



# Coinz

---

## Implementation Report

Prepared for:  
Informatics Large Practical Coursework 2  
School of Informatics  
The University of Edinburgh

*Likhitha Sai Modalavalasa s1684168*

# Table of Contents

<b>Coinz</b>	<b>3</b>
Algorithms and Data Structures	3
Download of Geo-JSON maps	3
Starring Data Structures	4
Ambiguities	4
Parsing Maps and Adding markers	4
Starring Data Structures	4
Ambiguities	5
Storage of user's progress in playing the game	5
FireBase - UsersWallet	5
SharedPreferences	6
! Challengez overcame	6
Detection of the coinz	6
Starring Data Structures	7
Banking coinz	7
Starring Data Structures	8
Messaging Coinz	8
Starring Data Structures	8
Ambiguities	9
Leaderboard	9
Ambiguities	10
Health	10
Ambiguities	10
Parts of Design which have not been Implemented	10
Hard Mode	11
Additional Features Implemented	11
Share Feature (Bonus Feature)	11
Progress of User Feature (Bonus Feature)	11
Initial Design of the game	12
Final Design of the game	12
Did things go according to plan ?	13
ScreenShots of Coinz in Use	14
Create Account Screen	14
Login Screen	14
Landing Activity	15
Navigation Drawer	15
	2

Map Activity	16
Wallet Activity	17
Messaging Activity	18
Leaderboard Activity	18
Health Activity	19
Share Stats	19
Acknowledgements	20

# Coinz

*Coinz is a location-based game that allows the user to collect virtual coins that are scattered around the central area of the campus. It's a multiplayer game in regards to the fact that users will be able to exchange coins amongst each other. In addition to that, there's a competitive factor wherein user's gamer ability is assessed based on the gold coins in their central bank account. This will be further elucidated in the report.*

## Algorithms and Data Structures

### Download of Geo-JSON maps

I utilised Async tasks to download the maps in the a background thread separate to the main thread as taught in the lectures. Downloading remote data is not a task that is usually done on the UI thread.

- `doInBackground`: Performs a fetch for the data at the URL that is (<http://homepages.inf.ed.ac.uk/stg/coinz/>) and store the data in JSON string to later parse it. This task is performed by the `DownloadFeaturesTask`. The data is read using a `InputBuffer` and is concatenated to a string.
- `onPostExecute`: Sends the downloaded content through the `DownloadCompleteListener` Interface.
- The maps are updated according to the date. The `downloadDate` is sharedprefs and if the today's date is different from the existing date, the map is update else the JSON string is retrieved from the `SharedPreferences`(as the JSON string is stored in the prefs too)

```
if (downloadDate != prefs!!.lastDownloadDate) {  
    val url = "http://homepages.inf.ed.ac.uk/stg/coinz/$downloadDate/coinzmap.geojson"  
    mapfeat = DownloadFeaturesTask(DownloadCompleteRunner).execute(url).get()  
    prefs!!.mapfeat = mapfeat  
} else {  
    mapfeat = prefs!!.mapfeat  
}
```

### Starring Data Structures

- The result is then stored in a *String* format to make the JSON parsing easier. The parsing is further explained in the next algorithm.

### Ambiguities

- The download of the maps at midnight: For now the user will have to exit the maps activity (trigger `onStop`) for the download of maps at midnight but could resolve it with another asynchronous thread. For simplicity of implementation, I did not fully resolve this ambiguity.

## Parsing Maps and Adding markers

- Every GeoJSON file from the URL has the same format wherein there's a list of features which contain details about the markers.
- GeoJSON provides a FeatureCollection data structure which can be acquired from the JSON string result obtained from the JSON object. The features list is iterable so the addition of marker will be executed inside the features iterator loop.
- Below is the code:

```
val featureCollection: FeatureCollection = FeatureCollection.fromJson(mapfeat)
val features: List<Feature>? = featureCollection.features()

for (f in features!!.iterator()) {
    if (f.geometry() is Point) {
        // Insert markers
    }
}
```

- In the loop, obtain all features like colours and id needed to add the markers, using the addMarker function of mapbox.

## Starring Data Structures

- Coin data structure - Coin is a Kotlin data class that defines the Coin as an object. It consists of the id, value and currency.
- Markers - (HashMap<Marker, Coin>) - Marker are a key-value data structure wherein the key is the marker as the markers could later be removed by their key when collected by user and the value is the coin data structure that is used to store the coin when collected.

