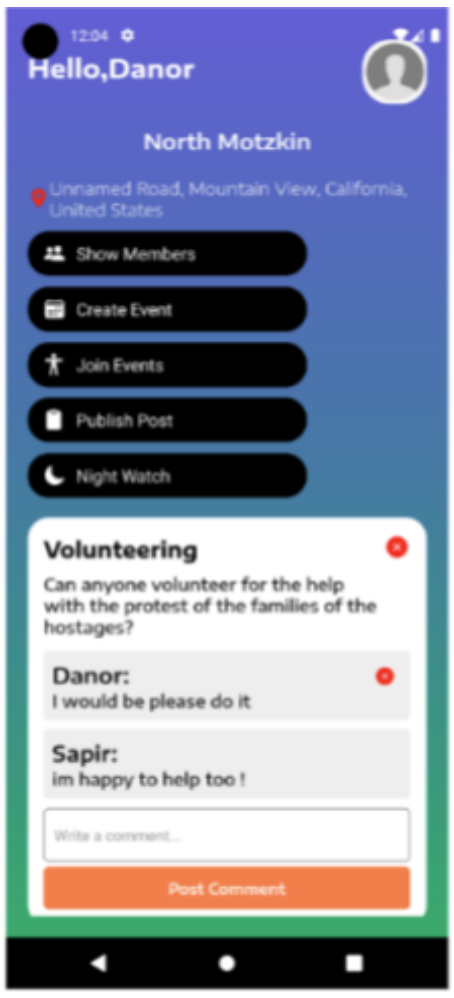

Figure 14: Community Screen

Description:

The Community Hub Screen is a hub for individual community activities and interaction.
At the top, the user sees a welcome message along with their community name, North Motzkin for example, and the community's location details.
The screen offers a suite of interactive features:

- **Show Members:** Lists all the current members of the community.
- **Create Event:** Allows users to create a new event for the community.
- **Join Events:** Displays upcoming events that users can participate in.
- **Publish Post:** Enables users to share posts with the community.
- **Night Watch:** Users can view and sign up for upcoming night watch events.

The user interface also displays posts made by community members.
Each post includes an option for users to comment or delete their own posts, as indicated by a red 'X' button.

Flow:

1.Upon entering the Community Screen, the user can perform several actions related to community interaction and management.
2.By clicking on Show Members, the user can view a list of individuals who are part of the community, enhancing the social aspect of the app.
3.The Create Event feature allows users to organize new events, which can then be shared with the community for participation.
4.The Join Events button provides a list of available events that users can attend, fostering community engagement.
5.With Publish Post, users can share updates, information, or discussions with the community, visible in a feed format.
6.The Night Watch feature gives users the ability to organize or join community patrol events, which is a significant aspect of the app's functionality aimed at safety and vigilance.
7.In the posts section, users can engage with community content through comments. Post creators have the ability to delete their posts, allowing for content management.

5.7 Community Create Event Screen (CommunityCreateEvent.jsx)

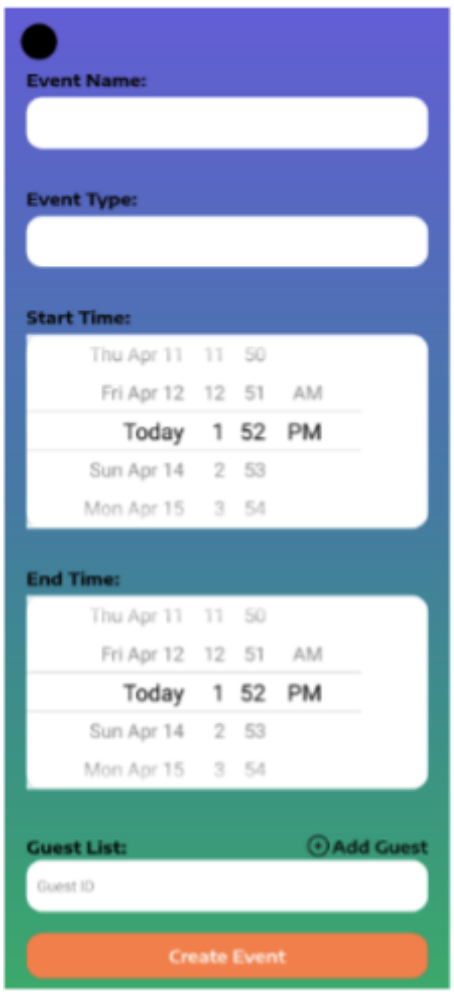

Figure 15: Community Create Event Screen

Description:
The "CommunityCreateEvent.jsx" screen is where community members can create a new event within their community hub. It provides a simple form for the user to input details about the event they wish to hold.

