udharKharcha

PROJECT REPORT

Submitted by:

Kanishk Chaudhary (ID:11840630)

Saransh Pushkar (ID: 11841000)

Shubham Arora (ID: 11841080)

Guided by:

Dr. Gagan Raj Gupta

INDIAN INSTITUTE OF TECHNOLOGY BHILAI

CS559 - Computer System Design

WINTER 2022

Introduction:

It is a very common scenario where one or more persons pays the bills for the group in their group's activities. In other events, some people from the prior event might not be involved or there might be new faces. As this goes on, it becomes difficult to keep track of who lent whom, when and why.
udhar Kharcha is our project to tackle such problems to track and settle debts and expenses based on events.

GitHub Repo Link: Link
APK File Link: app-release.apk

Features:

1. **Mobile number based authentication:**
    - Authentication is OTP based and password-less
    - The users will receive a one time password for both login and signup making mobile numbers the unique identification parameter for a user
2. **Bill Split:**
    - Enables users to split expenses in any transaction event.
    - Users can know how much one has to pay or take to settle the debts in events where there are one or more payers.
    - Each debt taker will be notified and the transaction will be valid only on their approval.
3. **Personal Expenses:**
    - Users can add events to maintain history of personal expenses along with the option to add a short description about where the money is used
    - Every event is associated with a date when the event is added
    - Whenever a user is involved in the bill split, then the bill amount corresponding to that user will be added to its personal expense, if positive
4. **Analytics:**
    - Users can analyze their weekly/monthly personal spending using a interactive bar graph
    - For the weekly option, the graph displays spending for the current week and previous 4 weeks with a week defined from Monday to Sunday
    - For the monthly option, the graph displays spending for the current month and previous 4 months
    - On clicking any bin of graph, the events associated with that time range are shown just below the graph
5. **Notifications:**
    - User will receive push notifications whenever:
        - involved in bill split and debt is added for it

    - other user approved the debt request
    - other user paid you money to settle debt
- There is also a bell-shaped icon at the top-right corner which displays all the past notifications to user

Database Schema:

We have used Amazon Keyspaces which is a scalable, highly available, and managed Apache Cassandra compatible database service. All the tables and their schema is shown below:

1. Event Details:

Column name	Type
event_bill	map<text, double>
event_detail	text
event_id	text
event_payers	map<text, double>
event_time	timestamp
pairwise_udhar	map<text, double>

2. User ID to FCM token:

Column name	Type
fcm_token	text
user_id	text

3. Notification Details:

Column name	Type
notification_body	text
notification_id	text
notification_time	timestamp
notification_title	text

4. User ID to Personal Expense events mapping:

Column name	Type
event_ids	list<text>
user_id	text

5. User 2 User details:

Column name	Type
event_ids	list<text>
from_user_id	text
pair_id	text
to_user_id	text
total_amount	double

6. User ID to Notification ID(s) mapping:

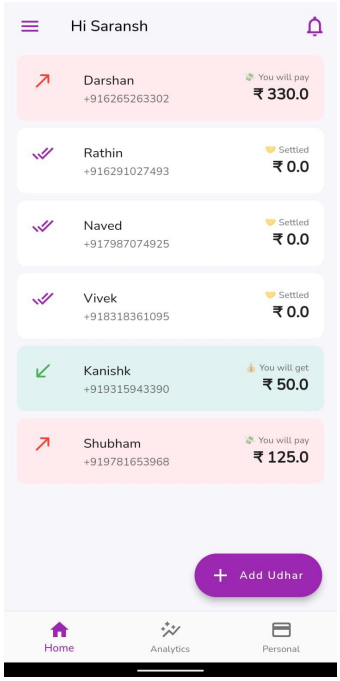
Column name	Type
notification_ids	list<text>
user_id	text

