

Informatics Large Practical

Coursework 2 - Coinz - Qais Patankar - s1620208
21st December 2018

Implementation

Firestore

Firestore is used to store data. You can see a model of the object structure in XML:

```
<users>
  <user key="uid">
    <name>FirstName LastName</name>
    <email>name@domain.com</email>
    <mapLastUpdate>timestamp</mapLastUpdate>

    <!-- Subcollection -->
    <map>
      <coin>...</coin>
      ...
    </map>

    <!-- Coins sent to this user -->
    <coinsIn>
      <coin>...</coin>
      ...
    </coinsIn>

    <!-- repeated for each currency -->
    <account-$CURRENCY>
      <coin>...</coin>
      ...
    </account-$CURRENCY>
  </user>
  ...
</users>
```

Note that this use of collections per “currency” leads to a limitation: you can only have a single account per currency.

The reason *email* is copied is that this information isn't guaranteed (i.e is Nullable). This will make project handover easier if future developers choose to use social accounts.

Coins and Accounts

Coins in an **Account** are commonly stored as a Set (there should ordinarily not be duplicate coins, and this enforces that constraint). However, other utilities (like DataManager) may accept a generic *Collection* to improve interoperability with Kotlin's filter/sorting methods (which only return array-like collections).#

Also, every Coin has a LatLng attribute. A future developer may want to save this memory space by removing it from picked up coins. Or it could be reused to provide a heatmap.

Geojson

The geojson is downloaded each day, converted to a list of Coins, and stored in the server. The server's list of coins is then used to show markers on the map (i.e, coins that can be picked up).

Picking coins up

Responding to location data is handled by the CoinFinder class (which is a bit of a misnomer, as it does not just deal with finding coins, it also deals with [bounds checking](#)).

Whenever a location is transmitted to a CoinFinder instance, all the coins within a 25 metre radius are collected into a set. That entire set is then removed via the DataManager. The reason a set is collected is to reduce the number of server calls.

DataManager (singleton class)

This singleton coordinates the use of Firestore and performs relevant updates to local copies of data. It makes data storage pluggable in the future (if we wanted to use a different data store) and makes data management simple throughout the application. It also performs optimisations and provides various data related utilities.

Functions defined by this singleton (such as *addFriend*, *deposit25*) are integral to the core functioning of the entire application. This singleton is also responsible for loading initial data from Firestore on "refresh" (a refresh is defined as a new login session or a new day starting) -- including loading of geojson.

This class does not have the glorious *green tick* as it leaves some extra functions around for future developers.

Backing fields

```
/**
 * Determines whether or not payments are enabled (with a backing field for optimisation)
 */
private var _coinsBankedToday: Int = 0
var coinsBankedToday: Int
    get() = _coinsBankedToday
    set(value) {
        // Don't do anything if it's already the correct value
        if (value == _coinsBankedToday) return

        Timber.d( message: "coinsBankedToday now set to %s", value)
        getUserDocument().update( field: "coinsBankedToday", value)
        _coinsBankedToday = value
    }
```

This allows a simple *coinsBankedToday* variable to be arbitrarily modified and persisted against the server. It is used internally but simplifies the rest of the code.

Reducing server reads

Some content (like lastMapUpdate) uses SharedPreferences as a *persistent* (across application sessions) caching layer (via [Prefs](#)). This can speed up startup time (the app doesn't need to check if the server day is different every time it is started, only once per day).

Prefs (singleton class)

This singleton coordinates the use of SharedPreferences and abstracts it all away into individual variables. This makes the use of SharedPreferences so much more simpler.

Batching of edits are also implemented, but were unused (not needed) for this project.

Example: firstTime

```
var firstTime: Boolean
    get() = prefs.getBoolean( key: "firstTime", defValue: true)
    set(value) = edit { it: SharedPreferences.Editor
        it.putBoolean("firstTime", value)
    }
```

This allows the boolean preference to be accessed and stored as a simple static variable, whereas in reality it uses SharedPreferences in the background.

Future improvements

Add a caching layer like in [DataManager](#), so that duplicate stores and reads are not needed.

Forward Compatibility

Some consideration has been taken for forward compatibility.

One such example would be, when creating Coin objects from the server provided geojson, we ignore invalid currencies under the assumption that more currencies may one day be added.

```
val currency = try {
    Currency.valueOf(feature.getStringProperty( key: "currency"))
} catch (_: IllegalArgumentException) {
    // App may be out of date if this is an unknown currency
    continue
}
```

Scalability

The system is not scalable as it loads all the coins in at once. The best way would be to have the coins stored in a database serverside and streamed to the user based on their current position.

If we decided to open the game area up we would need to revamp the entire system, especially considering that all coins are deleted when the day is refreshed, which is an expensive operation.

The aforementioned operation, however, is not expensive if performed inside a serverside “lambda” function.

The user interface is also not scalable for large numbers of coins. The specification may be altered to allow users to share arbitrary denominations of wallet currencies (i.e non GOLD coins). This would allow us to share a singular numeric monetary value, instead of a list of coins.

Features not completed

Most bonus features that were on my initial plan did not make it to this iteration of the game. Most bonus features of the current iteration of the game did not exist in the initial plan.

Features 2.1-2.5

The bonus features describe in the first piece of coursework were not created, and this is largely due to overestimation of tasks (create & share), an inability to secure data effectively (leaderboard) and some theming trickery (dark mode).

Feature 2.6 “Friends”

A primitive version of this feature was completed, however we are missing the ability to share your location & view a custom scoreboard.

Security

Security is mentioned in several places of my project plan, and the restrictions of this coursework (not being able to write server code) make it trivial to arbitrarily generate coins.

Nevertheless, I have created Firebase security rules in the hope that risk is reduced. Please read the below rules as it is explained in detail. Potential further expansion (in strictness) is also mentioned:

```
1  service cloud.firestore {
2    match /databases/{database}/documents {
3      match /users/{userID} {
4        allow write: if request.auth.uid == userID; // only this user can write to this user
5        allow read: if true; // anyone can read this user
6
7      match /coinsIn/{document=**} {
8        allow create: if true; // anyone can send coins to this user (only create)
9        allow read,write: if request.auth.uid == userID; // only this user can read/write coins sent
10
11        // New project developers may want to narrow this down to:
12        // - allow create: if request.auth.uid != userID;
13        // - allow read,delete: if request.auth.uid == userID;
14      }
15    }
16
17    match /{document=**} {
18      allow read,write: if request.auth.uid == userID; // this user can read/write any subcollectoin
19    }
20  }
21
22  match /{document=**} {
23    allow read, write: if false; // default that nobody has access to anything
24  }
25 }
26 }
```

Also, final note: there is no assertion for the positivity of a coin's value. This means that a user can generate their own (negatively valued) coin, and share this coin with any individual.