## Ambiguities

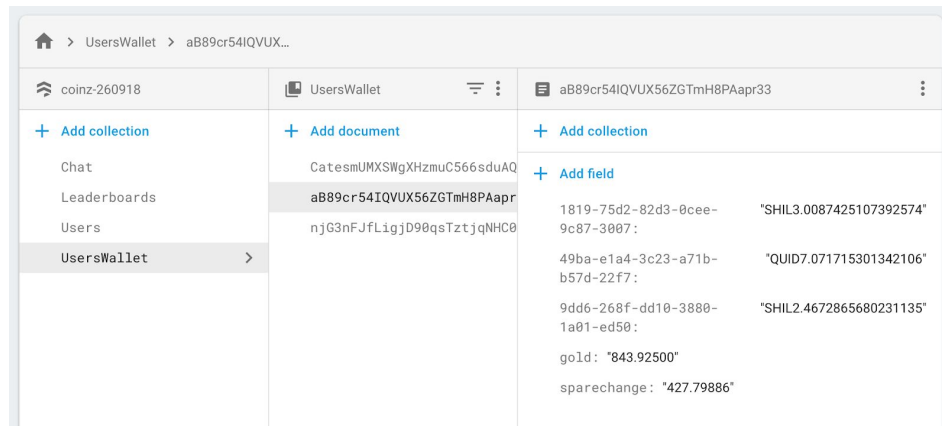
- The colours: The marker tiles are created from the mapbox website and are in the drawable folder so there are essentially 40 images. This implementation could be more efficient by potentially creating the images programmatically.

## Storage of user's progress in playing the game

There is a balance between the usage of Shared Preferences and FireBase storage. Initially, the idea of storing potentially everything in a XML seemed inviting but Shared Preference are used to store info that's needed across different activities and hard to pass through intents and not important enough to store in DB. For the same reason, the wallet is stored on FireBase and minimal data is stored in the Shared Prefs file.

The coins collected as stored as a HashMap and the values are set to the document at onStop(). When your activity is no longer visible to the user, it has entered the *Stopped* state, and the system invokes the [onStop\(\)](#) callback. Android documentation suggests that "You should also use [onStop\(\)](#) to perform relatively CPU-intensive shutdown operations. For example, if you can't find a more opportune time to save information to a database, you might do so during [onStop\(\)](#)".

The structure of database on Firebase:



Since database accesses are quite expensive, I tried to keep the structure and data as simple as possible.

## FireBase - UsersWallet

The UsersWallet collection is a collection of documents relating to unique users. It's unique because the user id generated by FireBase authentication are unique to each player in a game. Each of the documents contain the wallet of each user. The coins are stored as a key value pair where the coin id is the key and the the currency and value is stored as the the 'value' part. There are also *gold* and *spare change* fields which store the gold and spare change the user possesses.

The data insertion to one of these documents works as below:

```
//MainActivity.kt
override fun onStop() {
    super.onStop()
    //...
    val ref = FirebaseFirestore.getInstance().collection("UsersWallet")
    val documentId = prefs!!.currentUser

    ref.document(documentId).set(walletdb, SetOptions.merge())
        .addOnSuccessListener {
            Log.d(tag, "Things added")
        }
}
```

The gold and spare change fields are updated similarly from the wallet and messaging activity.

## SharedPrefs

- I implemented a Singleton pattern for the shared prefs to access the data across different activities. Since performance is not (normally) a high priority with reading and writing

preferences and lots of research, singleton seemed like a neat way of having everything in one place.

- Sample code of what it looks like:

```
//SharedPrefs.kt
class SharedPrefs(applicationContext: Context?) {
    private var pref = applicationContext?.applicationContext?.getSharedPreferences("MyPref",
MODE_PRIVATE)
    private var editor = pref?.edit()
    var currentUserName : String
        get() = pref!!.getString("current username", "")
        set(value) = pref!!.edit()!!.putString("current username", value).apply()
}
```

- Other information like the rates, daily coin constraints and id's of coin collected is also stored in Shared Preferences.

## ! Challengez overcome

- I took a decision to transfer data to FireBase after realising that shared prefs is not the brightest of ideas so creating a structure and transferring the data turned to be quite challenging.
- FireBase databases are asynchronous so all calls to read data are wrapped around callbacks to resolve that. Callbacks are essentially made to wait for the data before it's used.

## Detection of the coinz

```
//MainActivity.kt
for ((marker, coin) in markers.iterator()) {
    val loc1 = Location("")
    loc1.latitude = location!!.latitude
    loc1.longitude = location!!.longitude

    val loc2 = Location("")
    loc2.latitude = marker.position.latitude
    loc2.longitude = marker.position.longitude

    val distanceInMeters = loc1.distanceTo(loc2)

    if (distanceInMeters <= 25) {
        Log.d(tag, "Coin collected!")
    }
}
```

The detection of coins algorithm makes use of the distanceTo method in the Location API provided by Android. It takes in two location parameter and returns the approximate distance in meters between the locations. Distance is defined using the WGS84 ellipsoid. So, in the onLocationChanged method, I iterate through the markers on the map to see if the user within the range of a coin. The time complexity of the above algorithm depends on the number of marker on

the map at that given instance. If the number of markers is  $n$ , the runtime is  $O(n)$  that is linear. The `onLocationChanged` is called every time the user's location changes so ensures that it detects any nearby coins.

### Starring **Data Structures**

- `Loc1` and `Loc2` - `Loc1` is the location of the user and `Loc2` is the location of the marker/coin. The *Location* data type is initialised using a latitude and longitude.

## Banking coinz

The coins collected by the user are added to the FireBase in `onStop` as a `HashMap` as explained in the data storage. The coins never expire so they are forever in the wallet.