7. User Profile:

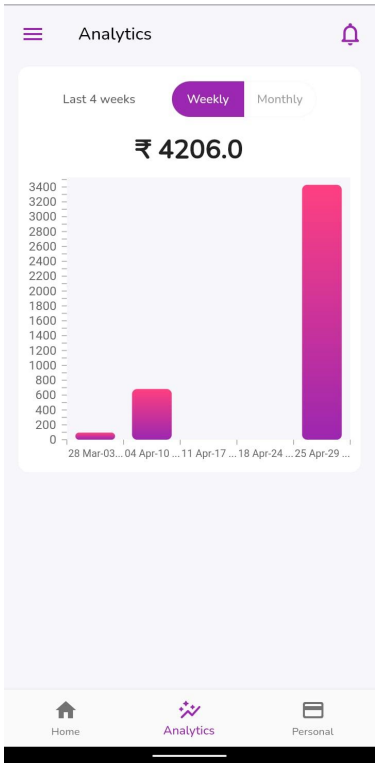
Column name	Type
phone_no	text
upi_id	text
user_id	text
username	text

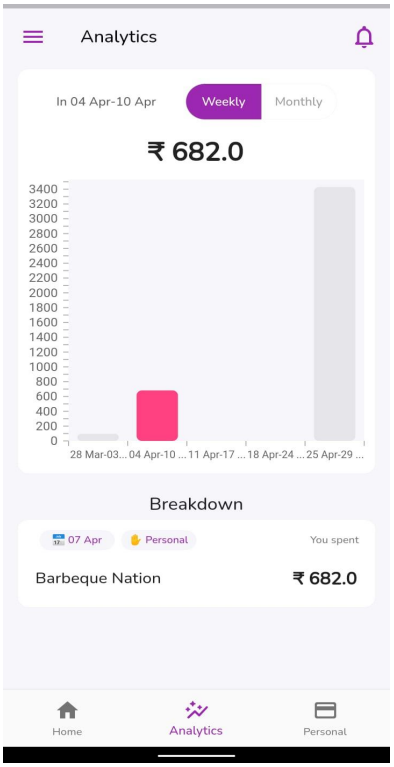
Implementation Details:

1. **Frontend:**
   The app is written in **flutter**. The interface is a 3 page navigation namely home, analytics and personal expenses. All the dependencies are in **pubspec.yaml** file.
   - Home Page (home_screen.dart)

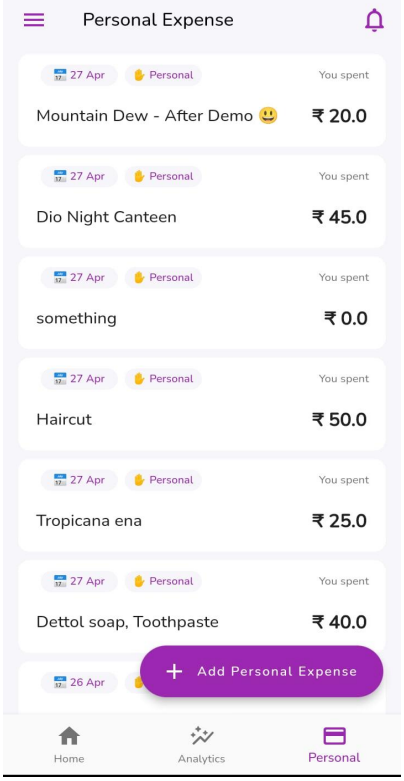


- Analytics Page (analytics_screen.dart)

