

Ledger

COMP 4521

Project Report

Group 7 – Group Ledger

Student 1:

LI FUK SANG

Student 2:

IP KA LOK

1. Introduction

1.1 Acknowledgement

Our team had noticed that some requirements for the **Project Report** overlaps with that of the **Project Proposal**. As our team had endeavoured to included as many details as possible in the project proposal, this project report contains *some* sessions that are taken directly from the initial project idea and the project proposal for your ease of reference. We will try our best, where possible, to distinguish what element had changed since the submission of the project proposal.

1.2 Problem Statement

Within a group of people, very often members of the group would make payment on behalf of the group. (For example, a person might go buy groceries for him and his roommates). This group now owes the person money. When these transactions happen often, settling the debt immediately is not an option. Recording the debt and settling in a later stage is also very troublesome.

See our **Initial Application Idea** for details. It includes detailed examples and use cases for our problem statement.

1.3 Proposed Solution

Our proposed application aims to provide an easy interface for people to record such kind of transactions. Every member of the group, with the app installed, will easily be able to see the debts incurred within the group. When it is time of settle the debt, our application also provides functions to automatically calculate how much each person should pay and receive in order to settle the debt.

In summary, our application aims to simplify the process of making transactions on behalf of a group of people.

2. Design and Implementation

This section details the design methodology as well as the completed user interface of our ledger application. All images are taken from the final app completed.

Each components stated below will correspond to a list of use cases. These use cases might be copied directly from our initial application idea. Some use cases are updated in this final report.

As some terminologies can be confusing, we will use the below scenario to illustrate a use case of our ledger application.

Peter lives with his roommates Bob and Alex. Today, Peter bought some pizza, which all three of them ate. As transactions like that happens all the time, Peter would like to record it down and settle the transaction later.

This example will be echoed as we demonstrate the usage of different components.

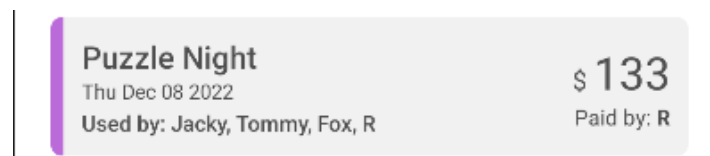
2.1.1 Transactions and Creating Transactions

Transactions

Type: **Object (Data)**

A transaction refers to a record where a person has paid for something on behalf of a group of people.

Transactions forms the backbone of our application. A transaction of not a page, instead it is a component which will be rendered in multiple pages. We will demonstrate how a user can add, modify, and settle transactions later in this proposal.

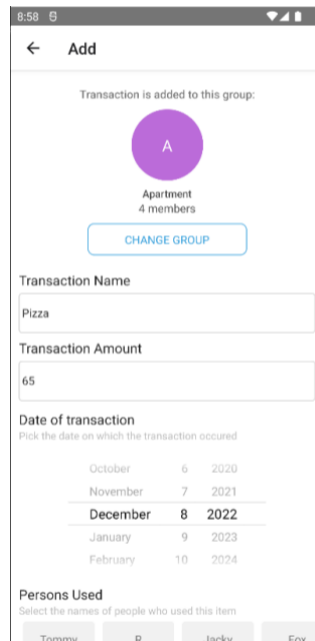


Create Transactions

Type: **Page**

Create transaction page is a page which will allow users to create and record transactions. The specific transaction will be added to a “Group” (See Group Page below). User would be able to input the following parameters:

- **Transaction Name:** What this record is for
- **Transaction Amount:** How much did this record cost
- **Person Used:** Who have used this transaction. This is important as these people would be responsible for paying for this transaction later when the user “Settles” the transaction.
 - *In our example, the person used would be Peter, Alex, and Bob.*
- **Person Paid:** Who paid for this transaction. Defaults to the user himself, although the user can also add transaction on behalf of other people. When this transaction is settled, the “Person Paid” will collect the money from the “Person Used”.
 - *In our example, the person paid would be Peter.*
- **Date of transaction:** When did the recorded transaction happen.



This transaction will then be recorded in the system. This serves as record purpose only and no transfer of money will take place until this transaction is settled.

This page corresponds to the following user requirements:

- **As a user, I want to create transactions easily.**
 - The application should allow users to add transaction records to any group they belong to, specifying what the transaction is for and the amount of the transaction.
 - The application should allow users to specify who paid for the transaction, and who has used the item of the transaction record.
 - The application should record the time and date which the transaction is created.

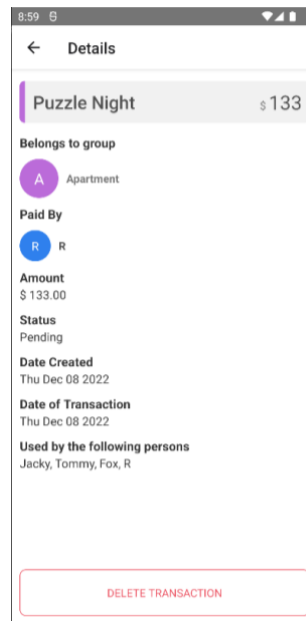
View Transaction

Type: **Page**

View transaction is a page which allows the user to view details of a recorded transaction.