1. The user can then navigate to the wallet where the coins are shown as in recycler/card view.
2. Each of the cards have a deposit button which when clicked deposits the coin in the user's bank. On the button click, the gold value in the database will be updated.
3. The user can deposit 25 coins a day after which the coins deposited will be considered spare change\*
4. The user will be notified when their deposits as allocated as spare change.
5. Once deposited, the coin will be deleted from the wallet.

\* Sound a bit different? Well, it is. The user must\* deposit 25 coins a day to convert other coins to spare change. This is to ensure that the user cannot keep the 'good' coins and send their friends the 'bad' coins so the user will have to go out of their way to collect high valued coins to get more gold! I am aware that it's an unconventional implementation but it is my interpretation.

### Starring **Data Structures**

- `BigDecimal` - The coin values are stored as a `BigDecimal` data type. Working with doubles of various magnitudes (say  $d1=1000.0$  and  $d2=0.001$ ) could result in the  $0.001$  being dropped altogether when summing as the difference in magnitude is so large. With `BigDecimal`, this would not happen. The precision is set to 5 in the code for consistency and readability in the app.

## Messaging Coinz

A user can send other user gold coins by *amount* not by individual coins, but amount. There's no limit to how many coins a user can message to another user so I decided to give the choice of amount to the user. The user will be able to see if another user has sent them a coin through a text view.

1. The user can to the message activity wherein they enter an amount less than or equal spare change they possess.
2. On the button click, the spare change will make its way to the chosen user. The receiver will see the value and sender's name in the activity.
3. The secret recipe behind this implementation is the database structure.



Chat >	Message >	+ Add field
Leaderboards	m.likhi	Name: "gunsaw"
Users	whatevs	Text: "20"
UsersWallet		

4. When a user sends a message, the Chat collection will access the document (or create a new document) with the receiver's name as the doc id in which the *name* field is set to the sender's name and the text is set as the value of the spare change. The listener of user is set to listen to the values in the doc with their username.
5. When the user receives a coin, the amount is added to their existing gold coins.

## Starring Data Structures

- BigDecimal - The coin values are stored values are again stored as a BigDecimal
- The values to the document are set as a map.

```
//MessageActivity.kt
if (amtSend != null){
    if(!userSend.text.equals(prefs!!.currentUserName)){
        val newMessage = mapOf(
            NAME_FIELD to prefs!!.currentUserName,
            TEXT_FIELD to amtSend.text.toString())
        firestoreChat.document(userSend.text.toString()).set(newMessage)
        .addOnSuccessListener {
            Toast.makeText(this@MessageActivity, "Message Sent",
                Toast.LENGTH_SHORT).show()
        }
    }
}
```






## Ambiguities

- I did resolve the edge case of a user not being able to send coins to themselves.
- If the username is invalid the app user will not be aware of that in the current implementation. An ad-hoc solution would be store the usernames in a document and then implement a spinner chooser so that the user can choose a user from the existing database user but it's not a great implementation in terms of security as we're storing the users in a document. In the given time constraints, I did not get to resolve this but it is a priority issue
- Multiple users recognition- The current implementation does not recognised multiple users sending in coins. So for instance, if two people A and B send in coins, A's message will be overwritten by B unless the user opens the app before B sends the message. A solution would be two implement a recycler view to show the messages and add the gold but again tried to implement a simple messaging feature.

## Leaderboard

This is one of the bonus features implemented wherein a user can navigate to the leaderboard activity to see their position on the leaderboard. The algorithm for this is:

1. The user navigates to the leaderboard activity.
2. The activity class calls an adaptor to fill in the list with the data from the database. There is a different leaderboard database for a cleaner implementation. The structure of the database is as follows:

 coinz-260918	 Leaderboards 	 aB89cr54IQVUX56ZGTmH8PAapr33 
<a href="#">+ Add collection</a>	<a href="#">+ Add document</a>	<a href="#">+ Add collection</a>
Chat	aB89cr54IQVUX56ZGTmH8PAapr	<a href="#">+ Add field</a>
Leaderboards >	njG3nFJfLigjD90qsTztjqNHC0	name: "whatevs"
Users		score: "843.92500"
UsersWallet		

3. Each of the documents contain the username and score of the user.
4. When the data is retrieved from the database, it is ordered (using querying) in Descending order and the contents of the document are cast to a custom *User* object and are displayed as list items in the recyclerview.
5. The query code is as following:

```
db.collection("Leaderboards")
    .orderBy("score",
Query.Direction.DESCENDING).addSnapshotListener(EventListener<QuerySnapshot> { value, e ->
    val players = ArrayList<User>()
    for (doc in value!!) {
        if (doc.get("name") != null && doc.get("score") != null)
            players.add(User(doc.get("name").toString(),doc.get("score").toString()))
    }
})
```

## Ambiguities

- The value of gold is rounded to the nearest Integer for better readability of the scoreboard
- The leaderboard is made under the assumption that the username is unique for every user. The username creation in my implementation is created by slicing the email string with the @ delimiter but a randomised unique creation could be a better solution.