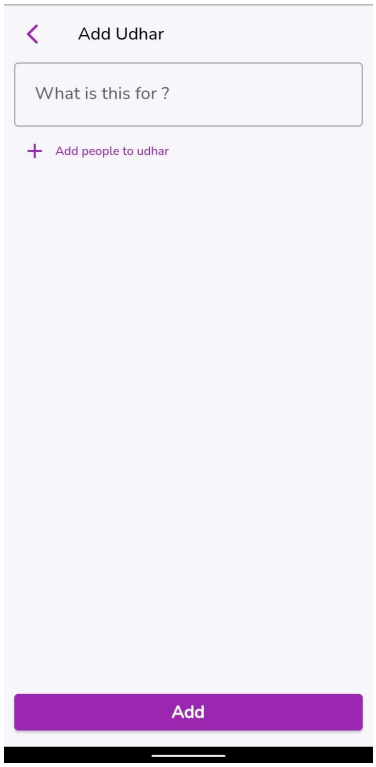


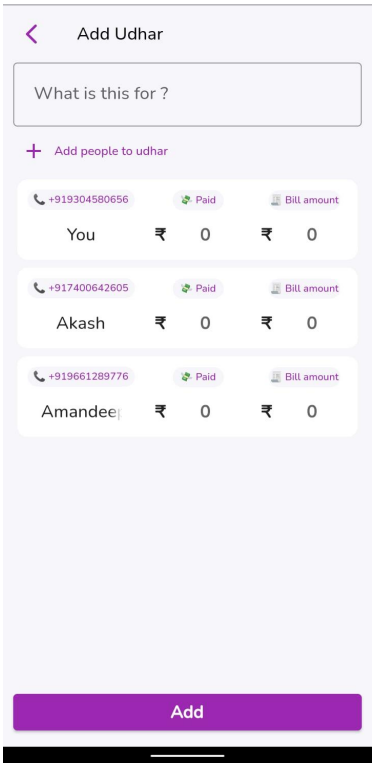


- Personal Expenses (personal_expense.dart)

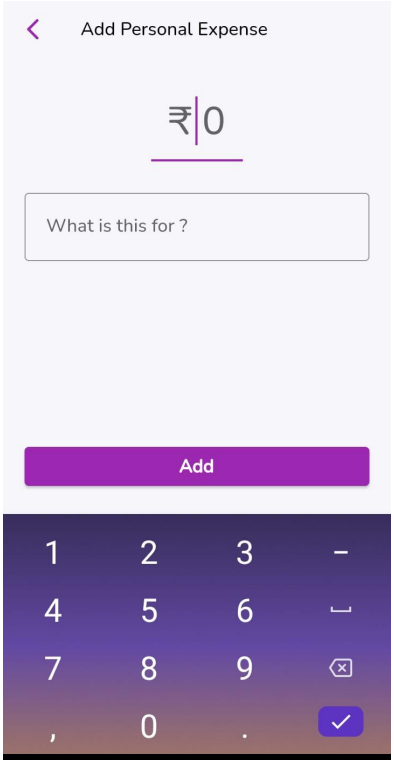


- Bill Split or Add Udhar (add_debt_screen.dart)

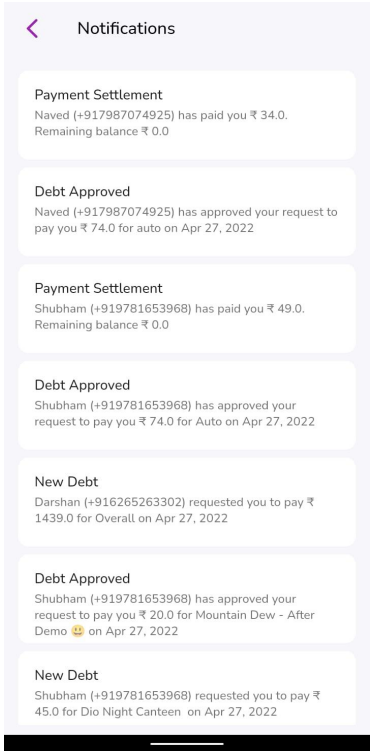




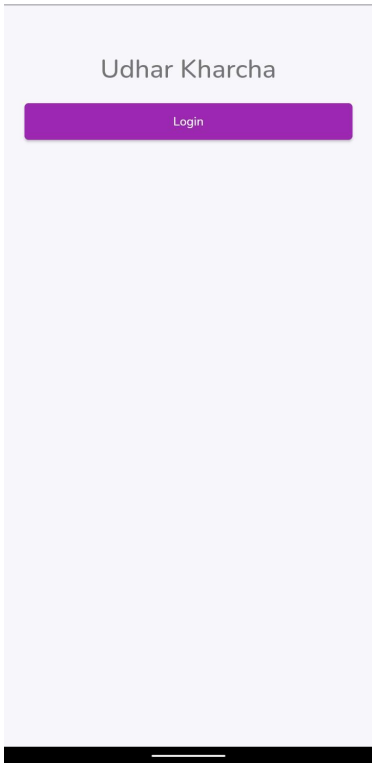
- Add Personal Expense (addPersonalExpense_screen.dart)



