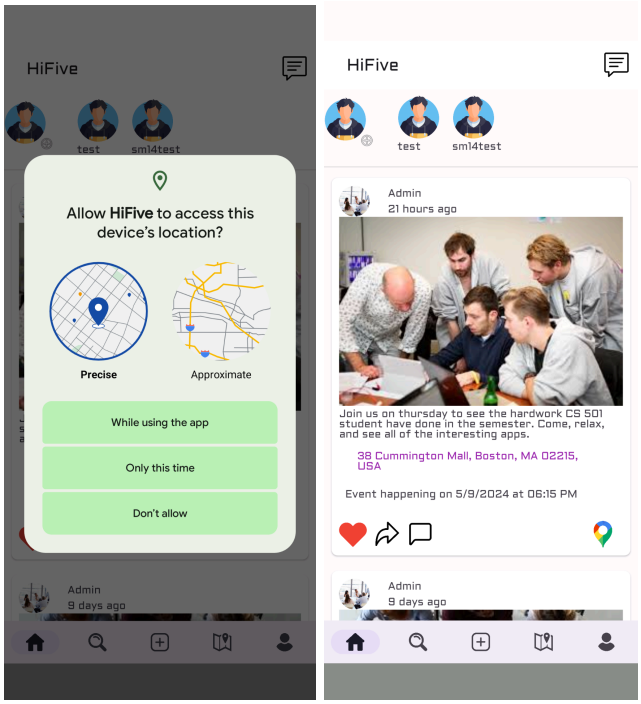
HIFIVE

Samuel Pak, Jonathan Thomas, Mariano Eliseo Amaya
May 9, 2024

Apps features and how they were implemented:

Home:



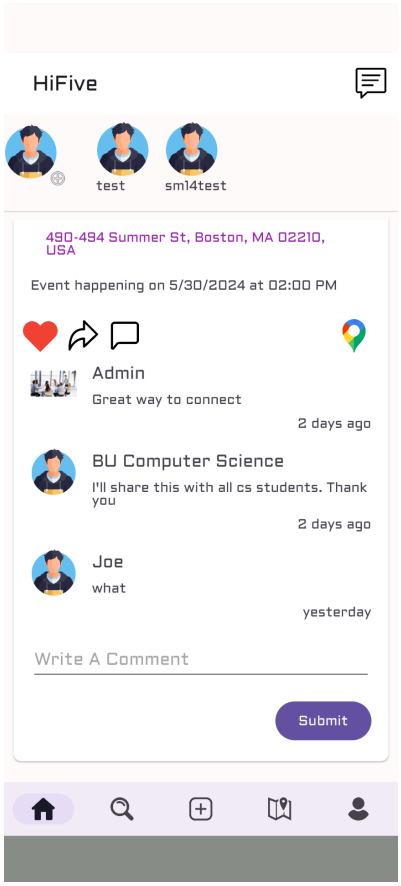
The FollowAdapter.kt is designed to display a list of users that the current user follows. It handles loading and showing user profile images and names in a RecyclerView.
Methods Used: OnCreateViewHolder(): This method inflates the custom layout for each item in the RecyclerView, which includes image and text views for displaying user details.
getItemCount(): Returns the total number of users in the follow list.
onBindViewHolder(): Assigns the user data (profile image and name) to each ViewHolder. It uses Glide for image loading, which is efficient for memory and disk caching images. Glide is chosen for image handling due to its efficiency and ease of use, especially for loading images from URLs, which is common in apps dealing with user profiles.

The PostAdapter.kt manages the display of posts in the app. Each post includes details like the post image, user who posted it, timestamp, and actions like liking or sharing.
Methods Used: OnCreateViewHolder(): Similar to FollowAdapter, it prepares the view structure for each post item.
getItemCount(): Returns the number of posts to display.
onBindViewHolder(): This method is more complex as it not only binds the post data but also interacts with Firebase to fetch and display the poster's user data. It handles user interaction for likes and navigation to Google Maps based on post data. Handling user interactions within onBindViewHolder makes it straightforward to manage actions specific to each post, such as liking a post or navigating based on its location data.

The HomeFragment.kt serves as the central hub where posts and follows are displayed. It initializes and updates the adapters for posts and follows.
Methods Used: loadProfileImage(), loadPosts(), loadFollows(): These methods fetch data from Firebase and update the respective adapters.
OnCreateView(): Sets up the RecyclerViews and other UI elements, binds data to adapters, and loads the initial data.
onOptionsItemSelected(): Handles menu selections, facilitating navigation to other parts of the app like the message section.
The separation of data fetching and UI setup helps in managing the lifecycle of the fragment effectively, ensuring that data is loaded at the right time (like resuming the fragment) and UI interactions are handled smoothly.

Data Fetching and Display: HomeFragment.kt uses FollowAdapter.kt and PostAdapter to display data fetched from Firebase. This makes the code cleaner and the app more maintainable.
User Interactions: Actions in PostAdapter.kt (like liking a post) potentially update the data model, which is then reflected in HomeFragment through adapter notifications.
Navigation: HomeFragment.kt facilitates navigation to the MessageFragment, linking different parts of the app in a user centric flow.

Add a comment:



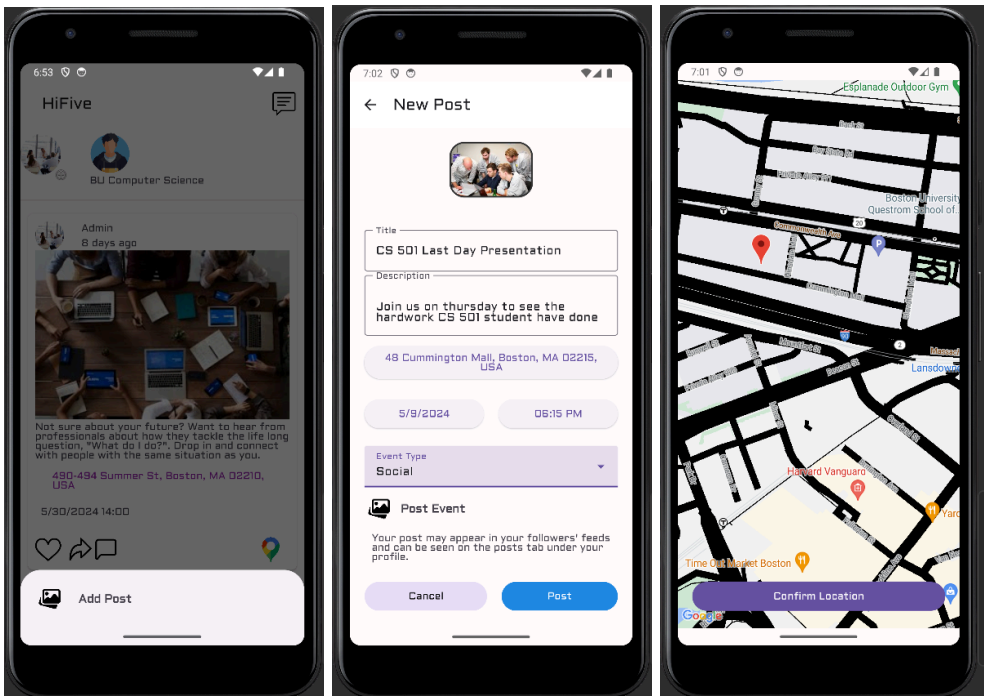
The process of loading and displaying comments in the PostAdapter.kt and CommentAdapter.kt classes involve several key components and methods that interact to provide a smooth user experience.
Each MyHolder instance of PostAdapter initializes its own CommentAdapter with an empty list. This adapter is responsible for rendering individual components.
The visibility of the comment section is toggled using a button. If the comments are visible, they are hidden when the button is pressed, and vice versa. When made visible, it triggers the loading of comments for that specific post. Comments are submitted via an input field and a button. Upon clicking the submit button, the entered text is pushed to Firebase under the specific post's comment collection, including the user's ID, name, and profile image URL fetched asynchronously.