All parameters recorded in the create transaction page will be displayed. In addition, the user will also have the option to delete the transaction.

This page can also be accessed by clicking on the transaction box in any page.



This page corresponds to the following user requirements:

- **As a user, I want to view the transaction so that I know how much I own/owe.**
 - For each transaction, the application should display the detail of the transaction.

2.1.2 Grouping and Group Members

Group

Type: **Object (Data)**

In the application, users can create and join groups. A group is a place where the transactions will belong to. Users can create and join group by generating invitation codes, which will be elaborated in the later sections.

In our example, a group named “Roommates” would be created. Peter and his roommates, Bob and Alex will be member of the group. The transaction “**Pizza**” will reside in this group. All three people can add transactions to the group, which will be visible to all three people.

Transactions must belong to one and exactly one group.

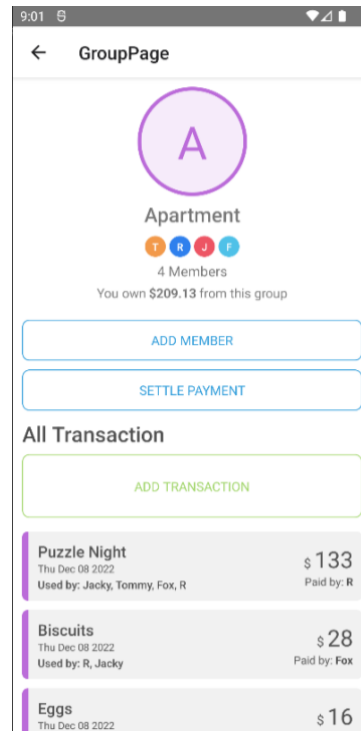
Group Page

Type: **Page**

Group page is a page where users can add transactions to a group.

Users can also review the transactions within a group and see how much this group owes him/ how much he owes this group.

It will also act as a home page for users to add members (See **Join Group**) and settle transactions (See **Settle Transactions**). User are also shown icons representing users within the group.



This page corresponds to the following user cases:

- **As a user, I want to create and form groups.**
 - The application should allow users to create groups, where the transactions are managed.
 - The application should allow each user to view and manage the group with their own devices.
 - For example, if a group of 3 people is created, any of the 3 users can modify the transaction record at any time with from their individual mobile phones. Changes made should be synchronized.

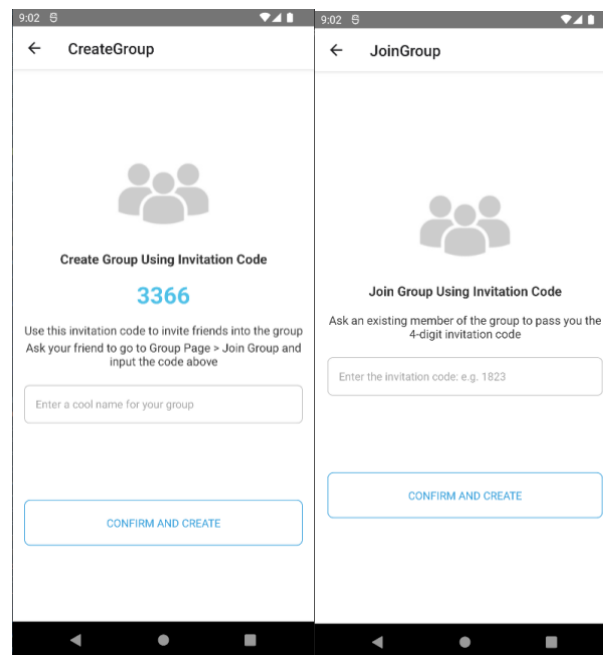
Join Group

Type: **Page**

Join Group/ Invite member is a collection of pages which would allow users to join new groups.

Group joining is on an invitation basis. In order for a new user to join a group, any existing members can use the “Add member” function to generate an invitation code. When the joining member enter this invitation code on the “Join Group” page, he will become a member of the group.

Invitation code is the only way for a user to join a group. Users cannot search up groups and enter unilaterally.



2.1.3 Transaction Settling

Settle Transaction

The settle transaction is a page which allows users to “Settle” outstanding transaction.

Up until this point, transactions are only recorded. If, at any point in time, users would like to “Settle” the transaction within the group and find out how much money he will collect/ pay the other members of the group, the settle transaction function will be used.