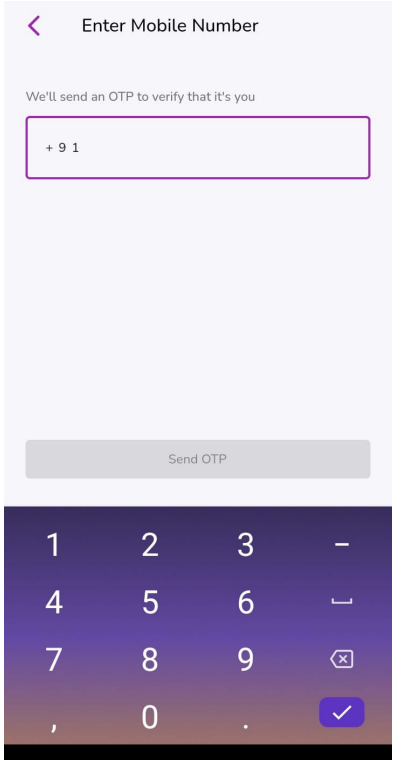
- Notifications Page (notification_screen.dart)



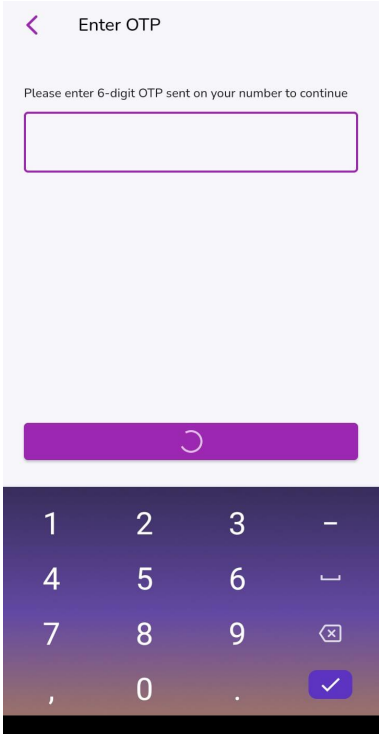
- Welcome Screen (welcome.dart)



- Login Screen (login_screen.dart)



- OTP Screen (opt_screen.dart)



2. **Backend:**
   We have used **3 t2.micro** EC2 instances with an ALB in front of them and Amazon Keyspaces as database service. For push notifications we have used **Firebase Cloud Messaging** of firebase**.**

   API Details:

   1. **'`/signup`':**
      - This takes **phone_no**, **username** & **upi_id** as input
      - **user_id** is calculated as hash of **phone_no**
      - An entry is created in **user_profile** table
   2. **'`/update_token`':**
      - This takes **phone_no**, **fcm_token** as input
      - **user_id** is calculated using **phone_no** and the token is updated in **user_fcm_mapping** table
   3. **'`/get_pair_details`':**
      - This takes 2 phone numbers, one who is using the app and one of the person on whose name the user has clicked/tapped
      - The event id(s) between pair of users (say **A** and **B**) found from **split_bills** table
      - Events are retrieved from both ways:
        - first in which **from_user_id** is of user **A**, means events in which **A** has given money to **B**
        - then in which **from_user_id** is of user **B**, means events in which **A** has taken money to **B**
      - Then the corresponding events are retrieved from the **event_details** table and sent as response along with additional flag whether user **A** has given money or taken in any particular event
      - Negative value in **pairwise_udhar** means the debt has not been approved yet
   4. **'`/approve_udhar`':**
      - This takes user phone no who is approving the debt, phone no of user who has given money and **event_id** associated with the debt
      - **pairwise_udhar** for corresponding event in **event_details** table for corresponding pair of users is changed from negative to positive
      - **total_amount** for corresponding pair of users is updated in **bill_split** table
      - At the end **fcm_token** of user whose debt is approved is retrieved and firebase cloud messaging service is called to send push notification
   5. **'`/pay`':**
      - This takes phone no of user who is paying money, who is receiving money and amount
      - If the amount user sending is valid and consistent with the data then **total_amount** for corresponding pair is updated in **split_bills** table