Loading and displaying comments: Comments for a post are loaded from Firebase when the comments section is opened. The loadComments method fetches comments, sorts them by timestamp, and updates the CommentAdapter to display these comments.

Comment Binding: The CommentViewHolder binds each comment to the layout. This includes setting the user's name, profile image, comment text, and how long ago the comment was posted using the TimeAgo library for relative time display.

This setup ensures that comments are dynamically loaded and displayed as part of each post. The interaction between PostAdapter and CommentAdapter is crucial for maintaining a clean separation of concerns. PostAdapter handles the overall post layout and toggling of the comment section. While CommentAdapter focuses solely on rendering the comments. This modularity makes the code easier to manage and extend, such as by adding features like editing or deleting comments.

Add Post:



The process of adding a post in the AddFragment.kt class involves several components and methods that handle user interaction, navigation, and data transmission between activities. Here's a detailed explanation of how it works.
AddFragment is a BottomSheetDialogFragment that provides users with a UI to initiate actions for adding different types of content.
Fragment_add.xml: The layout includes a LinearLayout that acts as button @+id/post. When clicked, it triggers an intent to start PostActivity.
EventHandlers (Post Buttons): Sets up an OnClickListener for the post linearlayout. When clicked, it: logs the current location pulled from a shared MapsViewModel. Creates an intent to launch PostActivity, passing the current location as an extra in the format "latitude, longitude. Starts PostActivity and finishes the current activity to prevent back stack navigation issues.

When AddFragment is created, it initializes the FragmentAddBinding to bind UI components from the XML layout. The user interacts with the UI by clicking on the post section. This is expected to be a straightforward interaction to initiate the posting process. The MapsViewModel is accessed to fetch the current location. This model is shared across the application or between fragments to maintain a consistent state, particularly for location data. The location is logged for debugging purposes and added as an extra to the intent. An intent is created and configured to launch PostActivity. This includes passing data (the current location) which PostActivity can use, for example, to show on a map or as metadata for the post. The current activity finishes after starting PostActivity to clean up the activity stack, ensuring that pressing back does not return to the AddFragment.

PostActivity.It is a central component in the application that allows users to create and publish posts. This activity manages user inputs for post details, handles media uploads, interacts with a map for location selection, and commits post data to Firebase Firestore.

ActivityPostBinding: The layout for PostActivity.kt includes fields for title, caption, image selection, location, date, and time setups. It also has buttons for submitting or canceling the post. It is used to bind layout views to interact programmatically. Intent receives the location passed from AddFragment to use as a default or assist in further location configurations.

Image selection (launcher1). ActivityResultContracts.GetContent() I used to open a gallery or file manager for the user to select an image. uploadImage() is a custom function that uploads the selected image to a specified directory in Firebase storage and retrieves the URL.

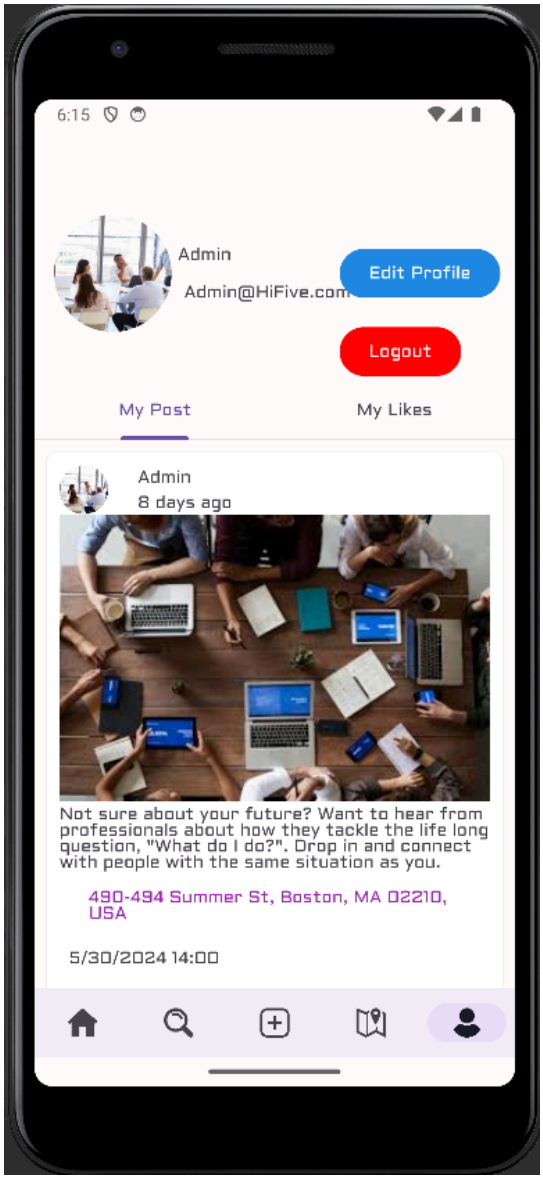
Location setup (launcher2). ActivityResultContracts.StartActivityForResult() starts MapsActivity for location selection. It receives the selected address and geographical coordinates back from MapsActivity. Handles updating the UI with the selected location and enables the post button if all conditions (like image upload) are satisfied.

Date and Time Pickers: DatePickerDialog and TimePickerDialog allow users to select a date and time for the event associated with the post. They utilize the device’s calendar to offer a seamless picking experience.

Event Handlers: Buttons (Post and Cancel): Commit the new post to Firestore or cancel the operation, navigating back to the Home Activity. Image and location pickers: triggers respective picker dialogs or activities.

This comprehensive setup ensures that PostActivity not only meets functional requirements but also adheres to best practices in handling user interactions, data integrity, and seamless navigation within the app.

Look at your posts:



MyPostRvAdapter.kt is designed to manage and display a list of posts by the current user in a RecyclerView. This adapter focuses on handling the presentation of user-generated content, primarily visual posts.
Methods Used: onCreateViewHolder(): Sets up the ViewHolder for each post item by inflating a layout from XML. This method prepares the layout that will hold each post's data.
getItemCount(): Returns the count of posts available for display, dictating how many items the RecyclerView will handle.
onBindViewHolder(): Binds data to each ViewHolder. In this case, it loads the post image from a URL into an ImageView using Picasso.