This page allows the user to select which transaction to settle. Afterwards, the application would sum up the transactions and calculate how much money each person should pay or receive (i.e., how much a certain person owns/owes after summing up all selected transactions). **If a person owns and owes money simultaneously, the system will calculate the net balance and display in total how much the person would pay or receive.**

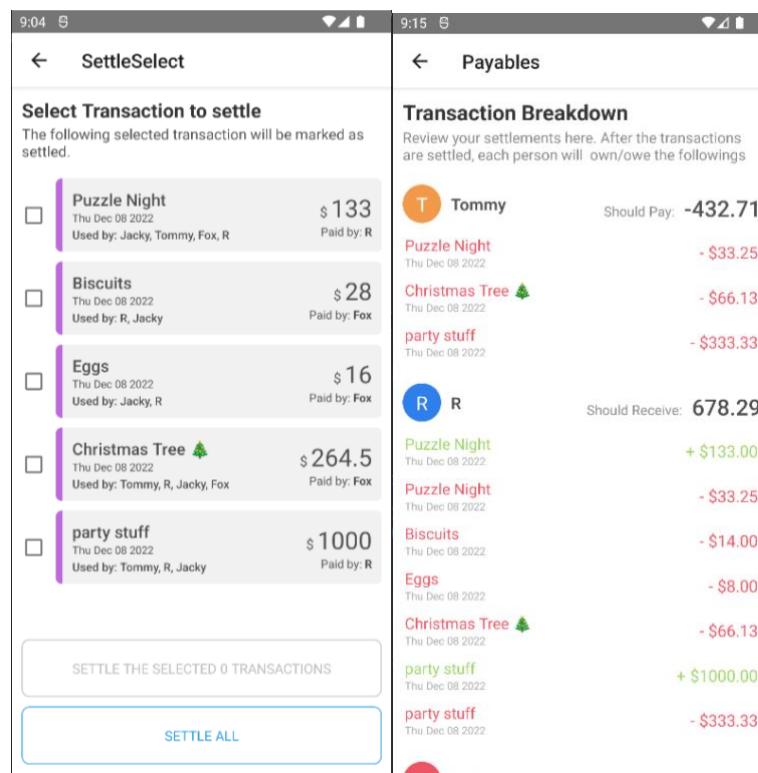
Our application will also display a transaction breakdown of that calculation so that each person would be able to see why he owns/owes a particular amount of money.

After the transaction is settled, it will still be retained in our system but will be marked “Settled”.

It is important to note that this application will inform the users how much money they should pay but **will not** have the functionality to let users transfer the money. It is up to the users to use applications such as PayMe to settle the debts themselves.

The settle transaction page corresponds to the following use cases:

- **As a user, at the end of a period, I want to settle selected transactions, and learn that who owes/owns how much money.**
 - The application should allow users to settle selected transactions.
 - The application should tell the users who has owned/owed how much money.
 - The application should give a breakdown explaining who a particular user would own/owe that amount of money.
 - The application will **not** offer any electronic wallet functionality and will not allow users to pay directly in the application. Instead, the application will provide information for the users to pay each other themselves.



How to pay

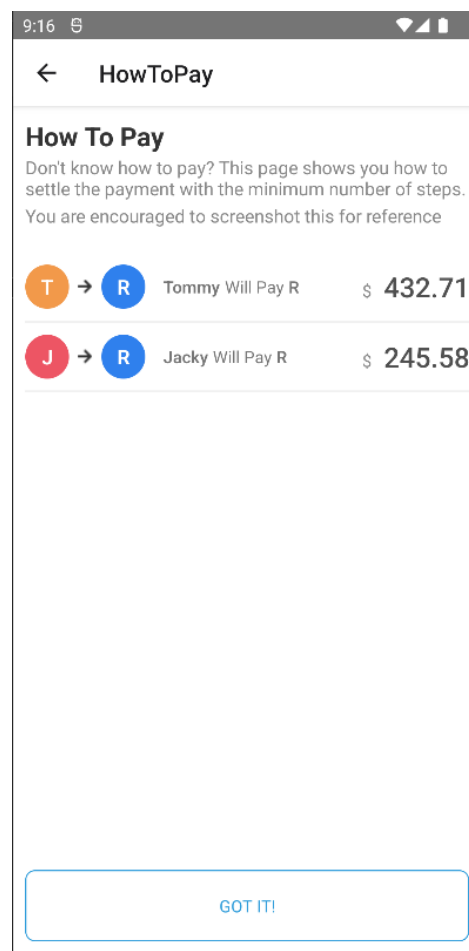
Type: **Page**

“**How to pay**” is a page which can be optionally displayed after a set of transaction is settled.

Refer to the previous screenshot in transaction breakdown. Information about how much each member owns/owe is displayed. This information is useful for knowing how much a certain person should receive/pay others. Yet, an unanswered question is who should pay who. In our previous example, it is intuitive, and Bob and Alex should pay Peter and the transaction will be settled. However, in the situation where more members are involved, the solution as not as obvious.

“How to settle” page aims to solve this problem by providing a simple graph to answer the who-should-pay-who problem. The graph matches the following criteria:

- Any person owes money would only have to pay the minimum amount of people.
- Any person who owns money would not need to pay anyone; he would just receive money.
- The amount which anyone would pay/receive matches directly with that displayed in the settle transaction page.