It may be useful for the team taking this project forward to add coin value positivity to the server code they will inevitably add.

Additional “bonus” features

Automatic pickup of coins

This application automatically picks up coins when you are within the required radius. It does not discriminate between the coins picked up - they all go into the relevant wallets.

View friend profile pictures

The application integrates with popular avatar service *Gravatar* to provide profile photos of the friends you add. This helps you easily spot your favourite friends for gifting.

The code for this is in `Friend.getGravatar` (original work):

```
// From: https://en.gravatar.com/site/implement/hash/ (inspiration)
// From: https://stackoverflow.com/questions/4846484/md5-hashing-in-android
fun getGravatar() : String {
    md5.let { it: MessageDigest! }
        .update(email.trim().toLowerCase().toByteArray())
        .return "https://www.gravatar.com/avatar/${Utils.toHexString(it.digest())}?s=128"
}
```

With support from `Utils.toHexString` (this utility is not original work):

```
// From: https://stackoverflow.com/questions/332079/in-java-how-do-i
fun toHexString(bytes: ByteArray): String {
    val hexString = StringBuilder()

    for (i in bytes.indices) {
        val hex = Integer.toHexString(i: 0xFF and bytes[i].toInt())
        if (hex.length == 1) {
            hexString.append('0')
        }
        hexString.append(hex)
    }

    return hexString.toString()
}
```

Smart deposits

The application exposes “bank largest” and “bank all” per wallet, and then “deposit up to 25”/“deposit all” for your entire bank. This does the relevant conversions of exchange rates and makes you benefit from picked up coins the most.

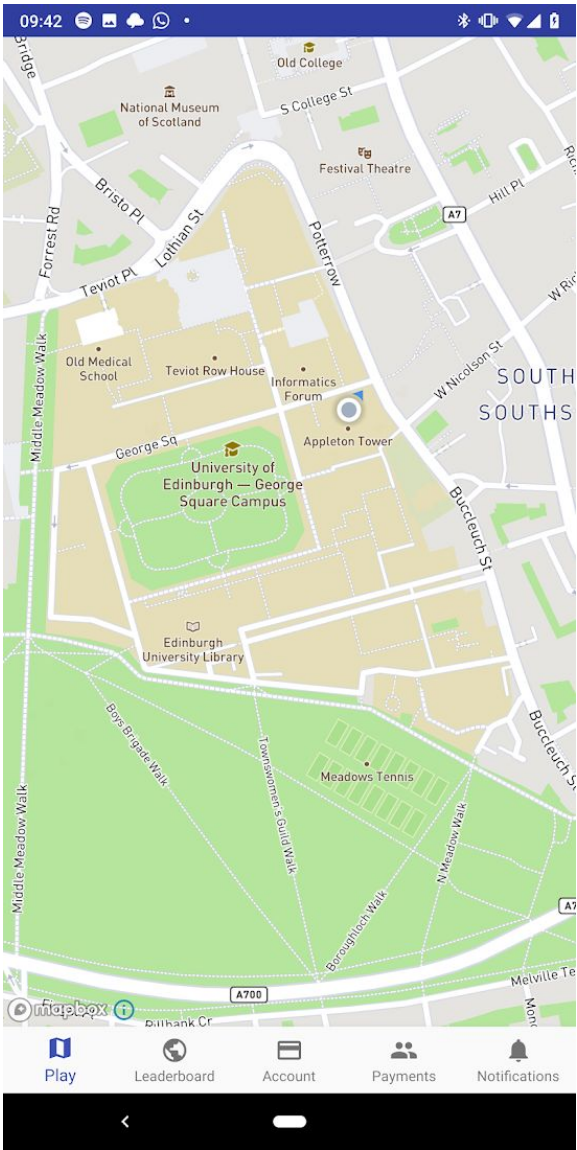
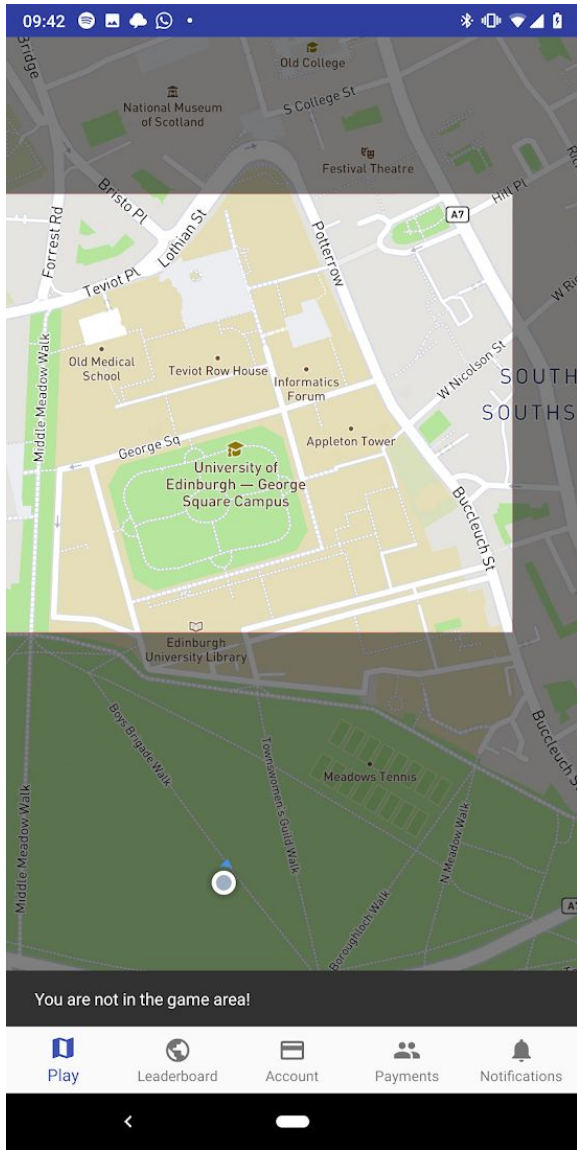
Welcome

When you first open the application, you will be greeted with a helpful splash screen giving you instructions on how to play the game. This is a quick non-interactive tutorial. See [Welcome Activity](#).

Game area indicator

The game tells you when you have left the game area by presenting a snackbar and greying out the rest of the world.