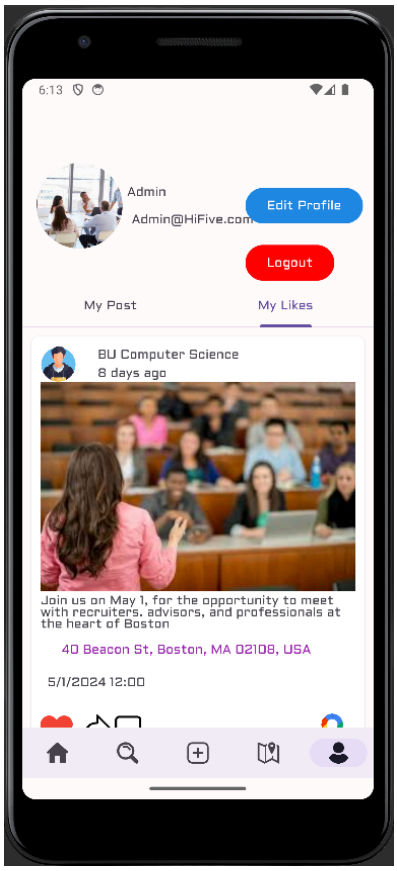
These methods are essential for RecyclerVIew functionality, enabling dynamic data binding and efficient memory management/ Picasso is used for its simplicity and robust handling of image loading and caching. Which is critical in a visually driven app component like this.

MyPostFragment.kt serves as a container for displaying all posts made by the current user. It integrates MyPostRvAdapter to render a list of these posts, handling data fetching and user interaction related to personal content.
Methods Used: OnCreateView(): Inflates the fragment's layout, sets up the RecyclerView with its adapter and layout manager, and initiates the loading of user posts.
LoadUserPosts(): Fetches posts from Firebase Firestore where the uid matches the current user's ID. This ensures that the user sees only their content. THe method updates the adapter's data set and refreshes the RecyclerView upon successful data retrieval.

The separation of UI setup and data handling enhances code readability and maintenance. Fetching data based on userId ensures personalized user experiences, making the app feel more user centric. Firebase.firestore provides a straightforward and efficient way to interact with Firestore databases, allowing for real time data updates and robust querying capabilities.

User-Centric Design: Both classes emphasize user-specific data. MyPostFragment directly interacts with Firestore to retrieve only the posts belonging to the current user, ensuring that the user's experience is personalized and relevant. By separating the adapter and fragment logic, the code remains modular and easier to manage. This separation allows for potential reuse of MyPostRvAdapter in other parts of the app where a similar presentation of posts is needed but might be filtered or managed differently.

Look at your likes:

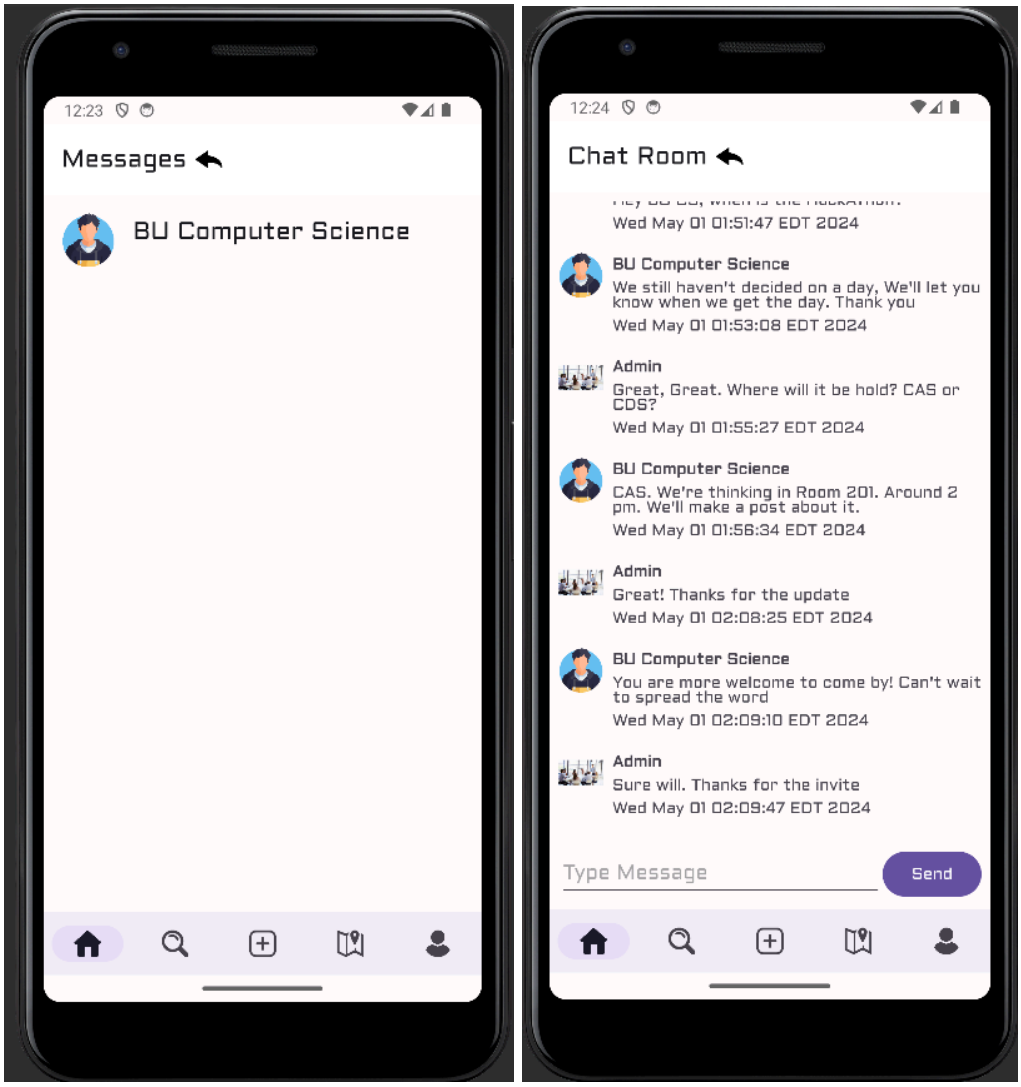


MyLikesFragment.kt is designed to display a list of posts that the user has liked. It allows users to revisit their favorite content from their profile section.
Methods used: onCreateView(): Sets up the fragment layout, initializes the PostAdapter.kt and configures the RecyclerView to display the liked posts.
LoadLikedPosts(): Fetches the list of post IDs that the user has liked from Firebase Firestore. It queries a subcollection under each user's document to retrieve likes. This method is critical as it forms the bridge between user actions and the data reflected in the app.
fetchPostsByIds(): retrieves actual post details from the Firestore database using the IDs gathered from the user's likes. It ensures that only unique posts are added to the list to avoid duplicates, using a set to track seen post IDs. The fragment utilizes Firestore to manage data retrieval, ensuring that only posts the user has interacted with are fetched and displayed. It uses Firebase Auth to identify the current user, linking actions directly to their account.

MyLikesFragment uses PostAdapter to render liked posts dynamically. When the fragment is active, it fetches liked post IDs and then retrieves the full data, passing them to the adapter. The adapter then takes over, fetching additional user details for each post and handling user interactions such as likes, shares, and navigation to other app sections via intents (like opening Google maps).

This ensures a great user experience, where data flow is maintained across user actions and database interactions, proving a seamless and interactive interface for users to engage with content they like.