Components and Flow:
Event Name: A field for the user to enter the title of the event.
Event Type: A dropdown or text field to specify the nature of the event (e.g., social, meeting, community service).
Start Time: A picker component where the user can select the date and time when the event will begin.
End Time: Similar to the Start Time, this allows the user to set when the event will conclude.
Guest List: Users can add attendees to the event. It seems there is an option to input a "Guest ID" which could link to a user's profile or a way to invite non-members.
Add Guest: A button to add additional guests to the event, potentially opening a search interface to select community members or to input details for those outside the community.

5.8 Community Publish Post Screen (CommunityPublishPost.jsx)

Figure 16: Community Publish Post Screen

Description:
The "CommunityPublishPost.jsx" screen provides users with the ability to share announcements, updates, and information with their community by creating a new post.
This screen is accessed by tapping the "Publish Post" button within the community hub.

Components and Flow:
Title: A text field for the user to enter the subject or title of their post.
Body: A larger text area for the user to write the main content of the post.
This could be detailed information about an event, a call to action, or any other relevant community message.
Publish Post: After filling in the title and body of the post, the user can submit it to the community by tapping this button.

5.9 Community Night Watch Screen (CommunityNightWatch.jsx)

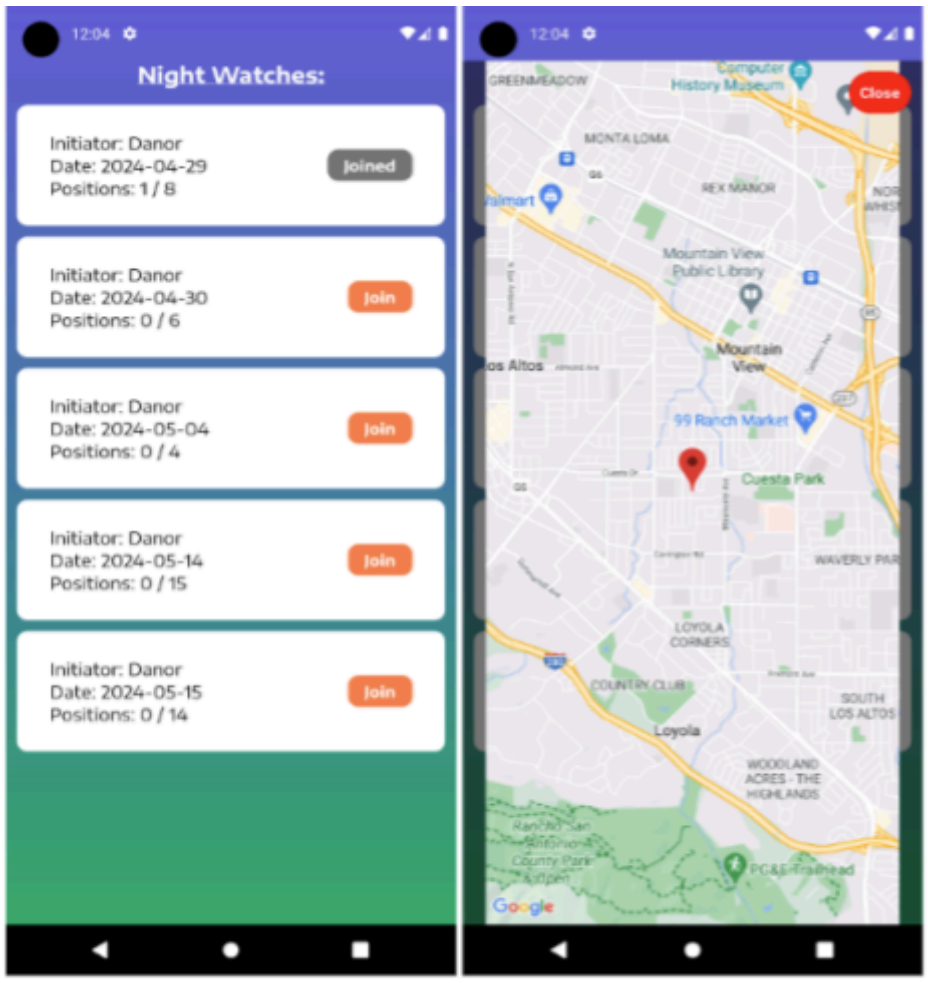

Figure 17: Community Night Watch Screen

Description:
The "CommunityNightWatch.jsx" screen allows community members to view and join night watch events within their specific community.
The interface presents a list of available night watch slots, showcasing the details of each event.
Components and Flow:
List of Night Watches: Each item on the list displays the name of the initiator (which appears to be the same for these entries, "Danor"), the date of the night watch, and the available positions out of the total (e.g., "1/5" indicates one position filled out of five available).
Join Button: An orange "Join" button is shown beside night watches with available positions, indicating that the user can sign up for participation.
Once joined, the button changes to indicate "Joined," confirming the user's registration for that event.

6 Maintenance Guide

In this chapter, we will detail the essential setup and running processes for both the client and server components of the UrbanHive project.