(Early stage screenshots, with no marker on screen, and non-currency BottomNavigationView)

Inside game area	Outside game area
	

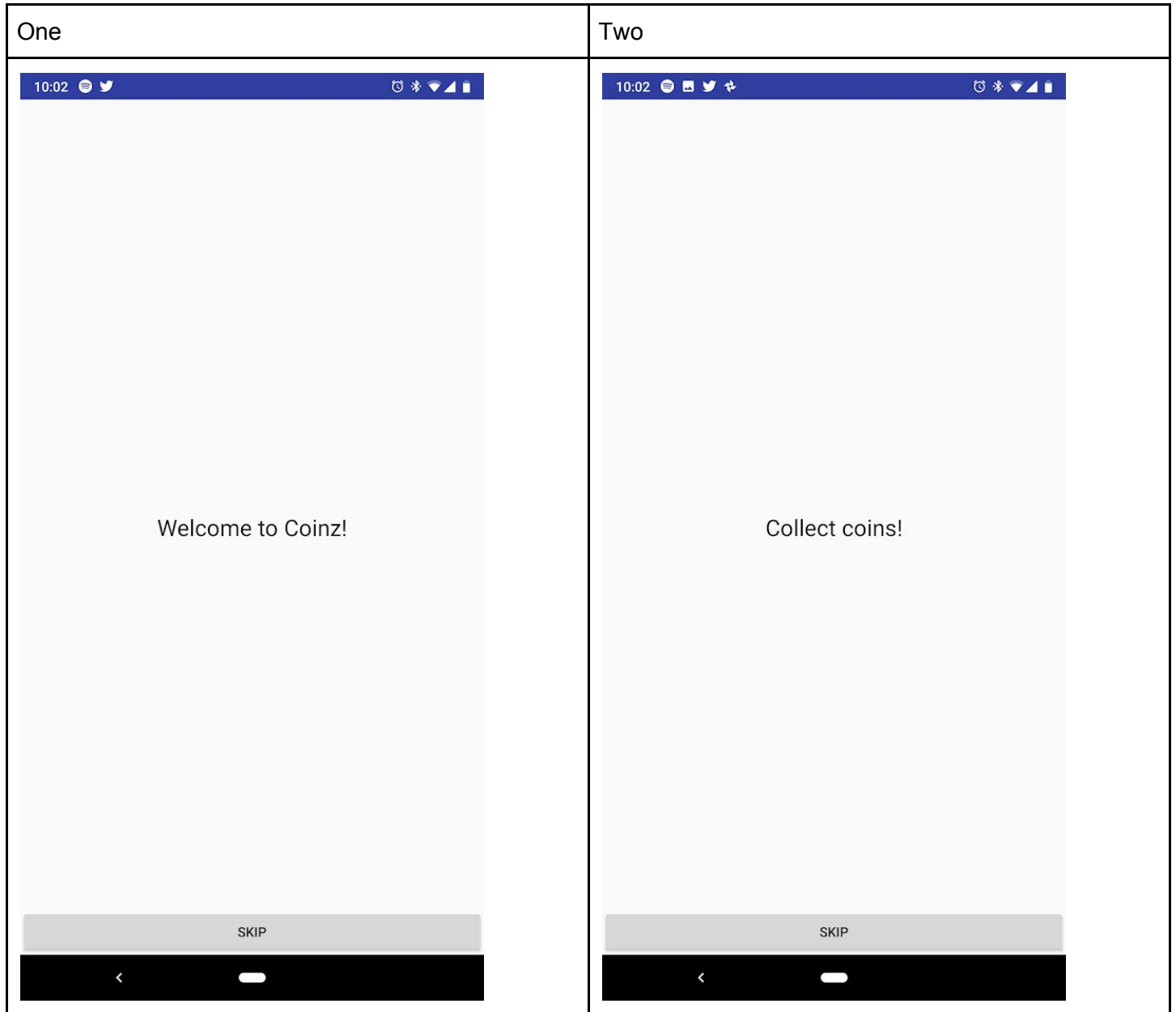
Application in use

Welcome Activity

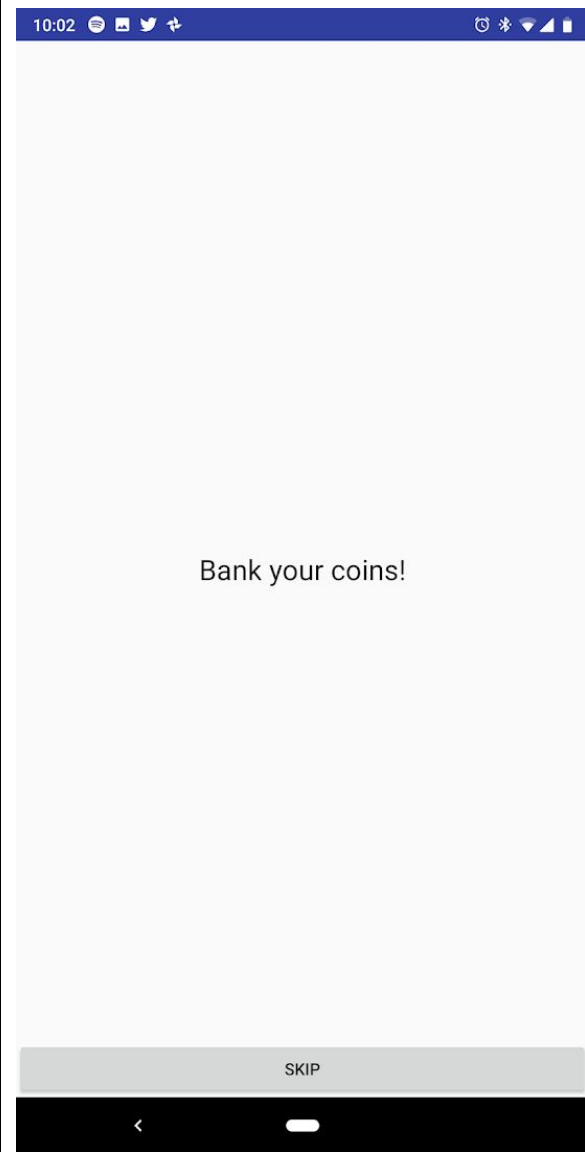
The first activity the user will ever see is the welcome activity. See [Welcome](#) above.

Currently these instructions are very primitive right now, but the idea is that once a development team takes over the project, they can put in some polish to produce educational graphics.