Messaging



MessageAdapter.kt manages the display of chat messages in a RecyclerView within the chat interface. It handles the layout and data binding for individual messages, ensuring that text, sender information, and timestamps are properly presented. Methods uses here are:
onCreateViewHolder(): Creates new views (invoked by the layout manager). Here, it inflates the message layout from XML using LayoutInflater.
OnBindViewHolder() Binds data to each ViewHolder. This method sets the message text, sender's name. Timestamp, and profile image for each chat bubble. I used GLide for image rendering because of its efficiency in caching and loading images dynamically.
getItemCount(): Returns the total number of items in the data set held by the adapter, which is necessary for the layout manager.

updateMessages(): Updates the dataset within the adapter and notifies it to refresh the view. This method is crucial for displaying new messages as they are received or fetched from the database.

ChatRoomFragment.kt handles the user interface for a chat session. It sets up the RecyclerView for displaying messages and manages sending new messages.
onCreateView(): Inflates the fragment's layout, sets up the RecyclerView with its adapter and layout manager.
onViewCreated(): Handles logic after the view is created, such as loading messages and setting up a button listener for sending new messages.
LoadMessages(): Fetches messages from Firebase Firestore and observes them in real-time with a snapshot listener, updating the UI accordingly.
SendMessageToFirebase(): Sends a new message to Firebase Firestore, including fetching the current user's details for inclusion in the message.

MessageFragment.kt is responsible for displaying a list of users that the current user can chat with. It fetches user data and populates a RecyclerView with user entries.
Methods used: onCreateView(): Inflates the layout and initializes the toolbar and RecyclerView.
loadUsers(): Fetches a list of users that the current user is following. It listens for real time updates to ensure the list reflects the current state of the database.
fetchUsersByEmails(): A helper method that fetches user details based on email addresses, ensuring that userIDs are correctly matched against their profiles in Firebase.

UserChatAdapter: is designed to manage the display of a list of users within a RecyclerView. THis list can represent friends, chat participants, or any collection of users. The adapter not only manages the layout of each list item but also handles user interaction by providing a clickable interface that triggers navigation to a detailed chat.
Methods used: updateUsers(newUsers, newUserIds) updates the adapter's data sources (userList and UserIds). It ensures that the list of users and their corresponding IDs are updates simultaneously and are of equal length to avoid inconsistencies. If the lists are mismatched in size, an exception is thrown to prevent runtime errors.

UserViewHolder: This inner class holds the references to the UI components within each item of list. It initializes these components and sets up a click listener that triggers the onUserClick callback with the ID of the clicked user. This design encapsulated the handling of view-related operations within the ViewHolder, adhering to good coding practices.

OnCreateViewHolder(parent, viewType): It inflates the layout for the individual components from XML, creating a new ViewHolder for each item. This method is called by the RecyclerView when it needs a new ViewHolder to display an item.

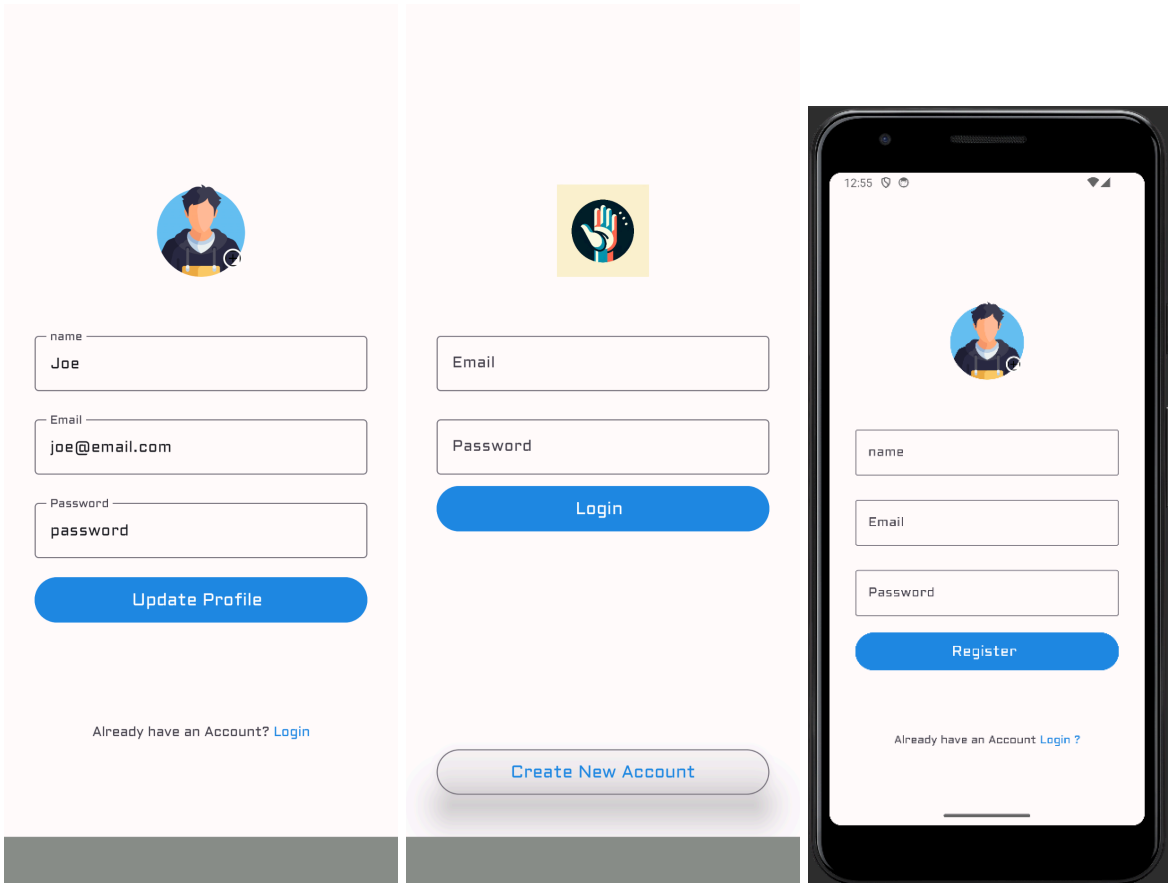
OnBindViewHolder(Holder, Position): This method binds data to an existing ViewHolder, setting the user's name and loading their profile image using GLide. It ensures efficient memory and resource usage, especially in list views.

getItemCount(): Returns the total number of items in the adapter, necessary for the RecyclerView to know how many items it needs to manage.

Relationship and Workflow:
MessageAdapter is used within ChatRoomFragment to display chat messages. MessageFragment manages a list of chat participants and uses UserChatAdapter to handle the display. This separation ensures reusability of components. ChatRoomFragment and MessageFragment interact with Firebase Firestore to fetch and send data, utilizing real time data handling to enhance user experience.

UserChatAdapter is used in a fragment or activity that manages a list of uses, such as MessageFragment. This component sets up the RecyclerVire and assigns this adapter to it. It interacts with a data model (User class) and Firebase FIrestore to fetch and display user data dynamically. User interaction handled by this adapter via onUserClick can trigger navigation or other actions, facilitated by callbacks defined in the containing fragment or activity.