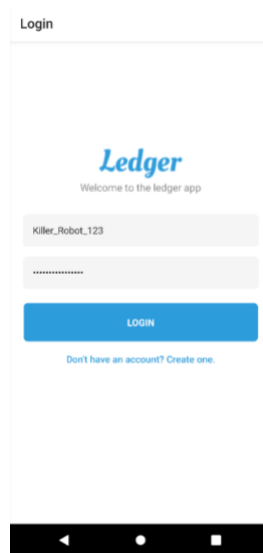


2.1.4 Login and Credentials

Login

The login page is the first page the user would see when he enters our application.

As most data are not stored in the local device, login credentials are necessary for our database to return the correct information. (See **implementation methodology**). After the user enter his login credentials, information would be returned properly.



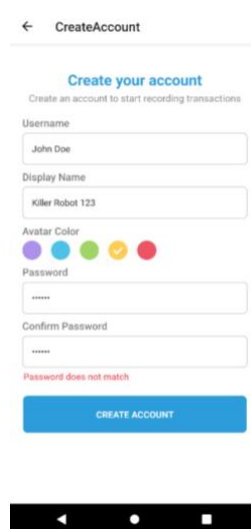
The login page corresponds to the following non-functional requirements:

- The application should implement adequate security measure such that only people who are properly invited into a group can add, remove, and modify transaction there.

Create Account

Create account is the page which the user will see if he would like to use our application but does not have an account. After the account creation, the user will be able to use the application immediately.

Upon account creation, a person object will be created.



Person

Type: Object (Data)

A “Person” is an abstraction of all data about a user which will be stored in the application.

The person object contains the following attributes:

- **Username:** The name which the user will use to login, it is also the name which our system identifies him.
- **Display Name:** The name which other people belonging to the same group will see.
- **Avatar Color:** A color for easy identification.

2.2 Implementation Methodology

2.2.1 Implementation Methodology Overview

This section highlights some of the technical implementation features used in our project. We will analyse how these implementation methods eased our development process, as well as helping us achieve greater results.

Snippets of codes will be attached to aid elaboration. For user interface examples, see session section 2.1.

2.2.2 React Native

The library **React Native** is used for the implementation of the project. All frontend components are rendered using react native. React native is a JavaScript library which rendered JS React code into native components.

The decision to choose React Native was made based on our team's familiarity with web development, as well as considering cross-platform capabilities for our app.

Our app was based strongly on the react-native typescript wireframe which is taken from the official website. Reference is attached in the latter part of this report.

2.2.3 Static Typing

JavaScript by itself is loosely type and type is inferred automatically by the interpreter. Our team realize this in the very early stages of the project, as well as it's negative implication for scalability. Hence, a decision was made to integrate type script into our application in order to achieve static typing.

Most data in the frontend are implemented to be strongly typed.

```
//Data contained in one transaction
export type Transaction = {
  amount: number;           //Amount of the transaction
  name: string;
  personPaid: Person;
  personUsed: Person[];
  timeCreated: Date;       //The time which this transaction record is created: Recorded automically by the app
  timeOfTransaction?: Date; //The time which this transaction took place: Input by user
  timeSettled?: Date;      //Will have undefined if not yet settled
  id: any;
  settled: boolean;
  group: string;           //Not the group data type, just an indication of what group it belongs to
  groupID?: any;          //UUID of the parent group
  color?: string;
};
```

The image above shows a notable example of our implementation of data representation of *a single transaction* in the frontend. It can be seen that each value of the object is typed with and attached with detailed documentation.

This minimized the time our team spent on de-bugging as mistakes are often caught before runtime. It additionally leverages some auto-complete features of the IDE as types can now be inferred. Furthermore, it enhanced and ease the communication process between frontend and backend developers as we can simultaneously work towards a common, agreed upon datatype.

2.2.4 Reusable Components

One benefit of using react and react native is the possibility of using reusable components. The frontend implementation of our app strongly leverages the power of reusable components.

In particular, a component is usually constructed as a **react component** (We use both class and functional components). Our group has created large amount of custom component which takes in various parameters (i.e., props). This allows our custom components to be as versatile as possible and be rendered in different context.

Within our app, transaction boxes, group icons, buttons and even texts are all custom-made components. Here, we use the “group icon” (round icon) to illustrate our design principle.

```
import { View, Text, StyleSheet, TextStyle } from 'react-native';
import myColors from "../../colors.json"

export interface IAppProps {
  iconText: string;           //Name to be displayed
  name?: string;             //Group Name
  color?: string;
  borderColor?: string;
  text?: string;             //Optional Text at the bottom of the icon
  textColor?: string;
  captionColor?: string;
  size?: number;
  useMargins?: boolean;     //Whether the group icon has margin
  iconTextSize?: number;
  iconFontWeight?: any;
}

export default function GroupIcon(props: IAppProps) {

  let styles = StyleSheet.create({
    main: {
      display: 'flex',
      flexDirection: 'column',

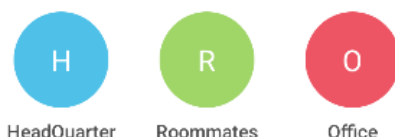
```

The picture above shows an example of a component which takes in large number of parameters to be rendered in different context. For instance, if we would like to render the component as a group icon, the following code is used.