## Health

The user will be able to view their health stats on the health activity. The health activity consists of three parts

1. Steps: The steps are calculated from the step counter sensor in android phones.

```
val stepsSensor = sensorManager?.getDefaultSensor(Sensor.TYPE_STEP_COUNTER)
```

2. The distance walked by the user is calculated by recursively adding the distance in the onLocationChanged function of the maps activity. The distance is divided by 1000F to show the distance in kilometres.
3. The calorie count is obtained by an approximate calculation derived from a metric table from a fitness/lifestyle website.
4. Since it's a game that revolves around physical activity, the health activity hugely contributes to the quality of the app. Further ideas, it could also be integrated to IoT devices like FitBits and fitness related devices.

## Ambiguities

- The calorie count could be more precise in terms of the calculation.

## Parts of Design which have not been Implemented

*"Things don't always go as planned. It's called Life"*

I delivered all the major design proposals made in my plan except for an extra bonus feature.

## Hard Mode

In my plan, I mentioned that if time permits, I would implement an extra bonus feature in addition to the two main bonus features but due to some design and database structure modifications, I had to abort the bonus feature implementation and in favour of more critical features. But I did have the algorithm in place which essentially was

1. If user selects to play in the hard mode, the user would be able to see only one randomly selected coin from the coins on the map.
2. As soon as the user collects the coin, they would be able to see the next coin.

But I did implement additional features that were not documented in my plan. Listed under the next session!

## Additional Features Implemented

*“Serendipity - (n) Finding something without looking for it”*

### Share Feature (Bonus Feature)

Surprised I didn't think of this earlier but share feature is in my opinion, a crucial feature in a game in that regards that it enables the user to share their stats in an easy way and also, free advertising of the game. The algorithm works as follows:

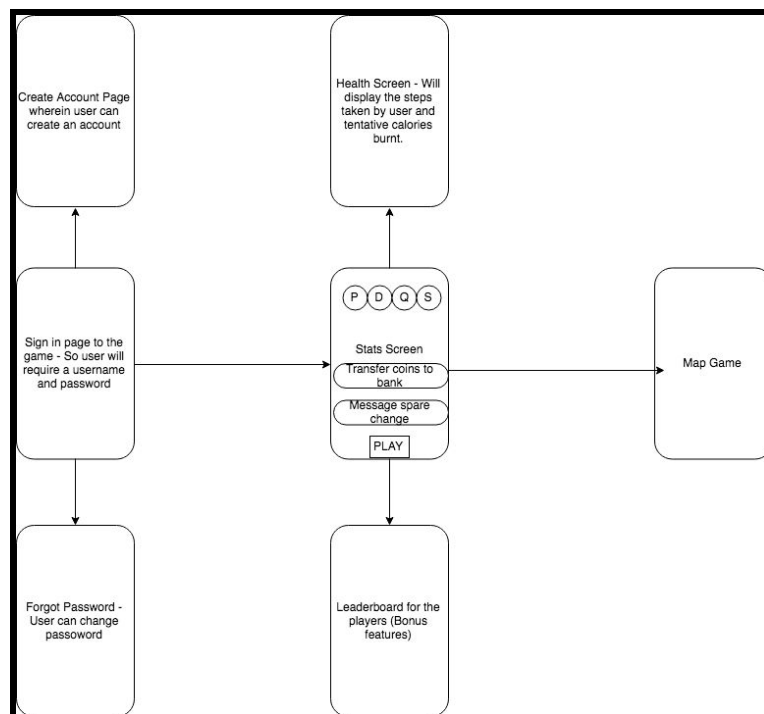
1. The Share Feature is created using the Share Intent feature.
2. On the landing page of the app, there's a share button which when clicked shows different social media platforms that the user can share their stats on. Have a look at the screenshots for more info on how it works!