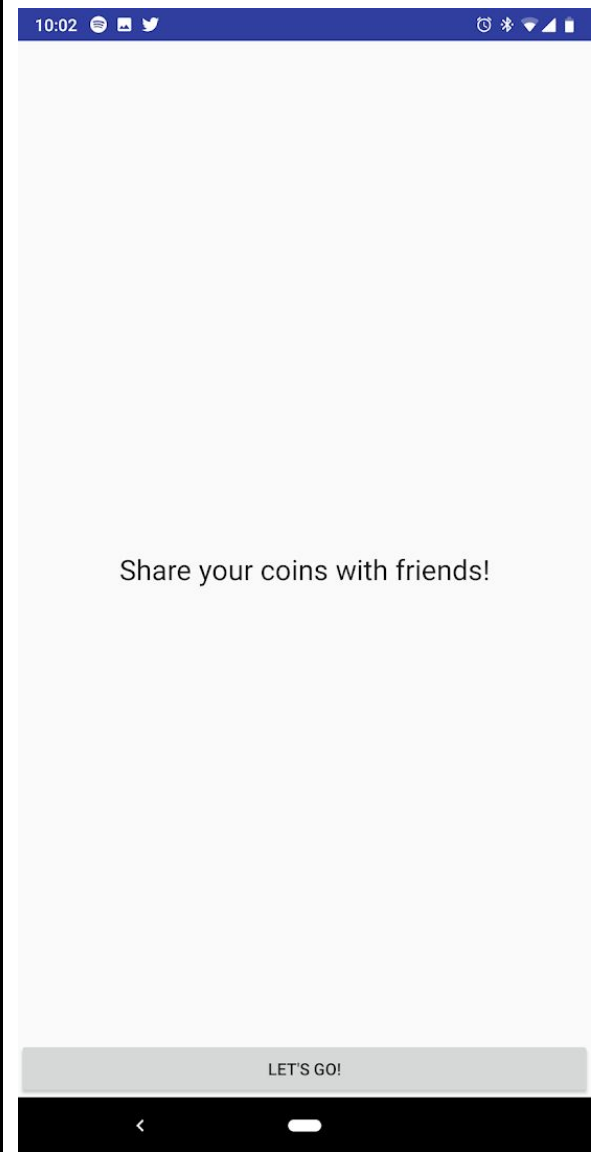
Note that the “Skip” button changes to “Let’s go!” on the final page. This uses a ViewPager.



Three

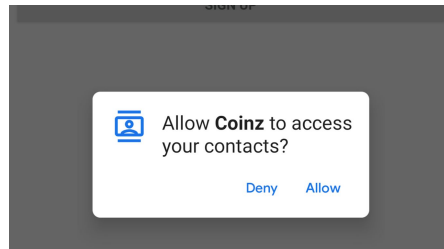


Four

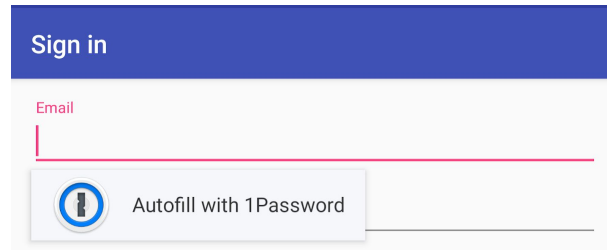


Auth

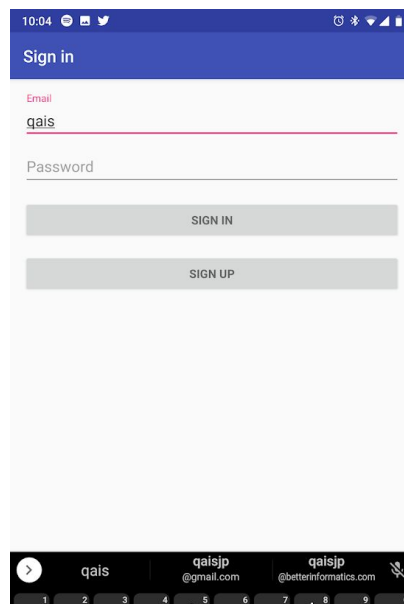
The login activity first asks for contacts access - not to steal your data, but to provide email hints.



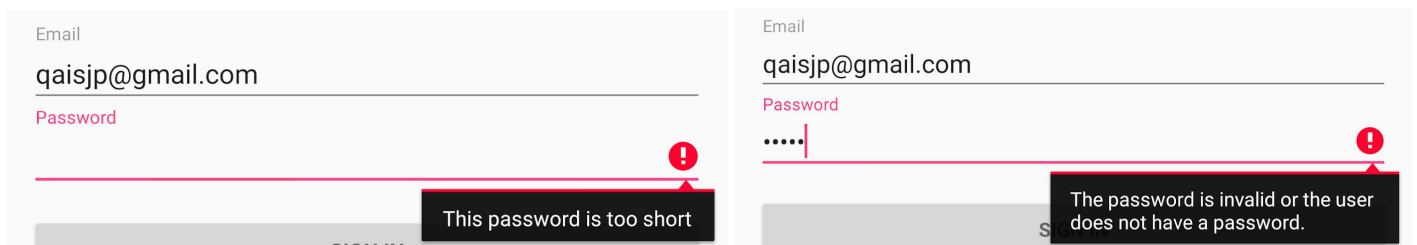
There is also autofill support:



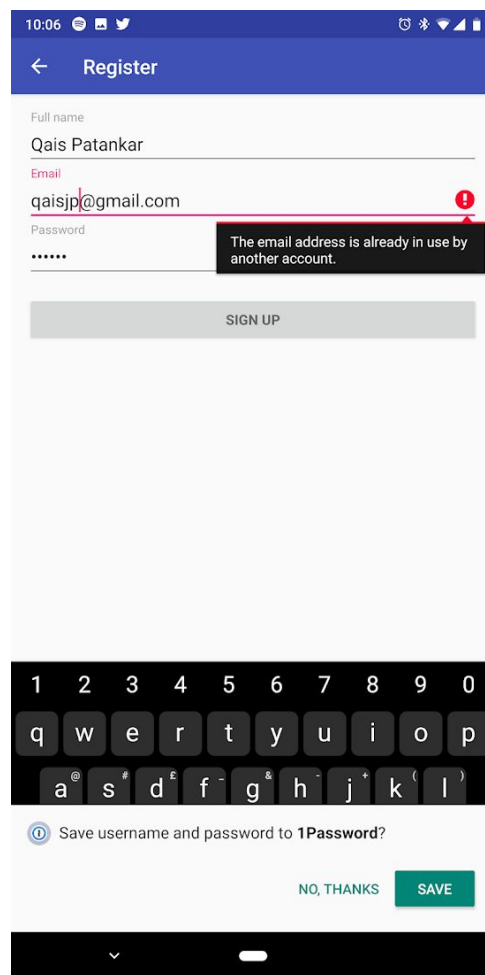
This is what the entire activity looks like:



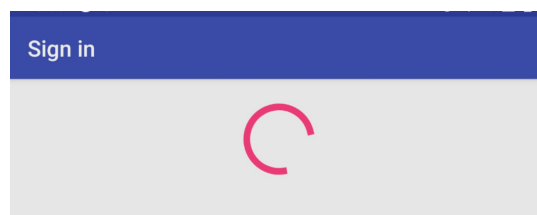
Error messages flag up the correct field:



Clicking the register button takes us to the register activity (with the back button set correctly), with email and password pre-filled (you need to fill in your name). Errors on this activity demonstrated below - also note that Android recognises this as a registration and prompts to save this in my password manager:

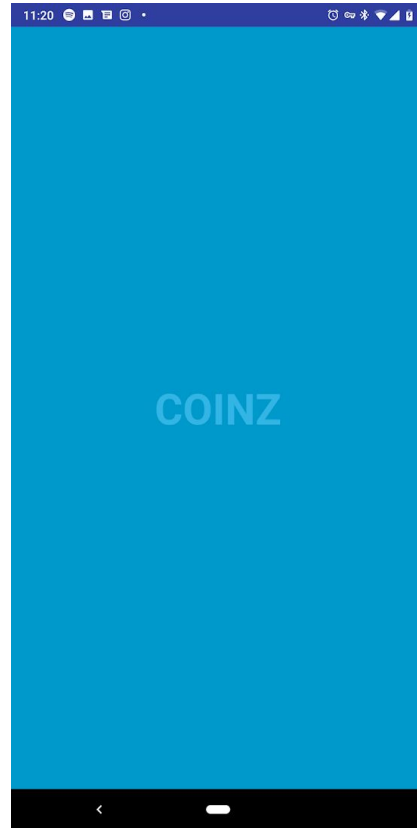


Finally, both sign in and sign up buttons show a progress indicator straight after pushing the button:



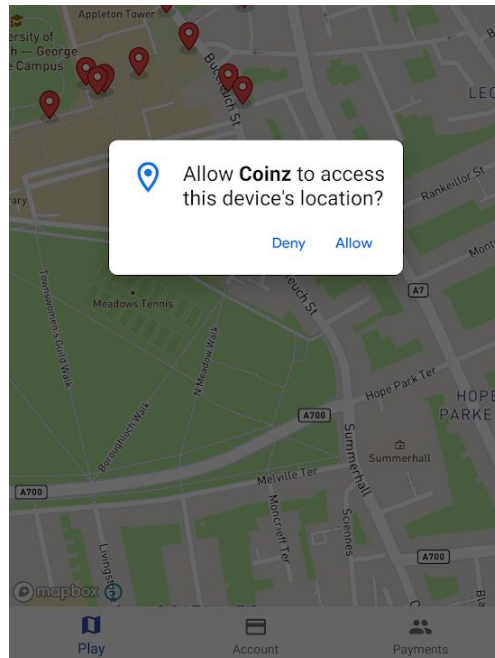
Splash

Straight after logging in you would be sent to PlayFragment (via GameActivity), but on subsequent application opens, when the app is checking your logged in status, the following splash screen is shown:

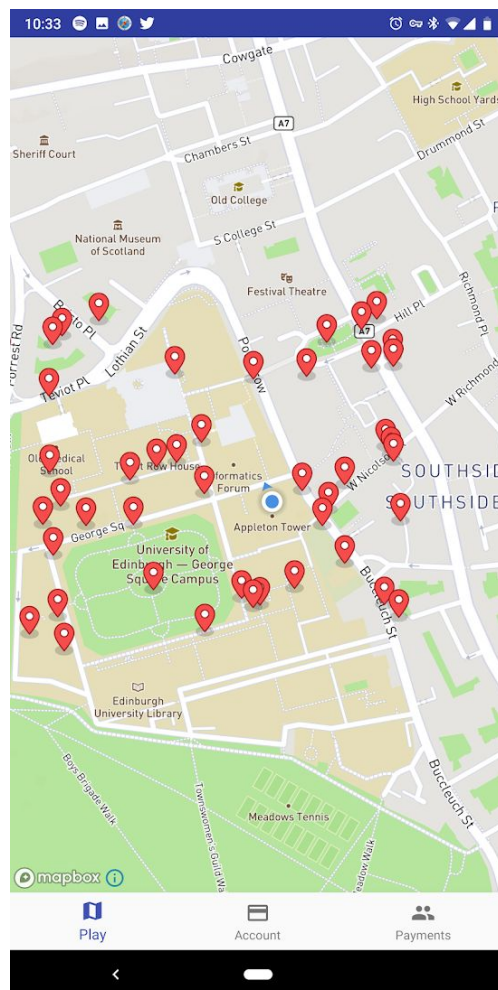


PlayFragment (via GameActivity)

Access to this fragment checks if you have the needed permissions, and prompts accordingly:



The user is on the map (with compass). Walking around will make coins disappear, and the coins picked up will be deposited into your wallets.



AccountFragment (via GameActivity)