Profile:



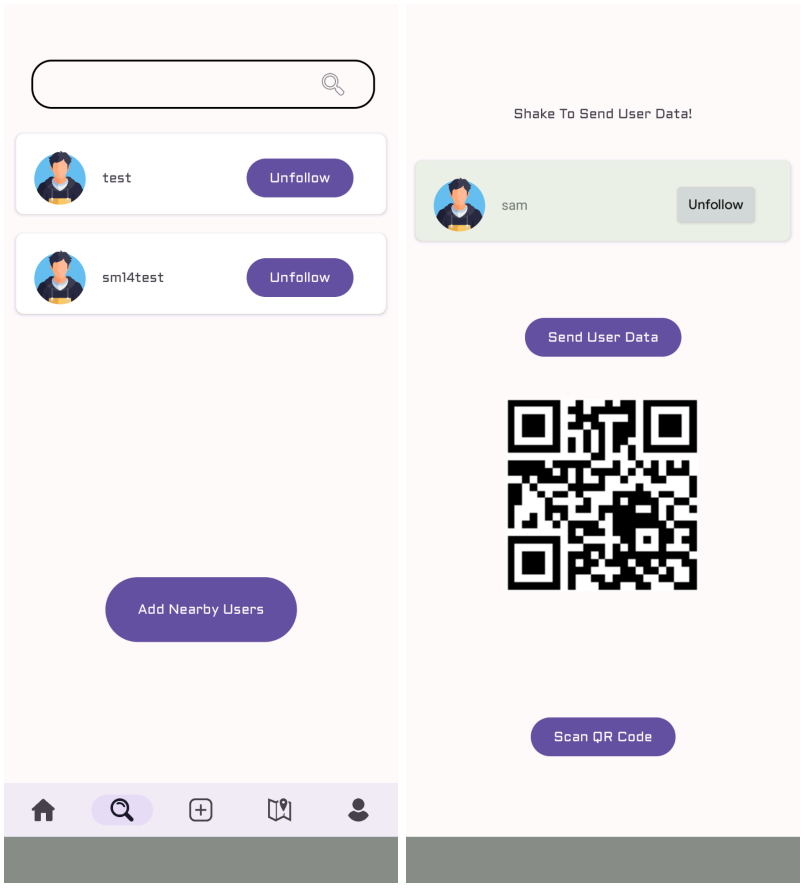
profileFragment.kt is designed to provide a space for users within the app where they can view and edit their profiles. This includes displaying their posts, likes, and other personal information. It also provides functionality to log out or update their profile.

Methods: onCreateView() inflates the fragment's layout and initializes UI components like buttons. It sets up listeners for these components to handle user interactions such as logging out or editing the profile.
setupViewPages(): initializes a ViewPagerAdapter to manage tabs within the profile for different sections like posts and likes. This method effectively segregates different types of user content, enhancing the user experience by organizing data neatly.
logoutUser(): Handles user logout by signing out from Firebase Auth and redirecting the user to the LoginActivity, ensuring the user is properly logged out and their session is cleared.
UpdateUserProfile(): Fetches and displays user information from FirebaseFirestore. It updates the UI with the user's name, bio, and profile image. If the user data is not found or incomplete, it provides mechanisms to handle these scenarios. It uses Picasso for image loading, which simplifies handling image caching and network retrieval.

SignUpActivity.kt serves dual purposes: registering new users and allowing existing users to update their profiles. It manages user input for creating or updating profiles and communicates with Firebase Firestore to store user details.
Methods used: OnCreate() sets up the activity layout and initializes form fields and buttons. Depending on the mode (new registration or update), it adjusts the UI and functionality accordingly.
RegisterForActivityResult(): Manages the result from image picker intents, allowing users to select and upload profile images. Which are then uploaded to Firebase Storage using a helper function. For new registrations, it collects the user's details and creates a new account using Firebase Auth. For updates, it simply updates the existing user's details in Firestore.

LoginActivity.kt handles user authentication. It allows users to log into their accounts using email and password. It also provides a link to the SignUpActivity for new users to create an account.
Methods used: onCreate() sets up the activity layout, initializes input fields and buttons, and sets up click listeners to handle the login process and navigation to the sign up screen.
Login Process: takes the email and password from input fields, uses Firebase Auth to authenticate the user, and handles success or failure scenarios. This activity primarily interacts with Firebase Auth to manage user sessions. It uses basic error handling to inform users of any login issues, such as incorrect credentials or network errors.This architecture ensures that user data is consistent and securely handled across different parts of the app, from logging in to managing profiles.

Search:



SearchFragment is a fragment that allows users to search through users that they are following, and also contains a button leading them to the AddUserActivity to follow new nearby users. The fragment employs a recycler view implementing the SearchAdapter that pulls up user data with a toggle button to unfollow each user that you are already following.

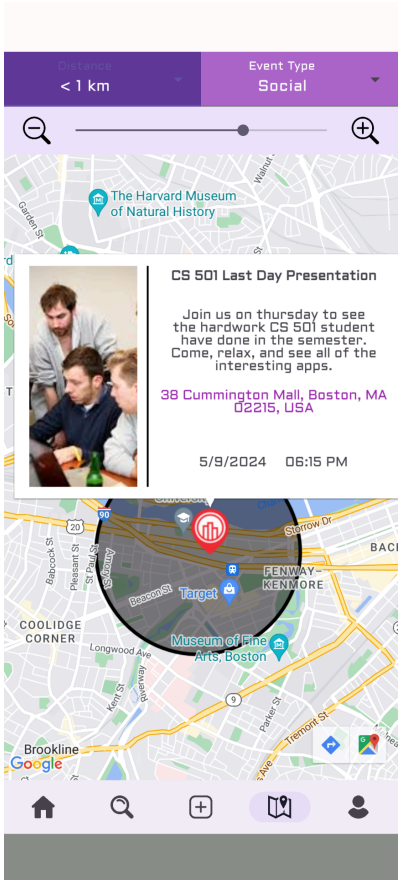
AddUserActivity is an activity that allows users to connect with nearby users. The activity implements Google Play services Nearby Connections API to handle bluetooth range communications, and calls the goQR.me QR code API to generate a QR code containing the user's data. The activity also implements the scan activity in the ZXing library to scan and parse QR codes. When a connection has been established between two users, they can shake their devices ("high five") or tap a button to send over their user data. User data can also be retrieved by scanning another user's QR code. Once user data has been received, it will populate a recycler view using the SearchAdapter allowing the user to follow or unfollow users whose data they have received.

Once the activity starts, we use the NearbyConnectionsAPI to start broadcasting the user's device under the "HiFive" service ID. This is displayed to the user in a textview saying "Searching for nearby users…" Once another broadcast of a nearby user is detected, the

device sends a connection request to the other device. Once the connection request is accepted, a toast is raised, stating that the user has connected to another device, and the textview is updated, prompting the user to shake their phone to send over their user data. This shaking gesture was implemented to emulate our initial user story of “High Fiving” to connect users. Connected users can then shake their device or press the fallback button to send their user data, which will populate the SearchAdapter recycler view of the other device with the same bar used in the search fragment with a toggle button to follow / unfollow that user.