Code:

```
<GroupIcon
  iconText="+ "
  name="Join Group"
  color={myColors.palette.leaf + "11"}
  textColor={myColors.palette.leaf}
  captionColor={myColors.palette.leaf}
  useMargins={true}
  iconTextSize={35}
  iconFontWeight="300"
/>
```

Result:



If we would like to render a person tag name:

Code:

```
<GroupIcon
  iconText={name}
  size={40}
  color={color}
/>
```

Result:

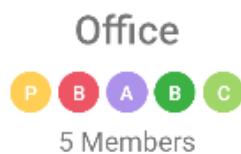


If we would like to render a small person icon in the group page:

Code:

```
this.state.group.members.map((p) => {
  return (
    <View style = {{marginLeft: 2, marginRight: 2, marginBottom: 5}}>
      <GroupIcon
        key={p.id}
        iconText={p.name}
        size={23}
        color={p.color}
        iconTextSize={11}
        iconFontWeight="600"
      />
    </View>
  )
})
```

Result:



Note that in all previous examples, only a small amount of code is needed to be modified to achieve very different results. The key idea is that when a component is created, we are essentially making an application programming interface for further development to use. Similar methodologies being applied to other components and used consistently throughout our project massively aided development and reduced development time.

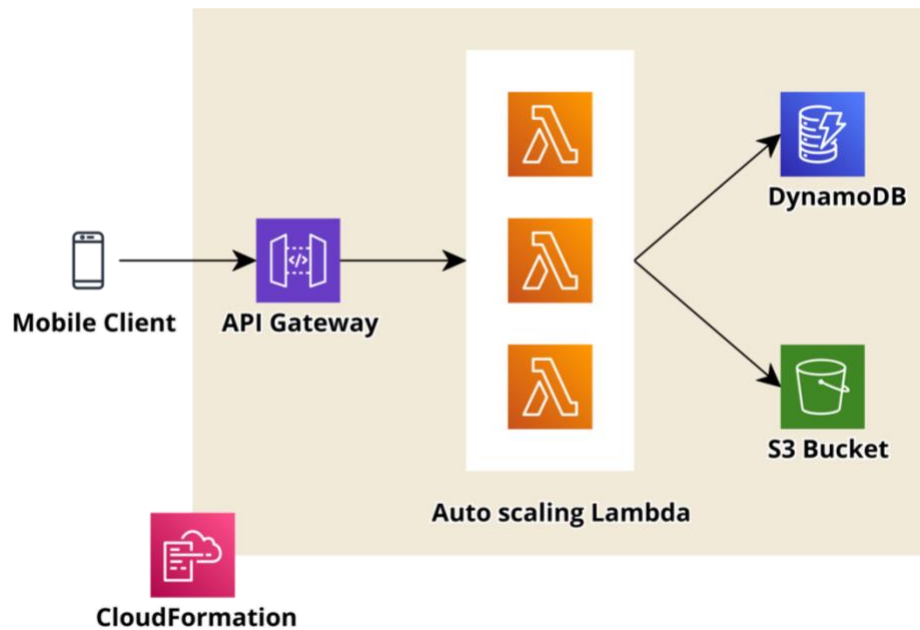
2.2.5 Navigation

By default, react supports single page applications. All navigations within our project are creates with the library react-navigation.

We used a function of the library to create a “Stack navigator”. Whenever the user clicks into a page, the new screen is added onto the stack, overlaying the existing screen. Users can also select to “Go Back”, (or use natively supported go back gesture) to navigate back to the previous screen. This is implemented with the “Go back” function of the stack navigator library.

This is adopted as stack navigation is used commonly in mobile apps, and therefore the experience would be most easy to understand by the user.

Infrastructure Overview:



Infrastructure Details:

- **API Gateway:** Hosts REST Api endpoints, direct the request to the corresponding Lambda functions for processing. It will also authenticate the incoming traffic. Please see appendix for the available endpoints.
- **Lambda:** Processes the incoming requests, perform data validation, perform read/write operations on DynamoDB or S3 Bucket then return the results as Api responses. The Lambdas will automatically scale-up to handle high traffic.
- **DynamoDB:** NoSQL database, stores the textual data, such as Users, Groups and Transactions. Please see appendix for the database schematics.
- **S3 Bucket:** Object storage service, stores the image data, such as Group icons and User icons.
- **CloudFormation:** AWS IaC agent, interprets the infrastructure codes and automatically provisions the above resources.

2.2.8 Data Security, Authentication and Authorization

As a financial application, we consider security as our top priority. We strived to ensure all data stored at the backend cannot be maliciously retrieved or manipulated through the following measures:

1. HTTPS connections

All backend endpoints are encrypted (HTTPS) with Amazon API Gateway certificate, this prevents Man-In-The-Middle attack or eavesdropping, especially when transporting sensitive data like login credentials and API keys.

2. Hashed password + Salt

All passwords are encrypted as hashed values on the database, **never as plain text**. It is calculated with SHA-512 with many iterations, with a unique salt for each password to prevent Rainbow Table attacks. Under these protections, users' original password will never be known to intruders or even system administrators like our team.

hash	salt
e6MQIXq4E...	s7uhNS6pe...
1TIs5OsRcf...	7XhLlcC1fF...

