Team 2

Design Document

Dhruv Bihani - 93378453; Ishita Gupta - 99817512; Jayati Gupta - 34268201;
Nick Chen - 80611197; Manjot Singh -

Video

Link to our video presentation: https://youtu.be/nstYRlauqK4

Prototype Recap

In our prototype for the FoodHood app, we've designed a system for a mobile application that allows users to easily share their leftover food with others. The main interface includes a scrollable feed of food posts on the home screen, supported by a robust Firebase backend for data management. In terms of functionalities, we included a full range of frontend and backend features such as the ability to post and view items, and manage user's profiles with a user registration system.

System Architecture Design

We are using the **Client-Server** architecture pattern for our project. This is because the client-server model is a well-established and widely utilised model where the workload is partitioned amongst the *clients* (requesters of a service) and the *servers* (providers of a resource or service).

Our project, in particular, uses **Flutter** and **Firebase** for backend services.

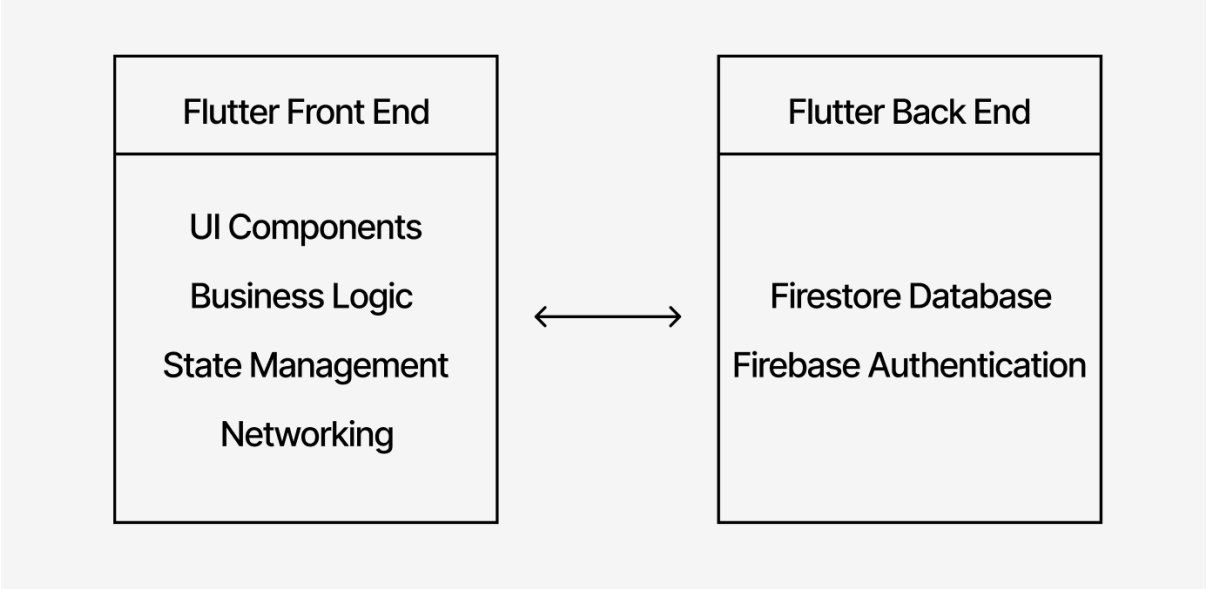
Thus, the client-server architecture has great benefits for our project needs such as:

1. Separation of Concerns:
    a. *Client (Frontend):* Responsible for presenting data to the user and handling user interactions.
    b. *Server (Backend)*: Manages data storage, processing, and business logic. It responds to client requests, performs computations, and interacts with databases.
2. Scalable - We can scale the client and server independently.
    a. If the users increase, then we can scale the server infrastructure without affecting the client.
3. Centralised data management - All the data is stored on one server.
4. Resource utilisation - Clients can run on devices with low resources while servers run on machines with larger processing powers.