Users can leave this activity by pressing the back button, and onResume(), the search fragment will update the list of followed users with the newly followed / unfollowed users. Additionally, the connectivity endpoints will be terminated to prevent battery drainage by the transceiver usage from the NearbyConnectionsAPI

Maps:



MapsActivity:
This is accessed when the user needs to add a location to an event post from the PostActivity. It inflates the activity_maps layout which uses Google Maps API to generate a map. The map is centered around the user's last known location which is as obtained from the phone's GPS location. The user is prompted to allow location permissions when they first launch the app. The location is stored in the MapsViewModel.
The user can select a location using a marker on the map interface and then send it back to the PostActivity for post creation. It automatically detects the address of the marker using geocoding. The user must choose a valid location (addressable) or they will not be allowed to submit.

MapsFragment:
This fragment is accessed from the navigation bar at the bottom of the app's main flow by tapping the map icon. This generates a map using the Google Maps API in the fragment_maps layout. This map is also centered around the user's last known location. Using stored posts in the FireStore database, it populates the map with interactive markers at each event's location with the option to filter based on distance and type. Pressing on an individual marker brings up an infobox that shows more details for the event.

MapsViewModel:
This stores the user's last location and is utilized by both MapsActivity and MapsFragment.

API's used:

goQR.me API: A simple API that takes https requests to encode data as a QR code, which the endpoint returns as a PNG. We utilized this API instead of processing the data locally on the device because it was very straightforward to implement, as we simply used the Picasso library to load in the image directly from the endpoint.

NearbyConnections API: NearbyConnections is an abstraction layer for the local communication technologies present on android devices. It combines NFC, bluetooth, and short range wifi transceiver data to connect nearby devices based on a variety of connection strategies (including peer-to-peer direct, star, and cluster topologies). First, a service ID and UID is broadcast from devices. When nearby broadcasts are detected by a device searching for other devices, it will then send a connection request to devices in accordance with its connection strategies. Incoming requests are then handled by the device's connection strategy, and compatible devices (by service ID and connection strategy) are connected to each other once this request is received. Once connected, devices can send each other data either as data streams or byte payloads. For our use case, we decided to transmit user ID information as a byte payload, which is then interpreted to populate the Search RecyclerView.

Google Maps Api: We used Google maps to show a map with all the events. Here you can filter base on distance and time. We also used google maps to add when a user wants to create a post. The user will click on the address field and maps will open. Here the user will pin point the location of the event.

Roadblocks/Surprises/Workarounds.

- As we may have mentioned during our earlier project updates, the connectivity requirements of our core user story proved much harder to implement than we had initially expected. With much of the connectivity documentation online being deprecated, it was hard to find a nearby connection technology that worked with the android devices in our team's possession. Luckily, with a lot more research and testing, we came across a connectivity implementation that works, NearbyConnections API, and we have stuck with that technology from that point on.
- Initially, we encountered issues where not all posts had timestamps, especially if the posting feature didn't originally include this data. A workaround was to delete all the posts from the database, and make a function to ensure that every new post automatically includes a timestamp upon creation to maintain consistency.

- Another issue that we encountered is that once we started adding posts, it took some time for it to load, especially pictures. To address this issue, we utilize Firestore's pagination capabilities to load data incrementally rather than all at once. Implementing lazy loading in the RecyclerView would further enhance performance, making the app more responsive.
- We had some challenges integrating live location data into posts, especially ensuring that the location is accurately captured and reflected in each post. A workaround we did to address this issue is to ensure that the location is fetched and stored only when the user opens the app to avoid discrepancies caused by location changes. We also implemented a clear UI/UX flow in the PostActivity where the user confirms their location.

Technical challenges:

- As the volume of posts and comments grew, it became crucial to ensure that the Firebase Firestore queries were optimized for efficiency. We started by implementing indexes and fine tuning the queries to fetch only necessary data, reducing the load time and improving performance.
- Implementing features like live messaging and comments was challenging, especially to ensure that all users see updates in real time without reloading the app. We used Firebase Firestore's real time capabilities, which push updates to all devices automatically. This required careful structuring of my database to ensure updates were efficient.
- Ensuring that the app worked seamlessly across different devices was crucial. We opted for responsive design principles in the front end development and we also tested extensively on multiple emulators.
- Maps/Location Synchronization: Due to the nature of the app with multiple activities/fragments, it was hard to ensure the user's location was captured correctly. This required careful passing of data through a combination of intents and a Viewmodel to make it work.

What would you do differently?

- Instead of using .get() for fetching posts, we would use real time listeners like .onSnapshot() that automatically update the UI upon data changes. This method minimizes the need for manual refreshes and can improve the user experience by displaying updates instantly.
- The current error handling primarily logs issues or displays Toast messages. We would implement a more robust error handling strategy that might include retry mechanisms or error recovery strategies. Enhancing user feedback beyond Toast messages, perhaps with dialogs or status indicators, would improve the user experience, especially in failure scenarios.
- The application retrieves posts and comments from Firestore on each invocation without any caching strategy. Implementing a caching layer would decrease load times, reduce

network usage, and provide a smoother and more responsive user experience. This could be achieved using tools like Room for local data storage which synchronizes with the remote database.

Future Work

- Continuously refine the user interface based on user feedback, this can significantly improve usability and aesthetic appeal. Some things we can consider implementing are Material design components and transition. This would enhance the visual appeal and functional smoothness of the application.
- Integrate real time functionalities such as instant notifications when new post are added or updates. This could be achieved using Firebase Realtime Database or Firestore's real time capabilities to keep the user experience dynamic and engaging.
- As the content grows, implementing more advanced search and filtering options will be essential. This could include searching by keywords in posts, or filtering by date. Such features would enhance navigation and user engagement.
- Expanding social functionalities such as tagging other users in comments or posts, or even creating private groups or forums within the app could significantly increase user interaction and retention.
- Implementing analytics to track user behavior and feedback mechanisms within the app would provide valuable insights into how users interact with the app. This data can drive future improvements and feature additions.
- Enhancing the app to function offline by caching data locally would allow users to access posts and comments without an internet connection. This feature could be particularly valuable in areas with poor connectivity.
- Integrating with other APIs and services could enrich the functionalities. For example, integrating with weather services for event based posts or using AI to suggest captions or tags based on the post content.
- Implement security features such as one time tokens for use in user data transfer, (e.g. make the QR codes only work for a certain amount of time, instead of encoding the plaintext UID)