3. User access right

There will be multiple groups for multiple users, one should NOT be allowed to retrieve and modify transactions in groups they do not belong to. Under normal uses in the Ledger application, there would not be this issue as frontend codes will validate the request parameters before sending. But in case someone tries to directly interact with our API endpoints, our endpoints will verify whether the invoker has the access right to perform that action by checking the provided user ID and API key.

For example, an intruder wants to add a \$99999 transaction to a group he is not invited into, creating massive debt for the group members via the "/create_transaction" endpoint. The endpoint will spot this attempt and return a 403 unauthorized error.

```
Example Response 403 Forbidden
Body Header (1)
"User is not a member of this group!"
```

2.2.9 Database

AWS DynamoDB serves as the database for Ledger. It is NoSQL and supports on-demand capacity mode, which matches well with the serverless infrastructure design.

There are in total 4 tables, Credential, Person, Group and Transaction. By using IaC, they can be configured with just a few lines of codes, take the Transaction table as an example:

```
32
33 // Create the Transaction Table
34 const transactionTable = new Table(stack, 'Transaction', {
35   fields: {
36     groupId: 'string',
37     timeCreated: 'string',
38     id: 'string',
39   },
40   primaryIndex: {partitionKey: 'groupId', sortKey: 'timeCreated'},
41 });
42 transactionTable.addGlobalIndexes({
43   id_index: {
44     partitionKey: 'id',
45   },
46 });
47
```

Some tables are configured with secondary indices, which helped make queries more efficient. Take the above Transaction table as an example, the primary indices are “groupId” and “timeCreated”, which are useful when trying to query all transactions belonging to the same group and sorting them by creation time. On the other hand, the secondary index “id” is useful when querying a particular transaction with its unique ID.

All database configurations are stored in **“backend/stacks/MyStacks.ts”** file.

2.2.10 Serverless functions

After an endpoint receives incoming traffic, the content is directed to serverless functions hosted on the cloud, the AWS Lambda. They are responsible for processing the data and decide the response of that endpoint invocation, such as rejecting the request for insufficient access right, accepting the request then retrieving the relevant data from DynamoDB and returning them as the response.

The Lambda functions are in NodeJS runtime and written in TypeScript, which matches the frontend language. It helps ensure all request and response data are of the same type and can be used without further processing.

As part of the IaC infrastructure, these serverless functions can interact smoothly with other services like API Gateway and DynamoDB.

All functions are stored in the **“backend/services/functions”** folder.

2.2.11 Race condition

In this context, race condition refers to the database operation instead of CPU process. It happens when 2 or more clients/instances modifies the same entry in a table simultaneously, leading to data loss.

Assume at time=0, database stores a list of users belonging to the group "School": ["Fox", "Tommy"]

At time=1, both Arnold and Jacky wish to join the group "School". They both invoke one "/join_group" endpoint to add their name to the list. So, there are currently 2 instances trying to modify the list.

A **poorly written** Lambda function would do the following:

- | | |
|--|--------------------------------------|
| 1. Copy the list from database to local | Database: ["Fox", "Tommy"] |
| 2. Add the name "Arnold" to the local list | Local: ["Fox", "Tommy", "Arnold"] |
| 3. Replace the database with local list | Database: ["Fox", "Tommy", "Arnold"] |

This ran into a race condition, since both Arnold's and Jacky's instances are invoked simultaneously, the list they copied from database are not the most up to date. After replacing the old list with the new ones, one of the either **data will be lost**.

Database = ["Fox", "Tommy", "Arnold"] OR ["Fox", "Tommy", "Jacky"], but it's supposed to be: ["Fox", "Tommy", "Arnold", "Jacky"] instead.

As stated in section 2.2.8, our team takes data integrity very seriously. We have designed all our lambda functions to **only modify data item in-place** and uses **ConsistentRead** parameters for data retrieval. This is a code snippet of a safe version of the above operation:

```
// Append the userId to the members field
UpdateExpression: 'SET #members = list_append(#members, :newMember)',
ExpressionAttributeNames: {
  '#members': 'members',
},
```

2.2.12 Internet Access (broadcast receiver)

Ledger **cannot** function without having access to the internet. Without the internet, it may not access the cloud database and retrieve the Groups and Transactions. Therefore, Ledger will verify if the user's device has connected to the internet with the help of the **netinfo** package.

This package will broadcast the internet connection status of user's device. Ledger will prompt the user to connect to the internet when the broadcast signal is "false"

3. Testing and Evaluation

Throughout the development, our team has employed several testing methods to evaluate the performance and integrity of Ledger.

3.1 Unit tests

Our application is made up of many tiny building blocks, if one of them fails, the application may crash and affect users' experience. We want to make sure they are all still intact after every major development progress, so we do not accidentally break a feature without noticing it.