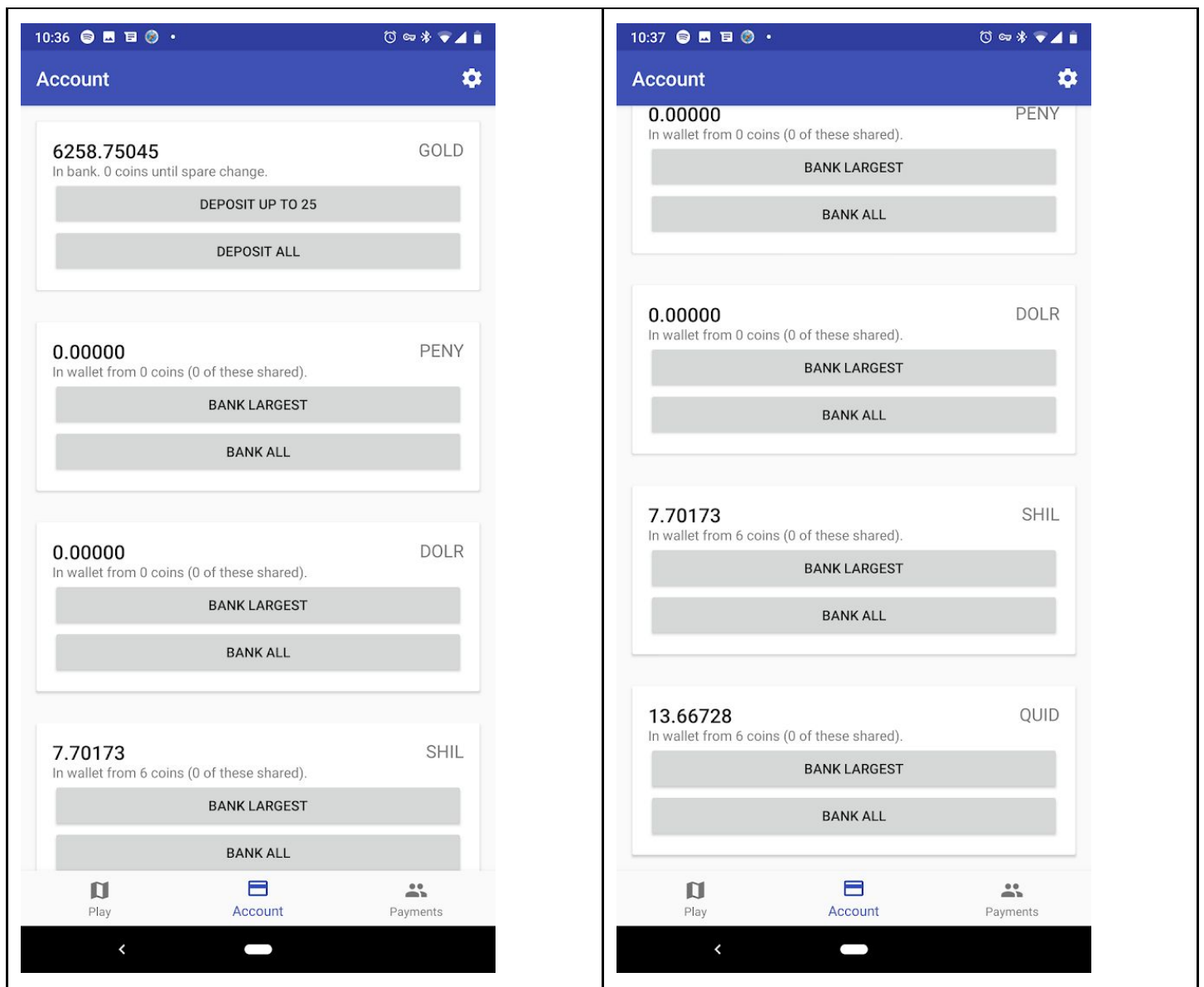
This is a RecyclerView using an AccountViewAdapter.

This fragment shows a list of your accounts (to the user this known as your “bank account” in GOLD, and numerous “wallets” in other currencies).

Actions:

- **Deposit up to 25:** bank coins from all wallets enough to reach the non-shared 25 threshold for the day. This takes into account exchange rates.
- **Deposit all:** bank coins from all wallets up to 25 threshold + all shared coins. This takes into account exchange rates.
- **Bank largest:** bank largest *bankable* coin from this wallet.
- **Bank all:** bank all coins in this wallet up to the non-shared threshold + all shared coins.

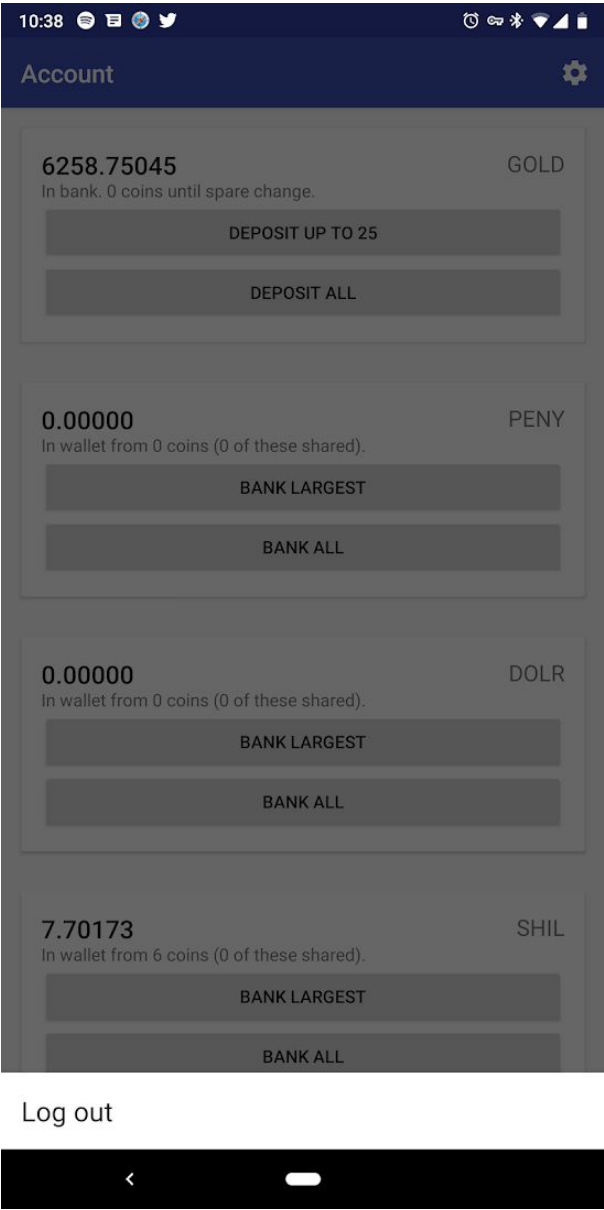
Note that a *bankable* coin is either: any coin if threshold not met, only the shared coins if the threshold has been met



SettingsDialogFragment (via AccountFragment)

This fragment is triggered from the gears button in the toolbar.

This has support for more buttons and is easily extendable.