### Progress of User Feature (Bonus Feature)

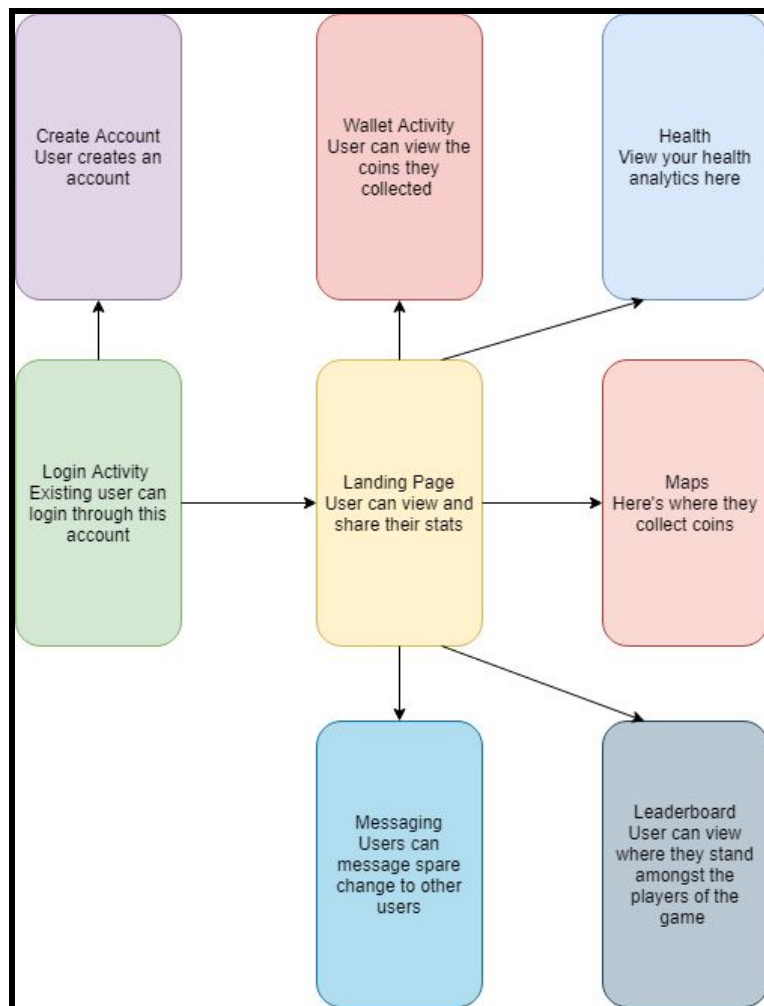
The user will be able to view their progress through a progress bar in the maps activity of the app. Since there's a limit on the number of coins deposited, a progress bar is a good way to continuously aware the user of where they stand on that particular day.

1. The progress bar is essentially a counter to the coins and is displayed in the Maps Activity.
2. The user will be able to view this in the Maps activity.

### Initial Design of the game



## Final Design of the game



## Did things go according to plan ?

No. I was a couple of weeks behind on my plan but I did leave a lot of time testing and did some contingency planning which turned out to work greatly in my favour. Apart from technical skills, I learnt a lot of soft skills like time management, risk management and contingency planning. Another important skill that I developed is thinking about edge cases. Manier times when coding, developers tend to mostly think about the basic implementation but there are edge cases to almost EVERYTHING.

## ScreenShots of Coinz in Use

*"Design is not just what it looks like and feels like. Design is how it works."* - Steve Jobs

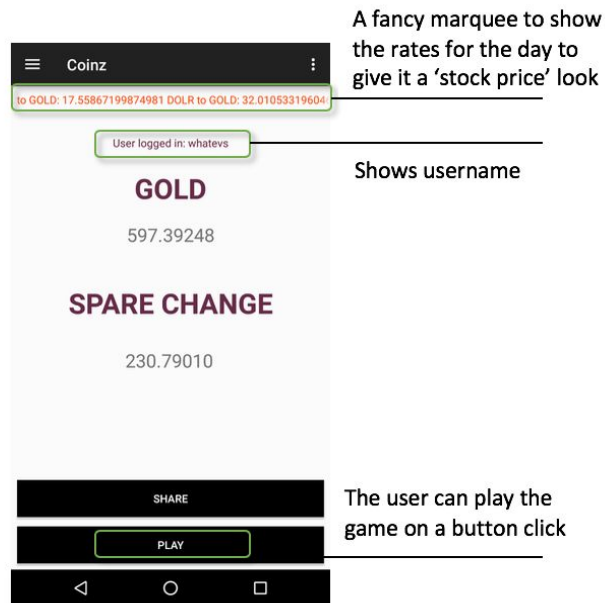
### Create Account Screen

The 'Create Account' screen features the Coinz logo and tagline 'Collect em' all' at the top. It contains two input fields: an email field and a password field. The left screenshot shows the email field populated with 'harveySpec@yahoo.com' and the password field with masked characters '\*\*\*\*\*'. The right screenshot shows the email field populated with 'donnaPaulsen@gmail.com' and the password field with the placeholder text 'Enter password'. A 'CREATE ACCOUNT' button is located at the bottom of each screen.

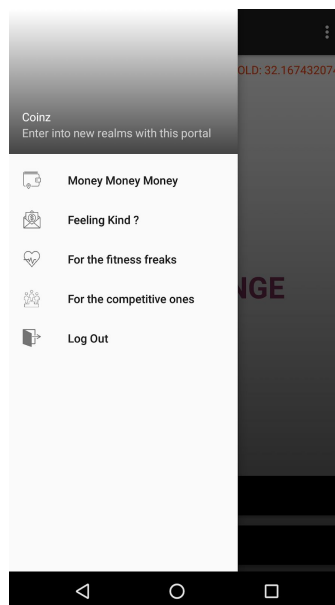
### Login Screen

The 'Login' screen features the Coinz logo and tagline 'Collect em' all' at the top. It contains two input fields: an email field and a password field. The left screenshot shows the email field populated with 'm.likhi@yahoo.com' and the password field with masked characters '\*\*\*\*\*'. The right screenshot shows the email field populated with 'harveySpec@yahoo.com' and the password field with masked characters '\*\*\*\*\*c'. A 'LOGIN' button is located at the top of each screen, and a 'CREATE ACCOUNT' button is at the bottom. The left screen also displays an 'Authentication failed.' message.