The components we have written until now are:

- Flutter for front-end (Client):
    - UI Components: Developed using the Flutter framework.
    - Business Logic: Written in Dart language, handling user interactions and the data processing.
    - State Management: Utilises the built-in StatefulWidget and StatelessWidget of Flutter's state management solutions.
    - Networking: Communicates with the Firebase backend using the *cloud_firestore* and the *firebase_auth package*.
- Firebase for back-end (Server)
    - Database - Firestore is a database provided by Firebase and it stores our data in collections and documents.
    - Authentication - Firebase authentication is used to manage user login and sign-up and sign-out.

Architecture Diagram



Design changes

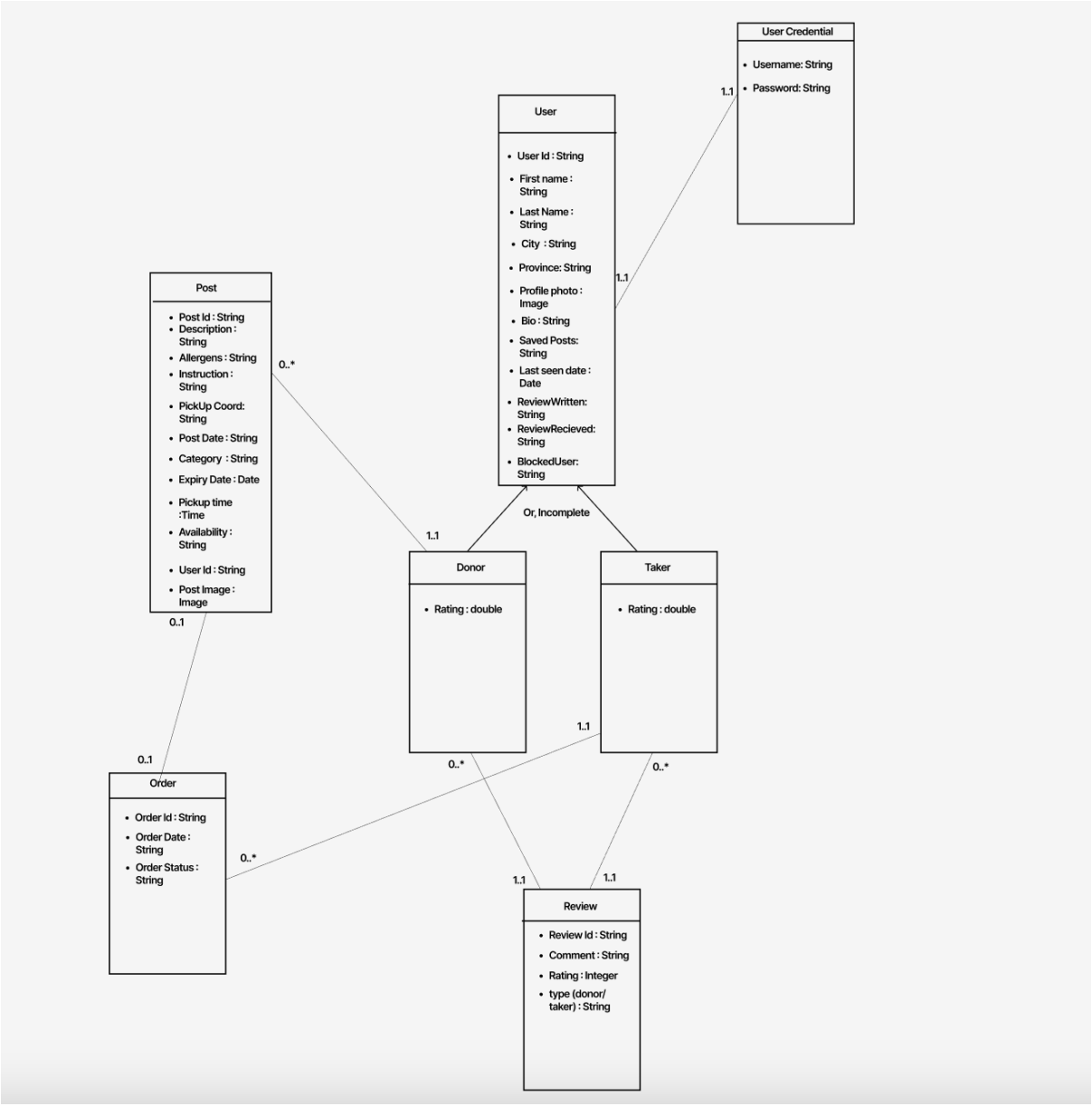
Initially, we wanted to try using the Micoservices system architecture as the concept of having multiple components all act as independent services felt modular and efficient, but we quickly realised that our features could not be independent and all heavily depend on one another. *This is why we switched to client-server architecture early on.*