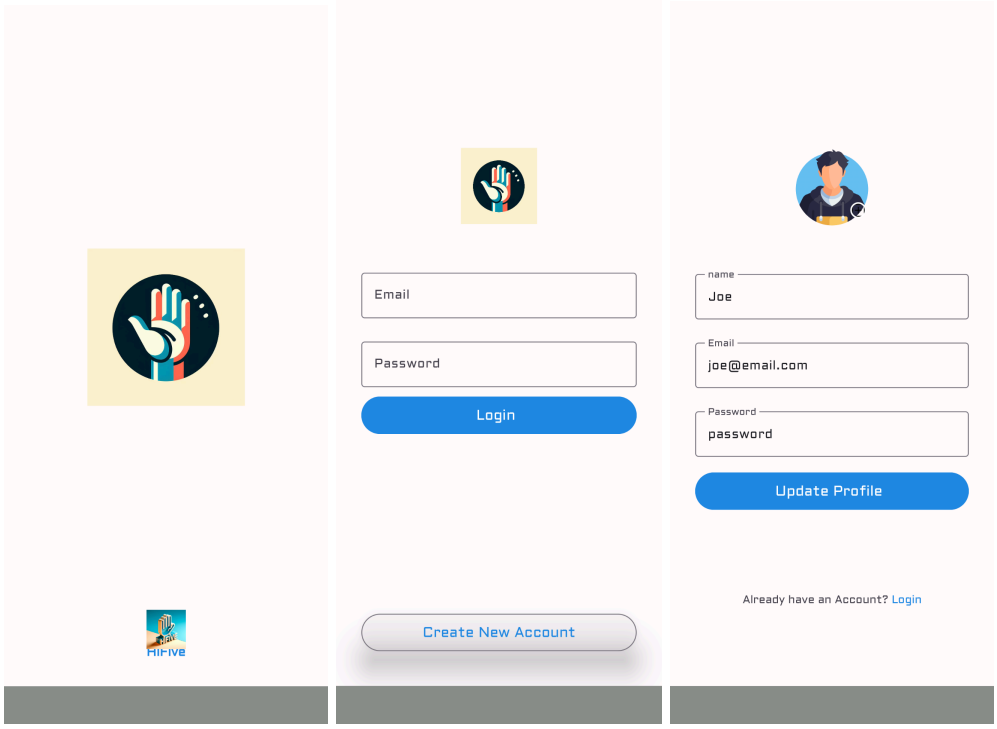
Bonus features implemented:

- Proper request of device permissions. ✅
- Proper interfaces for communication if using Fragments. ✅
- Use of RecyclerView or CardView where appropriate. ✅
- Good use of Menus. ✅
- Using Gestures and/or Accelerometer correctly (if needed). ✅
- Targeting multiple locales (must also do it well)
    - eg, English, Spanish. ✅

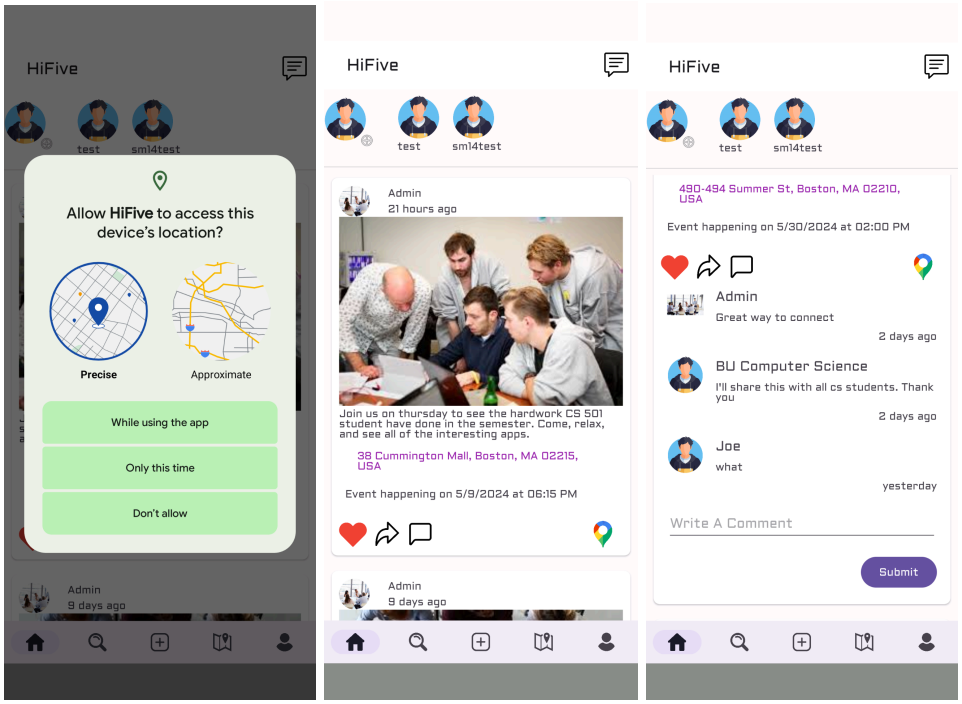
References:

Implement Firebase Auth: https://firebase.google.com/docs/auth/web/start
Retrieving Data: https://firebase.google.com/docs/database/admin/retrieve-data
Read and Write Data: https://firebase.google.com/docs/database/android/read-and-write
Upload files with Cloud storage: https://firebase.google.com/docs/storage/android/upload-files
How to use Firebase Storage:
https://stackoverflow.com/questions/39713875/how-to-use-firebase-storage
Save images to the Firebase Storage: https://www.youtube.com/watch?v=CWSiX_KzP4o
How to message users: https://www.youtube.com/watch?v=V9li7YP1NWg
Login and registration:
https://www.youtube.com/watch?v=tbh9YaWPKKs&list=PLlGT4GXi8_8dDK5Y3KCxuKAPpil9V49rN
Firebase comments system: https://www.youtube.com/watch?v=5A8KlFUgcAg
Instagram clone using firebase (Inspiration):
https://www.youtube.com/watch?v=3dsegOkif3Y&list=PLxefhmF0pcPnPuhlBPBq_FdG2GXpgrhVJ
Google Maps implementation:
https://www.youtube.com/watch?v=pOKPQ8rYe6g&list=PLHQRWugvckFrWppucVnQ6XhiJyDbaCU79
Google Maps Platform Documentation | Maps SDK for Android | Google for Developers
Save data in database:
https://www.youtube.com/watch?v=N4d4_zFR1nw&list=PL_QrnahGtqYAtYIoGod00kutYTvhq4HMU
Nearby Connections documentation:
https://developers.google.com/nearby/connections/overview

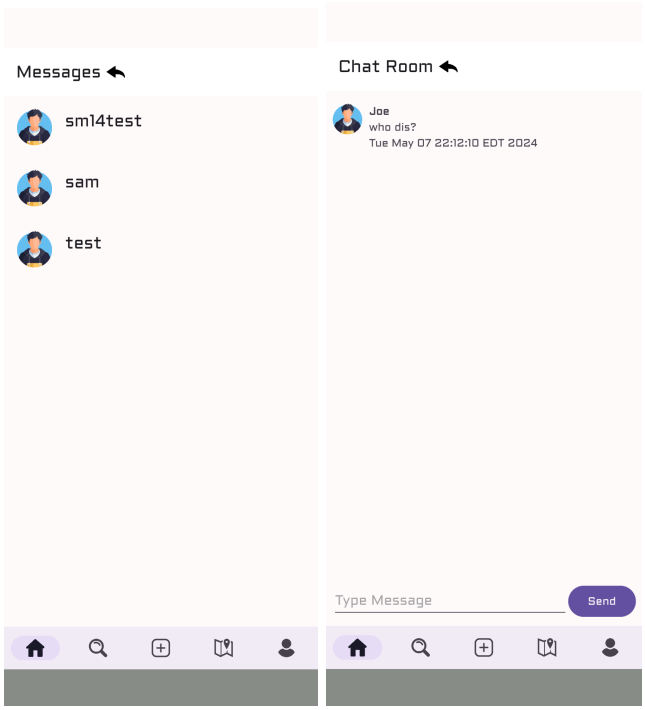
App Walkthrough:



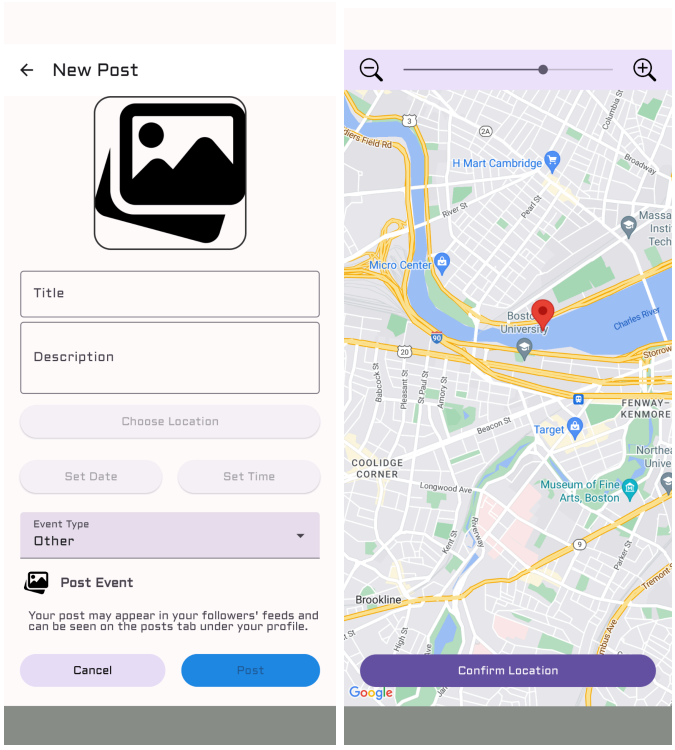
Home Page



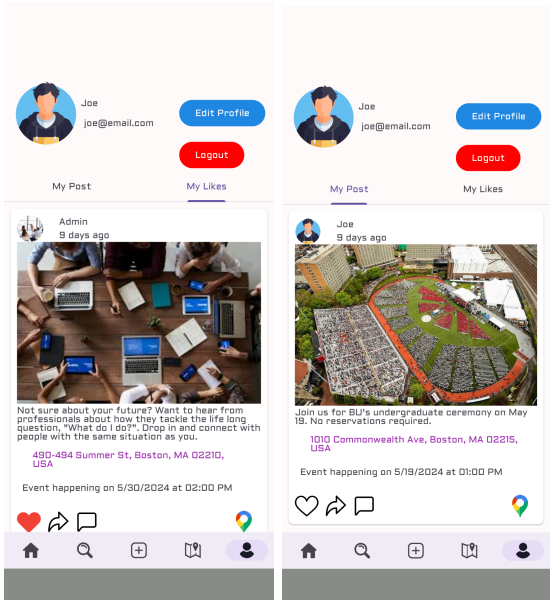
Message your friends:



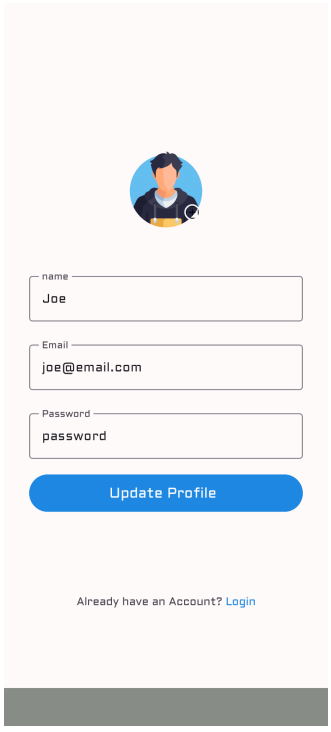
Add Post:



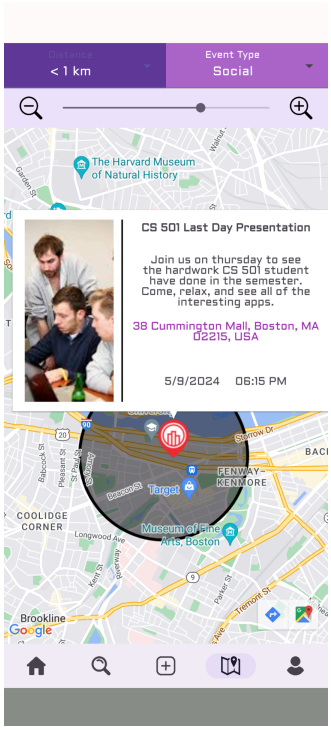
Look at your likes and your posts:



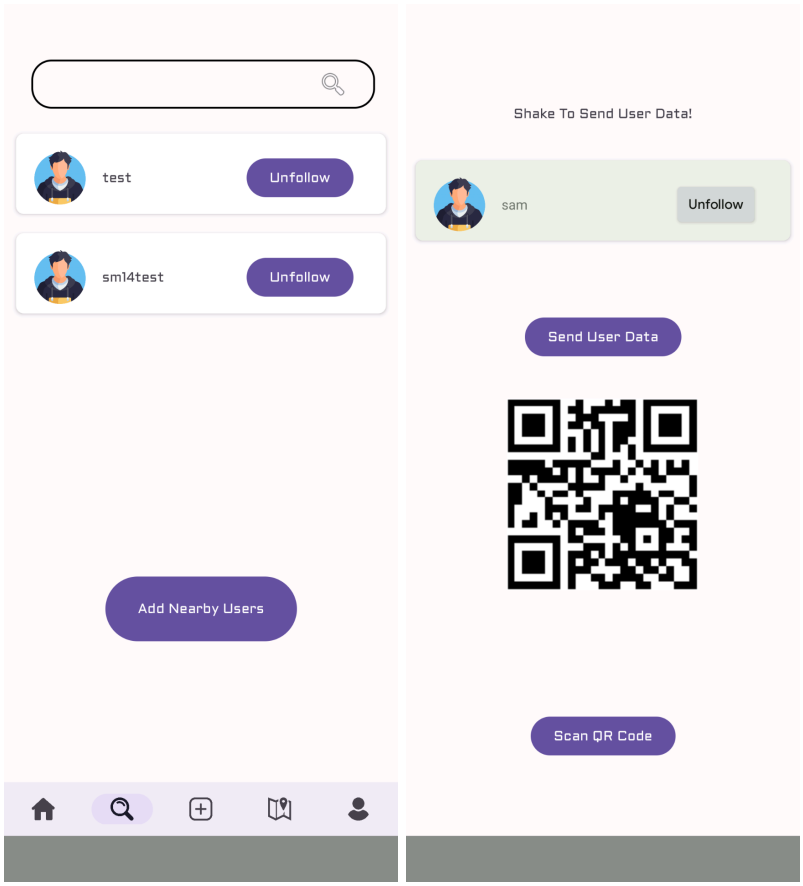
Update your profile:



Look at events near you:


Distance
< 1 km
Event Type
Social
The Harvard Museum of Natural History
CS 501 Last Day Presentation
Join us on thursday to see the hardwork CS 501 student have done in the semester. Come, relax, and see all of the interesting apps.
38 Cummington Mall, Boston, MA 02215, USA
5/9/2024 06:15 PM
Target
FENWAY-KENMORE
COOLIDGE CORNER
Museum of Fine Arts, Boston
Brookline
Google

Add users:


test
Unfollow
sml4test
Unfollow
Add Nearby Users
Shake To Send User Data!
sam
Unfollow
Send User Data
Scan QR Code