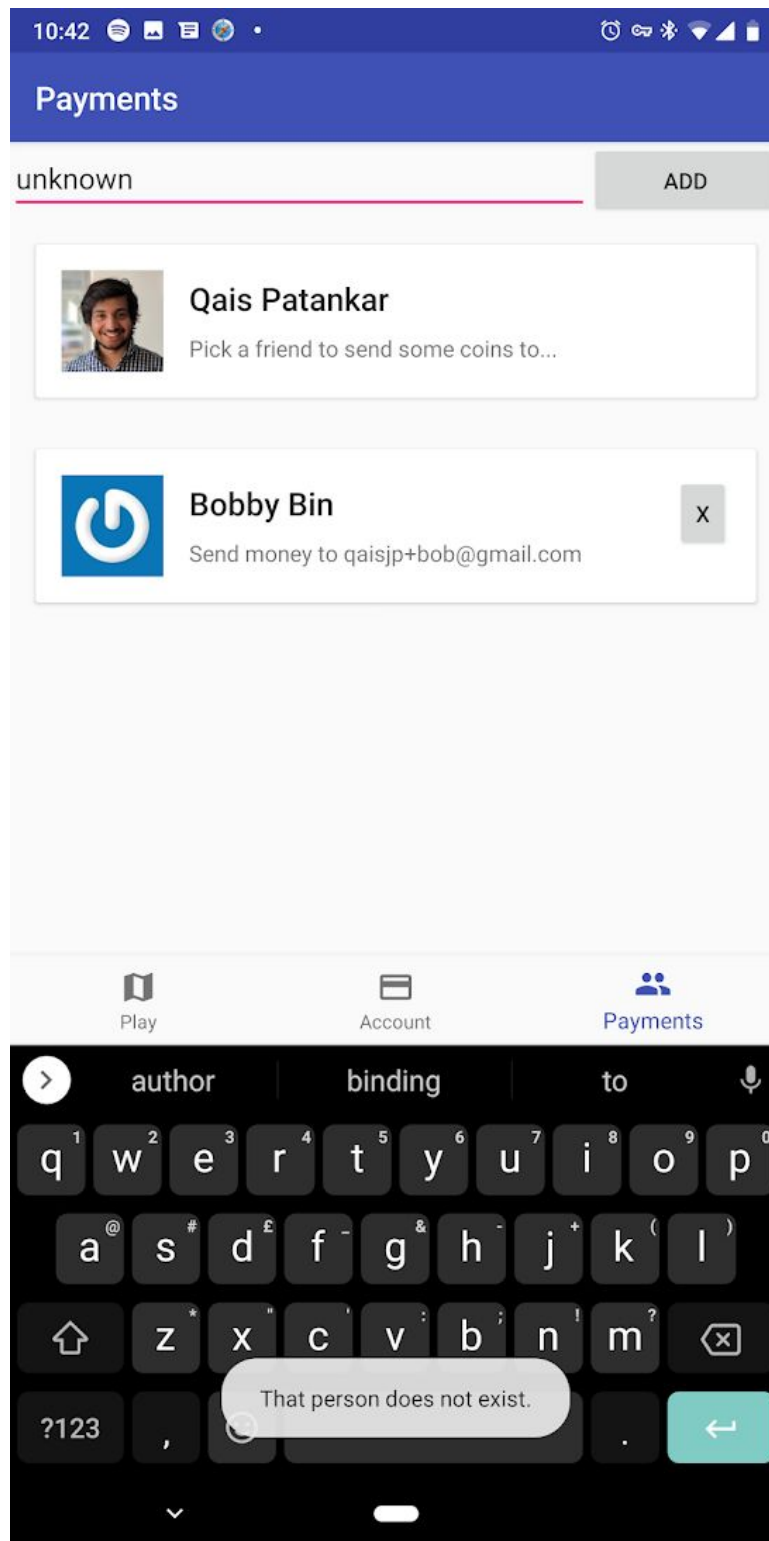
PaymentsFragment

This fragmentm, accessed from the bottom nav, allows you to:

- Add friends by email address
- Pay friends
- Unfriend friends (delete people from your friends list)

Friendships are not mutual.

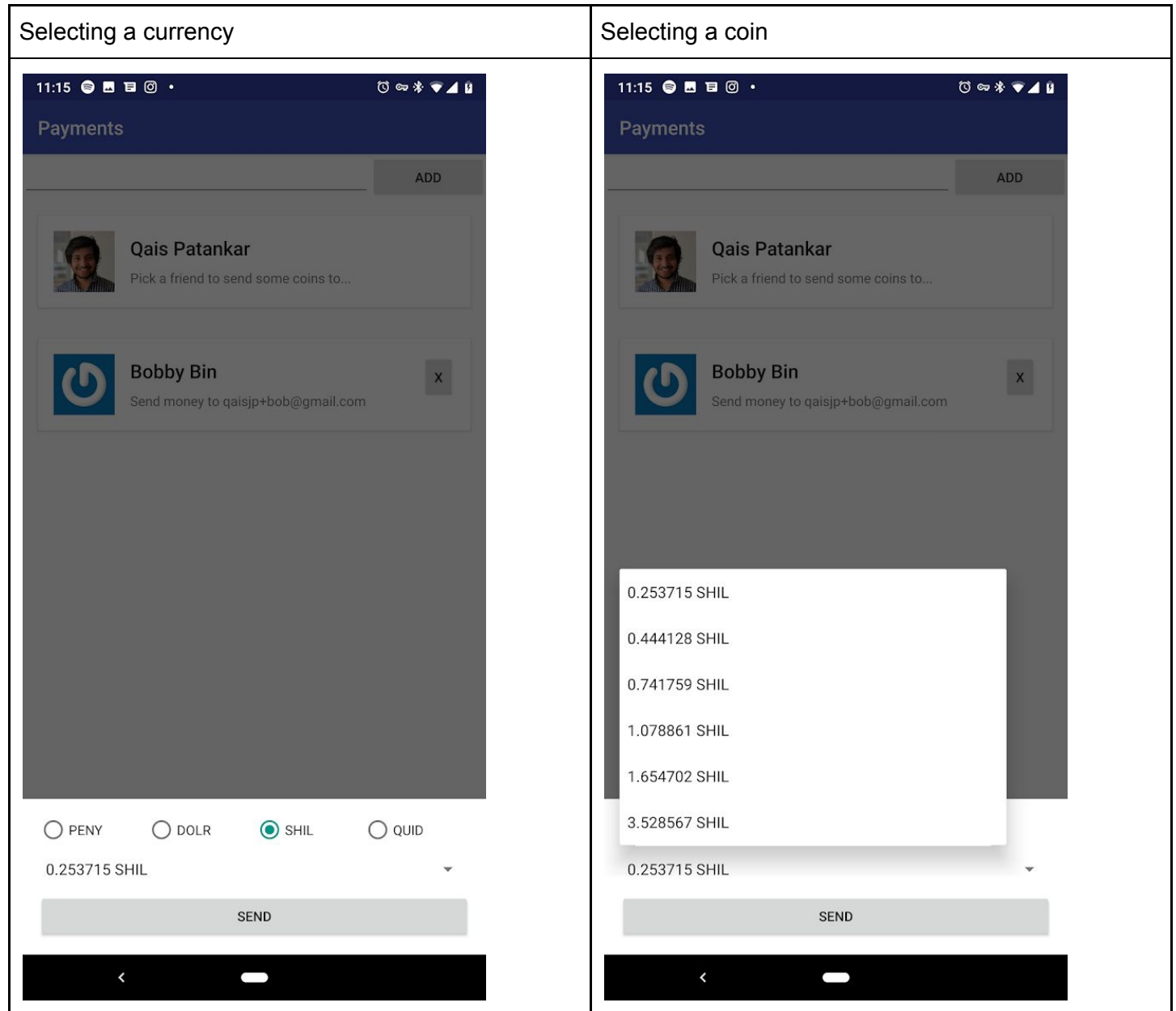
In the below screenshot, observe the error message “That person does not exist” when you try to add someone who does not exist.