We also initially had Networking in the Firebase Back-End as we wanted to group everything Firebase-related together in the back-end, but while creating this

document and doing more research, we understood that Networking is the layer we use to connect the back-end to the front-end, and hence it must live in the front-end.

Database Design

ER Diagram



Explanation of the ER Diagram:

We have a user table which stores all the information about the user like User Id, first name, last name etc. The user table is connected to the user credential table that has information about the username and the password. The user table has two sub groups

which are Donor (a user who creates a new post) and taker (a user who orders from a post). Both the Donor and the Taker have a rating associated with them. The Donor is connected to the post since they make a post about what food they want to give. The post stores all the information about a particular post which are the post Id, allergens, pickup instructions, etc. The Taker is connected to the Order which is further connected to the post. This is done because the Taker places an order from a particular post. The Order table stores all the information like order id, status etc. Both the Donor and Takers are connected to Review since they can provide a rating to each other after the order is completed.

In our Firestore, we have a collection called User. Under the User, the name of the document is the unique User Id . And under this document we have the rest of the information about the users like name, reviews written or reviews received. We have another collection called post_details. Under this collection, we have the document names such as the post id which are unique. Under this document, we have all the information about the post including the user id which connects the post to the user who made it. We have another collection called Orders. Under this collection we have documents which have unique names (Order id). It has all the information about the orders including the post id which connects it to the post and the user id which connects it to the user who ordered from a particular post. We have one last collection which is the review. Under this collection we have documents which have unique names (review id). These documents have all the information about the review like rating, comments, review type (tells us if the review was written for a donor or a taker) and also the user id which connects it to the user who wrote the review and to the users for whom the review was written.

Alternative design approach and Flaws

In our Firestore, we earlier had a collection called Order. The document name inside the collection was order1, order2 etc. Inside these documents we had various fields like orderID, order date, status, post id(to connect it to the post_details collection), uid (to connect to the user collection). Then we had a collection called Review. For each document under this collection we had, reviewid ,comment , rating , uid (to connect to the user collection). Then we had a collection called post_detail. Under this collection, we had documents which had description, title, instruction, first and last name of the person who created the post etc. Then we had one last collection called user. Under the documents for this collection we had firstname, lastname, city, province, userid, etc.

1. As is evident, the ids for any of the entities were not the document name, which means lookup times were considerably higher. For example, if the user is logged in they have access to their own uid, but if they want to look up their posts, they would have to search through every document in post_details collections and then retrieve where post_details.doc.uid = currentUser.uid. The

approach which we used (the correct approach) is much faster. The same applies to all the entities and their doc names.

2. In the alternative design, we just had one rating for the user. We separated the rating for the donor and the taker. By doing this, we made sure that a user can be rated separate values for different roles - as a donor or a taker - based on the review.

3. Previously we did not add the userid to the post_details documents. This led us to have first name and last names in both user and post_details. Also we figured out that first name should not be unique as two people can have the same first name or last name. Therefore we added userid to the post_detail documents and removed first and last name from there.

4. In the alternative design, we just had one rating for the user. We separated the rating for the donor and the taker. By doing this, we made sure that a user can be rated separate values for different roles - as a donor or a taker - based on the review.