Ensuring a consistent development environment is crucial for both building and maintaining the application effectively.

System Requirements

- CPU: A multi-core processor (Intel i5/i7/i9 or AMD equivalent) is recommended for efficient compilation and emulation, especially when using Android Studio and Xcode.
- RAM: At least 8GB of RAM, though 16GB or more is preferred for better performance with multiple emulators and development tools running.
- Storage: A solid-state drive (SSD) with at least 50GB of free space to handle the various installations and project files comfortably.

6.1 Client Project Setup

6.1.1 Development Environment Requirements

1. **Node.js**: The runtime environment for executing JavaScript code outside of a browser. Download and install from https://nodejs.org

2. **Visual Studio Code**: Integrated Development Environment (IDE) for editing and managing the project files. Available at https://code.visualstudio.com/

3. **Expo CLI**: A command-line tool that enables you to work with Expo apps. Install globally using npm:

```
npm install -g expo-cli
```

4. Android Studio: Provides an Android emulator for simulating Android devices. Download at https://developer.android.com/studio

5. **Xcode**: Necessary for iOS development, providing tools and the iOS simulator. Available on the Mac App Store : Xcode on Mac Store

6.1.2 Running the Client

1.Before running the client application, it is essential to install all necessary dependencies:

```
npm install
```

or, if you prefer using Yarn:

```
yarn install
```

This step ensures that all the required packages are locally available to build and run the project.

2. To execute a development build for testing on emulators or real devices, use:

```
eas build --platform android --profile development
```

or to build a development build for ios:

```
eas build --platform ios --profile development
```

then you will get a url for install it on the device
* you need an expo account for this, register here: https://expo.dev/

3. To start the client application after install the development build, use the following command in the terminal:

```
npx expo start
```

This command initializes the Expo development server and provides a QR code to open the project on a physical device and options to run it on Android or iOS simulators.

-The client is configured to communicate with the server running on localhost:5000.
If there are network issues while the server is running, the base URL can be adjusted in the config/config.js file based on the ip address that shows when running the server (in pycharm or terminal).

6.2 Server Project Setup

6.2.1 Development Environment Requirements

1. **Python**: Install Python from https://python.org to run the server-side code.

2. **PyCharm**: A Python IDE by JetBrains, optimal for managing Python-based applications. Download at https://www.jetbrains.com/pycharm/

3. **MongoDB Atlas**: A cloud database service to host and manage your MongoDB database. Register and configure at https://www.mongodb.com/cloud/atlas

6.2.2 Server Dependencies

- Before running the server, ensure all dependencies are installed. Setup a virtual environment and install packages from `requirements.txt`:

```
python -m venv venv
source venv/bin/activate  # On Unix/macOS
venv\Scripts\activate  # On Windows
pip install -r requirements.txt
```

6.2.3 MongoDB Atlas Setup

- Configure MongoDB Atlas:
    - Sign up and log in to MongoDB Atlas.
    - Create a new project and build a new cluster.
    - Navigate to the "Database" section and click on "Connect".
    - Choose "Connect your application" and copy the connection string provided.
- Local Setup:
    - Replace <password> in the connection string with your database user's password.
    - Modify your server's configuration file or environment variables to use this connection string.
    - Ensure network access is configured in MongoDB Atlas to allow connections from your server's IP address.

6.2.3 Running the Server

- Launch the Flask application using:

```
flask run
```

This command starts the server on the local machine, accessible via `localhost` on the configured port 5000.
or open the project in pycharm and set up an interpreter and run it from there.

6.3 Contact Information

- Email : support@urbanhive.com
- GitHub : https://github.com/Danor93/UrbanHive, https://github.com/saharoz1602/UrbanHiveServer

6.4 Contribution to Maintenance

Any contributions you make are greatly appreciated. If you have a suggestion that would make the application better, please follow these steps:
1.Fork the repository.
2.Create your feature branch (e.g., ‘git checkout -b feature/your-feature-name‘).
3.Commit your changes (‘git commit -am ’Add some feature’‘).
4.Push to the branch (‘git push origin feature/your-feature-name‘).
5.Open a pull request.

Alternatively, you can simply open an issue with the tag "enhancement". Your contributions will be reviewed and merged to improve the application.

References

1. Render,Commit and Mount - how react native works : https://reactnative.dev/architecture/render-pipeline
2. How cross platform works on react-native : https://reactnative.dev/architecture/xplat-implementation
3. React native documentations : https://reactnative.dev/docs/getting-started
4. Expo concepts : https://docs.expo.dev/core-concepts/
5. MongoDB Guide : https://www.mongodb.com/resources/products/fundamentals/basics
6. Python Flask : https://www.geeksforgeeks.org/flask-tutorial/
7. MVC design pattern : https://www.codecademy.com/article/mvc
8. Service layer use case for flask : https://www.oreilly.com/library/view/architecture-patterns-with/9781492052197/ch04.html