- Notification is sent to user who is receiving the money

6. **'`/bill_split`':**
    - This takes a map of participants paid in that event, a map consisting of each participant's bill amount and event name.
    - Udhar_takers and Udhar_givers are calculated based on each participant's paid amount and bill amount i.e if paid amount > bill amount then this user will receive money and if paid amount < bill amount then user has to give money.
    - Then Udhar_takers and Udhar_givers are passed to our min transaction algorithm which will return the minimum number of pairwise transactions to settle the current event.
    - Next, data is populated in the database in **event_details** and **split_bills** table. Only those records are stored where the payer has to take from the lender.
    - If a row already exists in **split_bills,** then the **event_id** associated with the event is appended to the **event_ids** in **split_bills** table**.**
    - **Min transactions algorithm:**
        - Between any two groups of **udhar_takers** and **udhar_givers**. If sum(**group_udhar_takers**) == sum (**group_udhar_givers**) then the min transactions required to settle these groups is len(**group_udhar_takers**) + len(**group_udhar_givers**) - 1.
        - Now using the above knowledge, the algorithm tries to create groups of **udhar_takers (group_udhar_takers)** and tries to satisfy the groups by creating **group_udhar_givers** for each **group_udhar_takers,** satisfying -> sum(**group_udhar_takers**) == sum (**group_udhar_givers**).
        - Optimizations:
            - Order of groups or the udhar_amount within the groups are not important.
            - Dynamic programming using bit masking and memoization is used to satisfy **group_udhar_takers** and make **group_udhar_givers** for each **group_udhar_takers.**
        - **Example: udhar_takers** = [30 ,40]
          Udhar_givers =[10, 25, 20 ,15]
          First we enumerate on **udhar_takers** to create all possible groups. i.e.
          [[30], [40]] and [[30,40]]
        - Now consider [[30], [40]] and try to satisfy each group, first is [30], this is satisfied only by group [10, 20] in **udhar_givers.** Next is [40], this is satisfied only by the group [25, 15] in **udhar_givers.** So total transactions are (1+2-1) + (1+2-1) = 4.

        - Now consider [[30, 40]] and try to satisfy each group. The first and only group is [30, 40], this is satisfied only by the group [10, 20, 25, 15] in **udhar_givers.** So total transactions are (2+4 - 1) = 5.
        - In this Example we get min transactions = 4 between groups ([30] and [10,20]) and ([40] and [15, 25]).
        - Further improvements:
            - This algorithm runs in exponential time complexity and guarantees minimum transactions.
            - We can use approximation algorithms to reduce the time complexity for saking the guarantee for minimum transactions.
    - Notification is sent to all users who have taken money with information to pay how much amount and to whom
7. **'`/get_udhars`':**
    - This takes phone no of user who is retrieving the information
    - Results are found in which input user has lent money and in which input user has taken money
    - Based the results found above total amount between input user and every other user in result is calculated and send as response
8. **'`/personal_expense`':**
    - This takes phone no of user who is adding expense, amount and short description of event
    - A new event is created in the **event_details** table and corresponding **event_id** is appending to list of current events of that user
9. **'`/get_personal_expenses`':**
    - This takes phone no of user who is retrieving information as input
    - From **personal_expenses** table list of events corresponding to that user is found and then the events are returned from **event_details** table
10. **'`/event_details`':**
    - This takes **event_id** as input and returns the bill and payment information of that event from **event_details** table
11. **'`/get_notification_details`':**
    - This take user phone no as input and returns all the notifications of that user
    - A **notifications_id** list is maintained in **user_notification_mapping** table and from that notifications, information is retrieved from **notification_details** table and is returned as response
12. **'`/analytics`':**
    - This takes **user_phone_no** and **type** (weekly/monthly) as input and returns the time range, events, amount sum in particular time range and total amount sum over all the weeks/months
    - All personal expenses' event_id(s) is found from **personal_expense** table

- This list contains event_id(s) of all personal expense events associated with that user, and we are interested in events lies in particular time range which is a sublist of retrieved list
- So we can use **binary search** to find the starting and ending index of this sublist since the event_id(s) are sorted according to time
- Then we retrieved the events corresponding to the sublist from **event_details** table and then aggregated according to type (weekly/monthly)

Design: *Design Link*