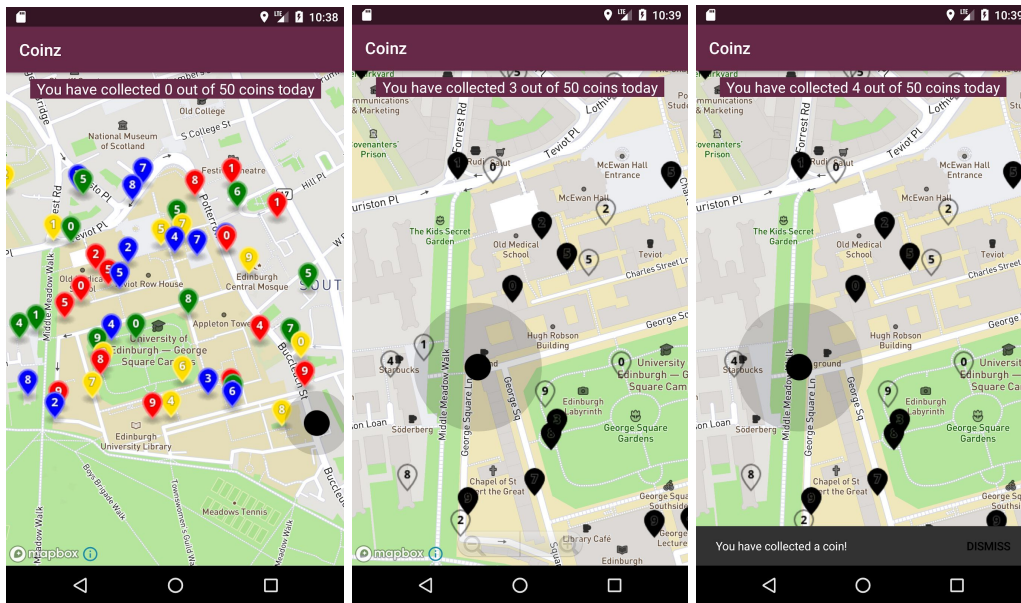
## Landing Activity



## Navigation Drawer



## Map Activity

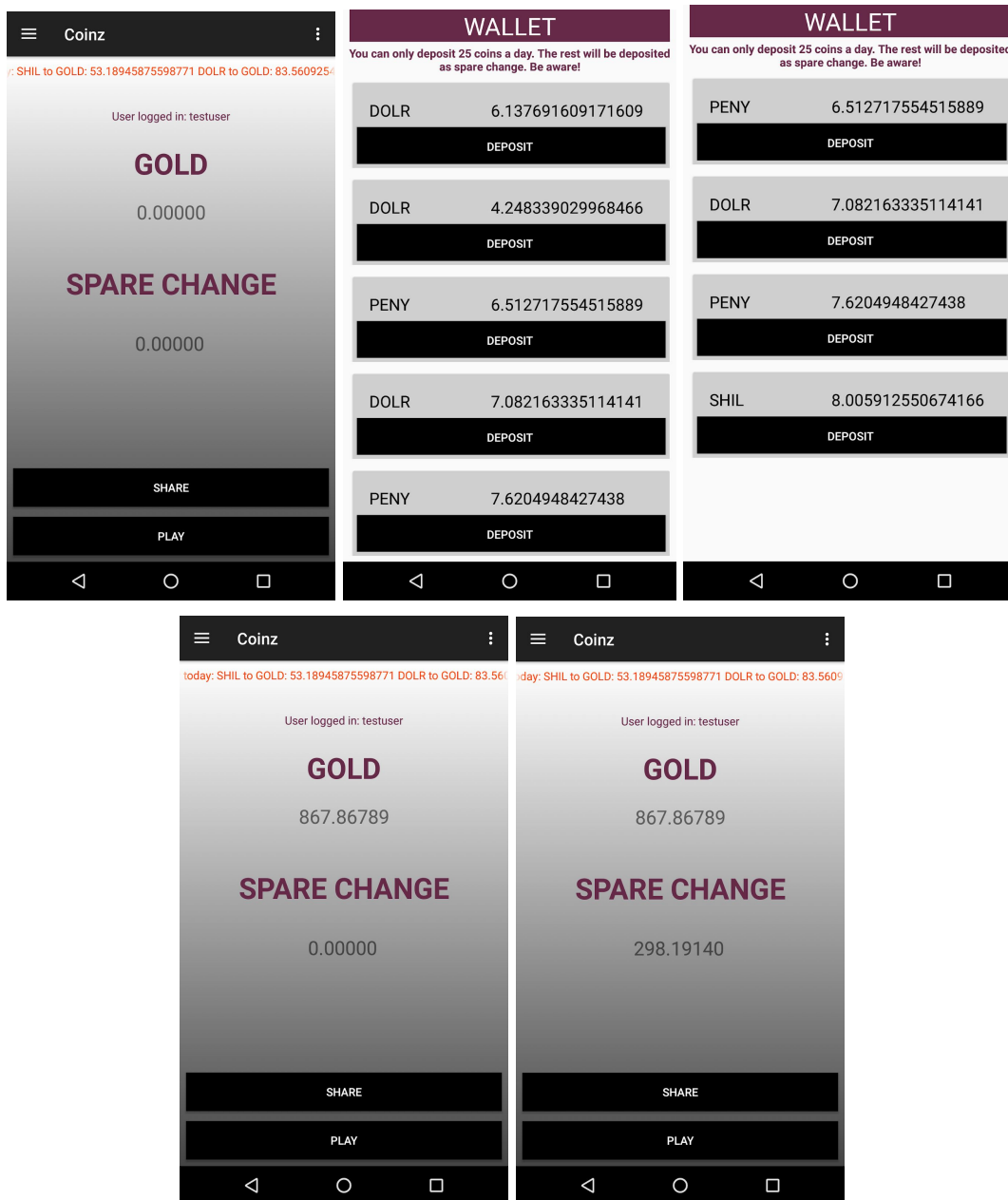


The first image shows the map when all 50 markers are on the map. In the second image, the user is approaching the coin with the symbol 1 and as soon as the user is within the 25 metre range, the coin is collected! A snackbar is shown to the user to notify the user that they have collected a coin but the collection of the coin itself does not require any user intervention.