5. We did not have the time stamps in the alternative design approach. Therefore we could not sort the post based on when it was made. We added the time stamps to the entities. By doing so, we know when the posts were created and their relevance in time. This helps in sorting the posts by time. It allows us to see the most recent reviews (if the user has changed their approach), look at recent orders, as well as notice trends over time.

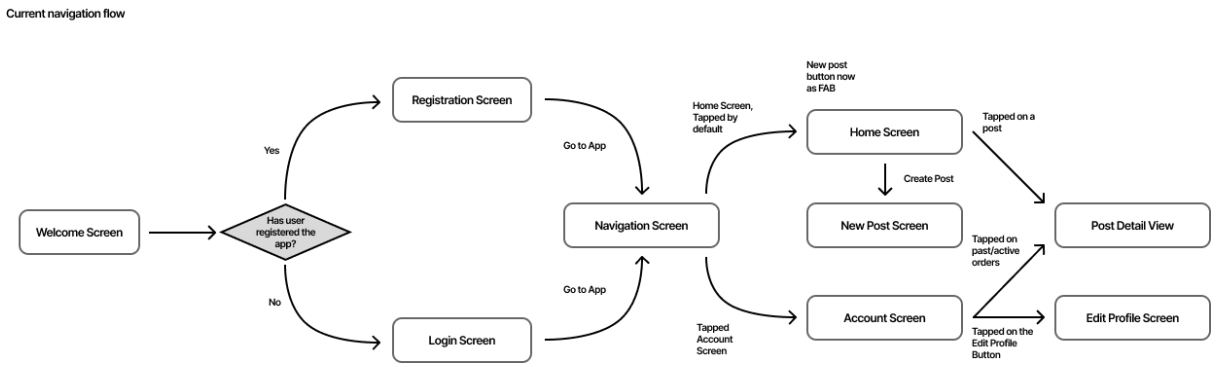
Feedback and changes

We received feedback from our TA, that it is better if we separate ratings for donor and taker because their role is different. That is when we created two subgroups of user - donor and taker, so that they can have separate ratings.

User Interface (UI) Design

Overview of Interaction Flow:

Current Navigation Flow



Explanation

The current navigation flow is designed with a focus on streamlining the user's experience when browsing through the FoodHood app. Upon launching the app, users encounter a Welcome Screen that leads them to either the Registration or Login Screen, depending on whether they are new or returning users. We added back buttons in both views that empower our users with the ability to leave either view if they accidentally reached a wrong page. This allows fast directing for users during login without adding unnecessary complexity.

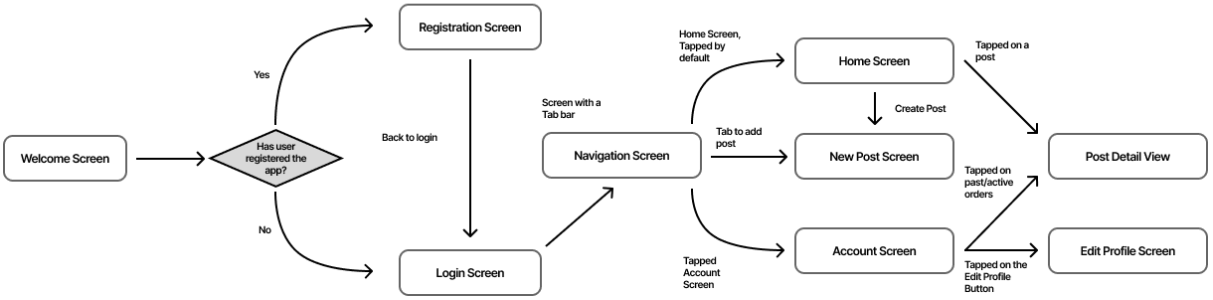
Once past the login or registration stage, users are taken to the Navigation Screen and from there directly to the Home Screen. The navigation screen overlays the main view of the app with a tab bar. The Home Screen is the page with the food posts listed, consisting of a search bar with a floating action button (FAB) for creating new posts. When a user taps the FAB button, they are directed to a create post screen, in which they enter the food post details.