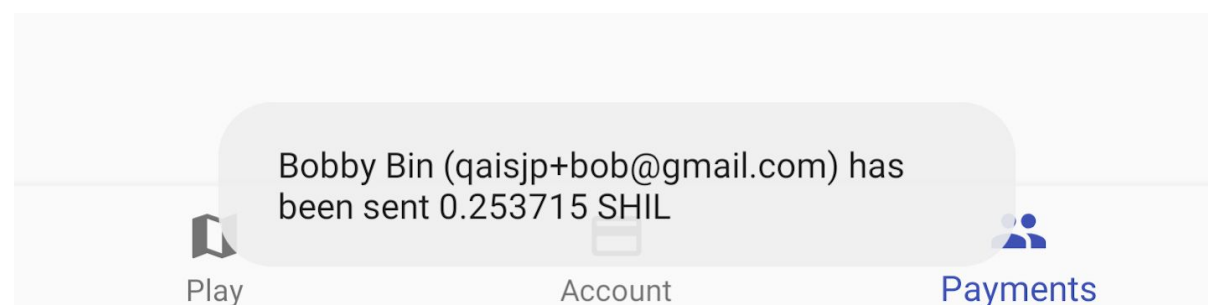
Making Payments

Clicking on a card will show a `PayFriendDialogFragment`, as shown below. You can select which wallet you want to take coins out of, and then pick specifically which coin you want to share.

The coins are ordered in ascending order by value.



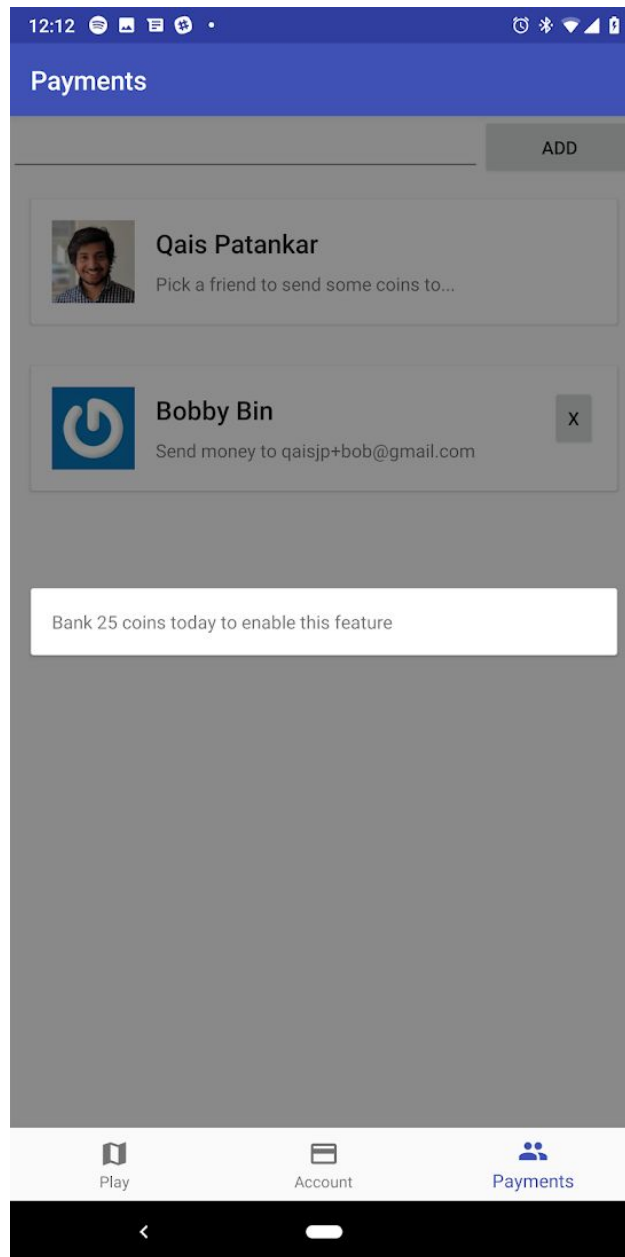
Toast after picking send:



If you haven't met the 25 coin threshold...

...then you cannot send coins to your friends. This has the unfortunate side effect of making it impossible to share coins that don't abide by the threshold (i.e, coins, in wallets, that are marked as shared).

This is what it looks like. You cannot click behind the "modal" (note that this is not a real modal):



Acknowledgements

Android Studio generated parts of the following classes:

- Login/Register activities
- Modal Bottom Sheet used in PayFriendDialogFragment and SettingsDialogFragment.
- WelcomeActivity's view pager
- SplashActivity (heavily cut down)
- *Potentially some other classes*

Gravatar hashing used the following documentation and urls:

- <https://en.gravatar.com/site/implement/hash/>
- <https://stackoverflow.com/questions/4846484/md5-hashing-in-android>
- <https://stackoverflow.com/questions/332079/in-java-how-do-i-convert-a-byte-array-to-a-string-of-hex-digits-while-keeping-l>

RegisterActivity's back button had help from this doc:

- <https://developer.android.com/training/implementing-navigation/ancestral>

Any fragments (at least AccountsFragment, PaymentsFragment) that make use of RecyclerView were assisted by the following doc:

- <https://developer.android.com/guide/topics/ui/layout/recyclerview>

Use of string plurals and string formatting had help from this doc:

- <https://developer.android.com/guide/topics/resources/string-resource>

BottomNavigationViewHelper is from the following doc:

- <https://stackoverflow.com/a/47407229/1517394>
- An attempt to move to API 28 was done to allow visibilityMode to be used. Unfortunately 28 is not supported by DICE's version of Android Studio, so I had to roll back.

Coin, Friend, Currency and other classes that implement *Parcelable* made use of Android Studio's "generate Parcelable" tool. All generated code here was edited to ensure efficiency and to ensure the code is best suited to Kotlin.

Use of Picasso in FriendsViewAdapter made use of the relevant docs:

- <https://square.github.io/picasso/>

Mapbox bootstrapping made use of *at least* the following docs:

- <https://www.mapbox.com/android-docs/maps/overview/location-component/>
- <https://www.mapbox.com/android-docs/maps/overview/data-driven-styling/>
- <https://www.mapbox.com/android-docs/maps/overview/camera/>