The progress bar shown in the above picture is a bonus feature wherein the user can view the number of coins they have collected on that particular day.

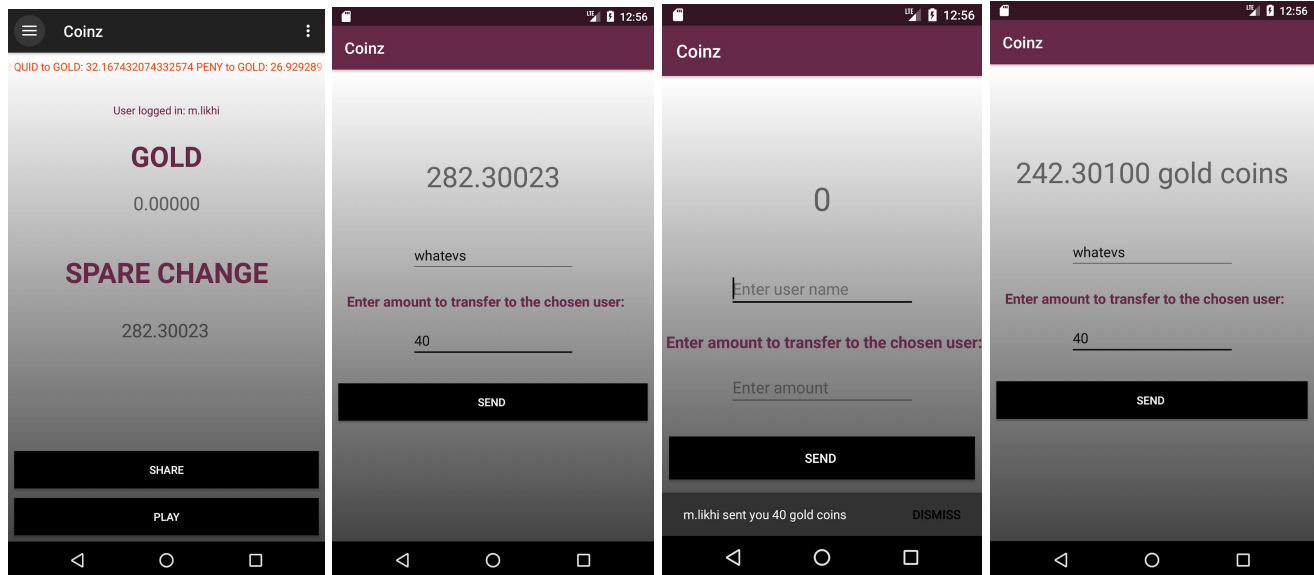


## Wallet Activity



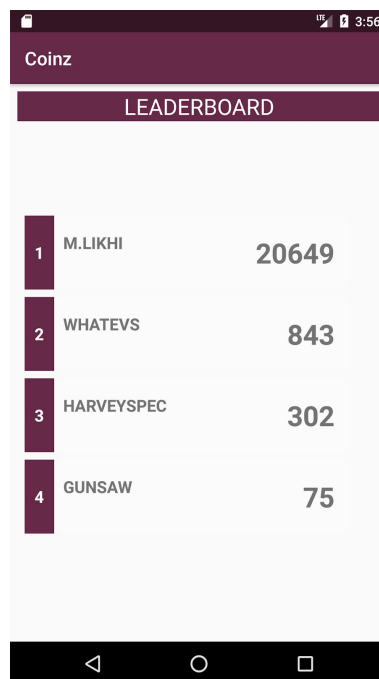
The figure shows the deposit of gold and spare change getting updated in the stats page. The wallet is implemented using a recycler view and a card view and the coins can be deposited on a button click.