Tapping on a particular post will open up the post detail screen where the user can see the description of the post. Navigating back from the post will go back to its original view.

When selecting the Account screen from the tab bar, the app will reveal an account screen with the profile detail and the order history. By tapping on the edit profile button it will allow users to edit their profiles by showing a form of profile details.

Alternative Navigation Flow Explanation:

Alternative navigation flow



In the alternative navigation flow, after the user is successfully registered, they are taken directly to the login screen again. This step aimed to enhance the security of the app but added an additional step in logging in to the app. Therefore, we decided to move with the current navigation flow where upon signing up, they are directly taken to the home screen.

In this alternate flow, the Navigation Screen includes a tab bar, which has buttons for switching between different pages for example home page, account page, create a new page etc. This, in turn, posed challenges to navigation within the app. Particularly, since the “create a post” page was accessed from the tab bar initially, there was no memory of the previous pages visited in the navigation stack and so clicking on “x” and “save” did not enable the user to go back. To address this issue, the design was changed to have the “create a post” page be accessible from the home screen via a FAB.

Design Choice Rationale:

The current design of the app is made to enhance user experience by making it easier for users to use it. It helps the user by reducing the effort and choices they have to make, making it less complex. The main screen of the app is quite essential which allows users to create posts, view lists of all posts in their city, along with the post details for each food posting. The tab bar allows us to This shows that the app really focuses on what users want to do most.

The other design option gives more ways to move around the app, but it's not as quick and easy for everyday use thereby reducing user experience . That's why we did not move forward with it. The chosen design makes the app easy and fast to use, which is what users need and wants.

Team 2

Recap of Features

Dhruv

1. HomePage (scrollable lists of all the posts) and testing (Front end):
    a. Created a homepage where you can view all posts created by a donor.
    b. Developed a post card class to serve as a placeholder for all posts, enabling retrieval and display on the homepage.
    c. For each post card, included a title, tags, the creator's name, and the creation date (all hard-coded as it was only a UI element).
    d. Designed a search bar UI.
    e. Implemented a filter button UI.
    f. Created Cupertino buttons for filtering by tags (UI), making them horizontally scrollable.
    g. Added a title at the top of the page.
    h. Conducted a test to verify if the UI widgets are properly built. The test checks for the presence of a search bar, Cupertino buttons, a scrollable list view, and text on the home screen.

2. HomePage (Backend functionality) and testing:
    a. Implemented a method to retrieve data from Firestore.
    b. Replaced earlier hardcoded values with data from Firebase, including titles and creation dates.
    c. Retrieved tags from a list separated by commas and displayed them on the post card.
    d. Assigned random colours (chosen from light green, light blue, light yellow, light pink) to the tags.
    e. Retrieved the user ID from the post collection and linked it to the User collection to obtain the first and last names of the post creator.
    f. Added a method to read multiple files from the database, creating a new post card for each file.
    g. Ran a query to ensure post cards are displayed in descending order by creation time.
    h. Made all post cards clickable, directing users to a page with more details about the post.
    i. Implemented search bar functionality, allowing users to filter posts based on text present in either the title or the tags.
    j. Created a test for reading data from mock Firebase (checking if the retrieved data is correct).
    k. Developed a test for the search bar (verifying if the user can input text and retrieve it from a post card generated by the search).