Targeting major component and functions, we have written some test scripts to depict **how they should operate** in normal circumstances. For example, pressing the login button should send the information to “/login_user” endpoint, and the endpoint itself should check the information against the database and return user ID and API key if the user exists. These checklists were written as codes using the **Vitest** framework.

A test script for the “/get_user_transactions” endpoint looks like so:

```
describe('Test: GET /get_group_transactions', () => {
  describe('Case 1: groupId parameter not provided', () => {
    it('Expect: Return 422 error', async () => {
      expect(
        await fetch(
          'https://5a4b3xg69l.execute-api.ap-east-1.amazonaws.com/...',
        ).then(response => response.json()),
      ).toEqual('Parameter groupId not provided!');
    });
  });
  describe('Case 2: Group does not exist', () => {
    it('Expect: Return empty array', async () => {
      expect(
```

Each describe() bracket contains the textual description of the test scenario, such as not providing a required parameter to the endpoint. Each it() bracket describes the high-level expected output. Each expect() and toEqual() chain will run the given scenario and compare the output with the expected result. This repeats until all test cases are covered.

We wrote comprehensive test cases. Their coverages span from normal operations to irregular/illegal requests, to ensure that each unit would work as intended and can handle unexpected errors without breaking. For the above endpoint, the test script reaches near 2000 lines.

3.2 Integration tests

Now that each individual components are working as intended, we want to make sure they work well when **combined**.

As mentioned in section 2.2, using TypeScript for development ensured each component communicated with each other using the correct data type/interface. This has already reduced a lot of run-time errors for us as the IDE catches them all during coding, even before compiling. Essentially, we are **“testing” as we are coding**.

```
navigation={this.props.navigation}
me={this.state.me}    Type 'Person | undefined' is not assignable to type 'Person'. Type 'undefined' is not assignable to
getData={this.getData}
```

A significant advantage of using TypeScript in both front and back end is being able to perform type checking for the request and response data, using the same interface/type declaration file directly.

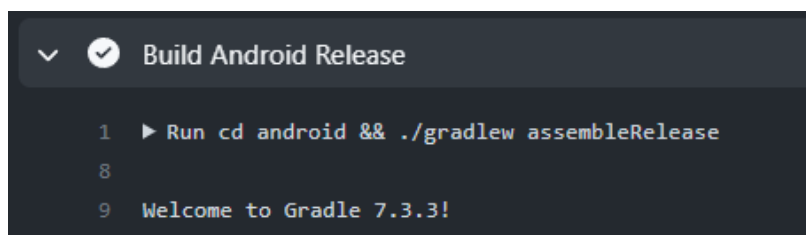
To demonstrate how TypeScript helped us to perform integration tests as we code, here is an interesting error caught by TypeScript:

The unit test indicates that both “Homescreen.tsx” (frontend) and “/get_user_transactions” (backend) are individually functional. But after integrating the two, the colours of the group icons were all wrong. Turns out the frontend and backend had two separate preferences for the spelling of the word “colour”, with and without the “u”. So, frontend tried to read the value of “color” from the object returned by the backend, which only has the value for “colour” instead. TypeScript caught this error, and the bug was quickly fixed.

3.3 User acceptance tests

After finished building the Ledger, we want to evaluate how well it perform in actual users’ hand. Is the UI clear and easy to use? Does it fit their needs? Are there any key features we forgot to develop? Those questions can only be answered by actual end users.

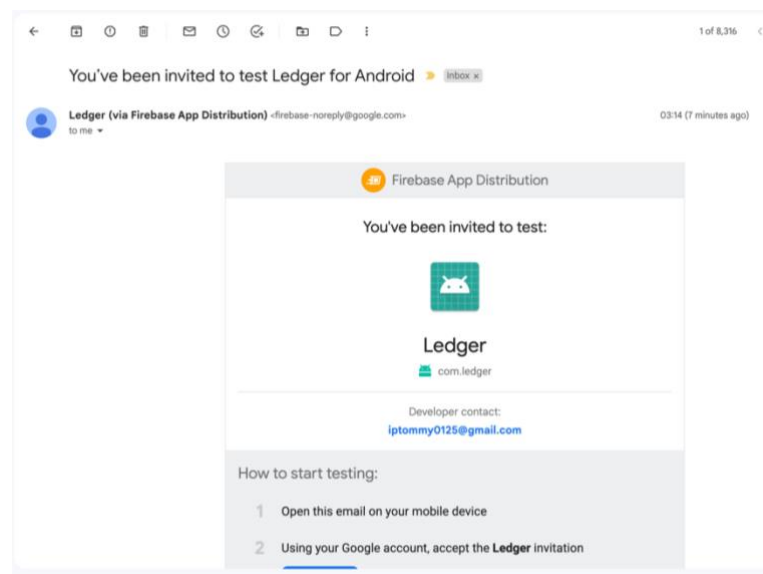
This project uses **Github Actions** to compile the source code into Android Package Kit (.apk files) and distribute them to testers/users. Whenever there are new pushes to the master branch, **Github Actions** will be triggered to automatically compile the Android app on ubuntu virtual machines and distribute the .apk file to **Firebase App Distribution**.



```
Build Android Release
1  ▶ Run cd android && ./gradlew assembleRelease
8
9  Welcome to Gradle 7.3.3!
```

```
12 i uploading binary...
13 ✓ uploaded update to existing release 1.0 (1) successfully!
14 i updating release notes...
15 ✓ added release notes successfully
16 i distributing to testers/groups...
17 ✓ distributed to testers/groups successfully
```

Testers/Users can subscribe to all the latest releases by submitting their email address to the registration system. Whenever there is a new build available, subscribed users will be notified by email along with links to download the latest version.