## Messaging Activity



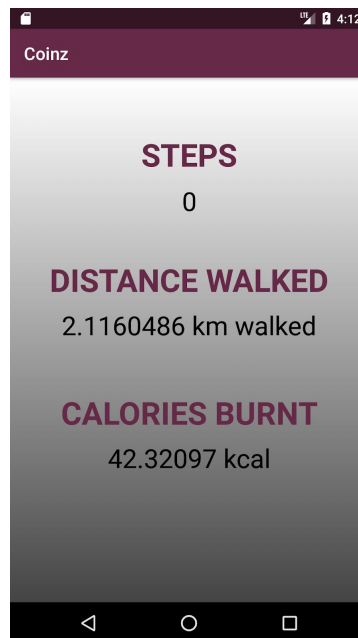
The first figure shows the status of user *m.likhi*. Second figure shows the user sending an amount of 40 gold coins to the user *whatevs*. The third figure shows a snackbar notification in *whatevs*'s phone. The fourth figure shows the new spare change of *m.likhi*

## Leaderboard Activity



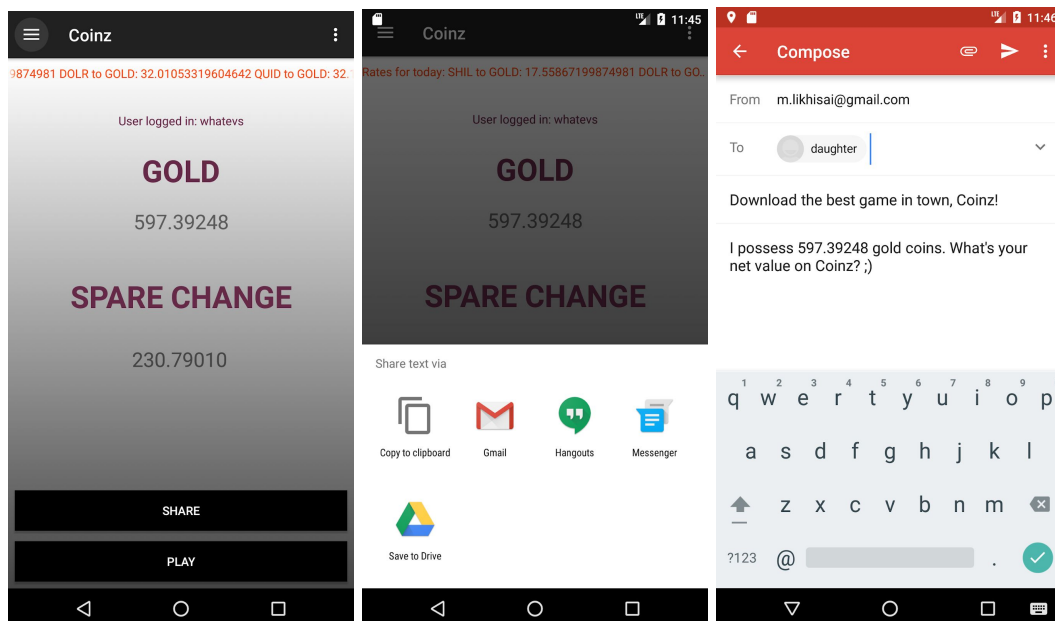
The leaderboard consists of the player ordering in the descending order of the scores.

## Health Activity



The steps will be updated real-time. Since there's no step counter in an emulator, the steps field is empty.

## Share Stats



## Acknowledgements

*"Special mention to Stack Overflow and Android Documentation without whom this project wouldn't have reach completion"*

### Everything Map related

- Video tutorials to help with MapBox setup (Source mentioned in lectures)
  - <https://www.youtube.com/watch?v=VzQEbGyGe4E>
  - <https://www.youtube.com/watch?v=QzxAwMxgBlg&t=1s>
  - <https://www.youtube.com/watch?v=Z5hGfsUt9pY>
  - <https://www.youtube.com/watch?v=jL5K5igH6pw>
- <https://www.mapbox.com/help/markers-js/>
- <https://www.mapbox.com/android-docs/api/map-sdk/5.0.1/com/mapbox/mapboxsdk/geometry/LatLng.html>

### Everything Storage related

- <https://blog.teamtreehouse.com/making-sharedpreferences-easy-with-kotlin>
- <https://guides.codepath.com/android/Storing-and-Accessing-SharedPreferences>
- Firebase related:
  - <https://cloud.google.com/firestore/docs/manage-data/add-data>
  - <https://cloud.google.com/firestore/docs/query-data/get-data>
  - <https://www.youtube.com/watch?v=kmTECF0JZyQ>

### Everything UI related

- <https://dzone.com/articles/android-example-progress-bar>
- <https://developer.android.com/guide/topics/ui/layout/recyclerview>
- <https://stackoverflow.com/questions/51944693/how-to-add-a-click-listener-to-my-recycler-view-android-kotlin>
- <https://www.codingdemos.com/android-recyclerview-example/>

### Everything bugs-related

- <https://stackoverflow.com/questions/12142255/call-activity-method-from-adapter>
- <https://stackoverflow.com/questions/27928/calculate-distance-between-two-latitude-longitude-points-haversine-formula>



The End. Oh yes, there's a green tick for all of my classes.