Manjot

Team 2

1. Configured no-reply email service for our app (back end)
    a. Created a no-reply account for our app, FoodHood, using firebase and configured the mail body.
    b. Configured the generation of dynamic links to simple webpages on firebase console.
    c. Should a user forget their password, they can provide their email address they used to sign up and they will receive a no-reply email with a dynamically generated link to a simple webpage where they can enter a new password. The webpage will reset their account's password. Functionality complete.
    d. When a user first signs up on the app, they receive a no-reply email with a dynamically generated link which takes the user to a simple webpage and verifies their email. If email is verified, it is recorded in the database.
    e. Since this was all configured on the firebase console, there was no automated testing possible for it (discussed firebase auth testing with Pragya during check-ins)
2. Created public_page.dart
    a. Created the UI for the public page, which displays a user's profile as it would be visible to others using the app
    b. The page fetches information from the database and based on secondary information (ratings), calculates the rating to display up-to-date score (if the user has been rated)
    c. The page has a "block" button, which can be used to block a user or unblock a user (if already blocked)
3. Wrote firestore_service.dart
    a. Wrote the .dart file that connects the main application to the firestore, configures the app and initiates an instance to read/write/query the database
    b. Wrote generalized functions to read from and write to the database, which are used in all of the other documents that interact with the database
    c. Completed the test with other teammates.

Ishita

1. Create a post Front-End implementation and testing
    a. Added input text fields for various post details.
    b. Included Cupertino search text fields for allergens, categories, and pickup locations. Resolved several bugs related to these search bars, completing this feature in the next phase.
    c. Integrated date pickers for selecting expiration dates.
    d. Addressed render flow errors and spent time resolving them.
    e. Implemented a time picker for choosing a pickup time.

   f. Fixed issues with the "cancel" and "save" Cupertino buttons not functioning as expected.
   g. Altered the design to add a "plus" button as a floating action button on the home screen for creating posts, rather than using a tab bar option.
   h. Developed a widget test for the front-end.

2. Create a post Back-End implementation and testing
   a. Added functionality to write post information to Firestore.
   b. Created a custom Cupertino chip widget for reuse in displaying data.
   c. Developed a reusable search bar component that searches through predefined lists, displaying selected items below the search bar as chips (using the custom Cupertino chip).
   d. Implemented a function in firestore_service.dart to write predefined lists of allergens, categories, and pickup locations to Firestore, for use in the create post screen's search bars.
   e. Modularized create-post code to enhance efficiency and resolved minor bugs.
   f. Established the mock_firestore_service file to simulate firestore_service for testing.
   g. Conducted a mock test to assess writing data to Firestore.

Jayati

1. Postings page UI implementation and testing
   a. Created a posting details page where users can see details about a post and reserve an order.
   b. Spent time modularizing code to allow for maximum efficiency as we code the UI for more pages.
   c. Created several components that will be reused as we continue to build out our UI such as an IconPlaceholder and types of Cards.
   d. Created changeable components depending on reservation status such as indicator tag and greyed-out button.
   e. Created UI tests that check the presence of certain widgets and buttons and if they operate correctly
2. Postings page backend and testing
   a. Connected the backend to my frontend using the methods created by other team members
   b. Changed my modularized code to have widget variable-passing to allow for reading from Firestore
   c. Changed how to display time since a user posted and read dates from Firestore.

    d. Created backend tests that mock firestore-service and added mock test to test reading and writing data to firestore.

Nick

1. Account page backend and frontend implementation with testings
    a. Created the Profile card that listened to the firebase changes and display user info such as name/email automatically whenever user made an edit to the profile
    b. Create the Order card component with a context menu while the user long presses the card to summon quick actions like edit order. The order card also has implemented a similar UI and behaviour of the posting card.
    c. Created a section where users can see their current order segmented by their states (Active/Past Orders). The user can view the posts that are currently active.
    d. Implemented UI and other edge cases for postings states (i.e. placeholder where there's no post etc)
    e. Ensure the smoothed connection between firebase and flutter code. Added appropriate loading indicator while content.
    f. Created widget test and backend connection test to ensure the robustness of the firebase read and write functionality.
2. Edit Profile backend and frontend implementation with testings
    a. Create forms and pickers allowing the user to edit their names, description, locations. Implemented photo pickers to select profile pictures.
    b. Location data and profile information are now fetched from the firebase.
    c. Implemented edit functionalities to allow users to edit their profiles and bios.
    d. Created widget test and functionality test to check if the connection between the flutter app and database is live.