We have invited family and friends to test out Ledger throughout the development cycle. The following list of features/changes were made based on their valuable feedback.

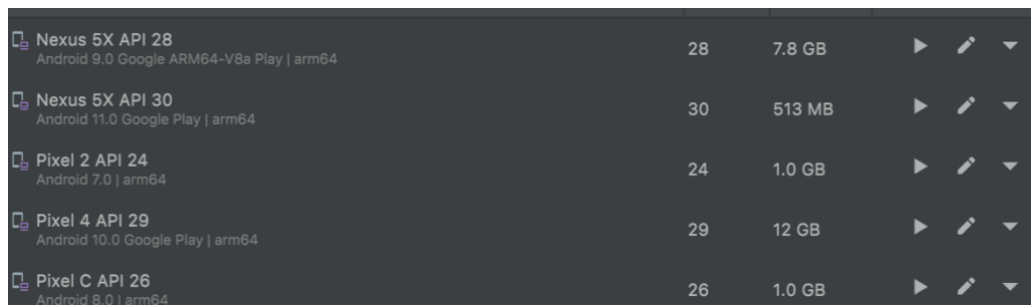
1. Added the option to settle all transactions
2. Showed all recent transactions related to the user across many groups
3. Sorted the group by alphabetic order

3.4 Device compatibility tests

Just because Ledger runs fine on our side, does not guarantee it will run smoothly on all other devices. We want to test how Ledger perform on devices with different screen ratios and Android versions.

We created several virtual android machines with Android Studio and ran the Ledger application on them, which **all operated well without notable issues**.

The testing devices have Android version from 7.0 to 12.0, each with varying screen ratios.



Device Name	API Level	RAM
Nexus 5X API 28	28	7.8 GB
Nexus 5X API 30	30	513 MB
Pixel 2 API 24	24	1.0 GB
Pixel 4 API 29	29	12 GB
Pixel C API 26	26	1.0 GB

According to Android Studio, this service range means our application should **support 92.7% of all existing Android devices.**



4. Conclusion

Our ledger recording mobile application is motivated by the fact that it is difficult to keep track of money within a large group of people.

It is our hope that our application can be used to ease such process, and therefore improve the quality of life of people living with a group. This can be particularly useful to college students living in a shared house or dormitory. The application can also be extended to people in offices, schools, or big families.

Ease-of-use, aesthetic user interface and application robustness lay the baseline of our application development. The tools we employ are chosen to achieve this result with the highest degree of ease.

Unfortunately, this development team do not have unlimited access to iOS devices, which means we could not develop an iOS version of the Ledger, despite being supported by React Native.

Still, we are happy to present you a fully-fledged Android application that can run flawlessly on most modern Android devices.

5. References

We explicitly acknowledge the following sources and libraries. The following technologies are employed directly to aid the development of our app.

- **React Native Typescript**
<https://reactnative.dev/docs/typescript>
- **ServerlessStack (SST)**
<https://sst.dev/>
- **CloudCraft**
<https://app.cloudcraft.co/view/f853a9b9-78bd-4045-be98-a79bdac6c084?key=7df9f270-f3f0-4c9f-a416-5264cde794e7>
- **react-native-community/netinfo**
<https://www.npmjs.com/package/@react-native-community/cli>
- **Git**
<https://git-scm.com/>
- **Vitest**
<https://vitest.dev/>
- **Github Actions**
<https://github.com/features/actions>
- **Firebase App Distribution**
<https://firebase.google.com/docs/app-distribution>
- **React Navigation**
To implement the navigation function of our app
Installed via `npm install @react-navigation/native`
<https://reactnavigation.org/>
- **UUID V4**
For the generation of random, unique id
Installed via `$ npm install uuidv4`
<https://www.npmjs.com/package/uuidv4>
- **React Native Date Picker**
Used in the new transaction page to select the transaction date
<https://www.npmjs.com/package/react-native-date-picker>
- **React Native Dialog**
For the generation of warning dialog box, where appropriate, in the new transaction page
<https://www.npmjs.com/package/react-native-dialog>

- **React Native Community Checkbox**
For generating “check boxes” in the settle transaction page
<https://www.npmjs.com/package/@react-native-community/checkbox>
- **Android Studio**
<https://developer.android.com/studio>

6. Appendix

- **Api Endpoints**
<https://documenter.getpostman.com/view/19999475/2s8YzRxhRk>
- **Database schematics**
<https://dbdiagram.io/d/635105334709410195a219e4>