



INFORMATICS LARGE PRACTICAL – REPORT

By Kieran Litschel (s1614973)

Additional features that were not in the design

1. Users are notified when they receive a gift

When implementing sending gifts I realised I had not discussed in my plan how I was going to handle the user receiving gifts. I decided that if a user is using the app when they receive a gift, a toast should be displayed informing them of the gift amount, currency, and who it's from. If a user receives a gift while offline or not using the app, they should receive this toast when they come back online or reopen the app.

Parts of design not realised

1. The coin exchange

I under anticipated how complicated adding offline play would be when I was designing my app, and this meant I ended up spending far more time on the coursework than I anticipated, meaning I did not have time to implement the coin exchange.

2. The weekly leaderboard

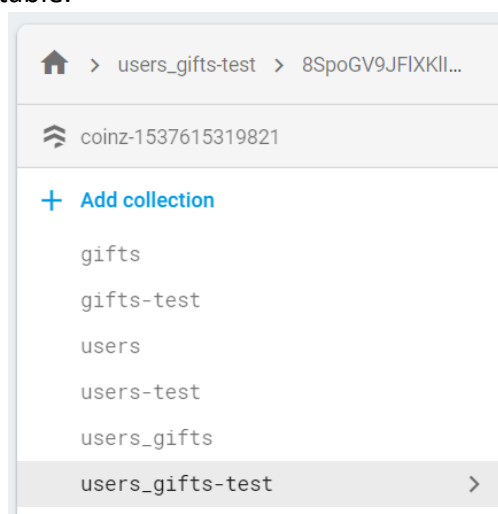
I ended up not using cloud functions for the leaderboard because I ran out of time. This meant I had to implement fetching the all-time leaderboard in a way that isn't as elegant as the way I presented in the design, but is functional with a small userbase, so is fine for the prototype. But I was not able to implement the weekly leaderboard as there was no way to do this without cloud functions, unless a single user updated all users weekly stats which seemed a bad way to do it.

Descriptions of Core Data Structures

Firebase Firestore

Collections structure

The primary storage location for data is in the Firebase Firestore, we have 3 collections, users, users_gifts, and gifts. We have two sets of these collections, one for users, and one for testers, we differentiate them by adding "-test" to the end of collections for testers. This allows us to keep user and tester data separate, ensuring espresso tests are repeatable.



A list of the 6 collections as displayed in the Firestore console.

Users collection

Each document in this collection has an id corresponding to the Firebase auth uid of the user it corresponds to. The uid in Firebase auth is fixed for each email address, and unique, so this is a suitable way to identify each user. Each of the user documents has 9 fields, each described below.

Name	Type	Default value	Purpose	Notes
DOLR	Number	0.0	Store the amount of DOLR the user has collected	
GOLD	Number	0.0	Store the amount of GOLD the user has collected	
PENY	Number	0.0	Store the amount of PENY the user has collected	
QUID	Number	0.0	Store the amount of QUID the user has collected	
SHIL	Number	0.0	Store the amount of SHILL the user has collected	
coinsRemainingToday	Number	25	Keep track of how many more coins the user can exchange with the bank today	
lastDownloadDate	String	-999999999-01-01	Keep a record of when the map was last downloaded, so we know when we need to download a new one	We set the lastDownloadDate to this default value as it is the minimum local date, so ensures that if we compare this date with the current one it is always before it. We store the date as a string type rather than a date as it will have to be parsed back into LocalDate anyway, so we might as well store it as a string to make this process simpler.
map	String	<i>Empty string</i>	Keeps a record of the markers that have been removed from the map	We could have also stored the map as a collection within the document, with a document for every marker. We opted not to as we'll need to be storing it as a string in the SharedPreferences file anyway, so it makes sense

				to store both in the same format.
username	String	<i>Empty string</i>	Keeps a record of the user's username, which is used for identifying them on the leaderboard and, for sending and receiving gifts	This means that the default value of the username isn't unique. But this is not a problem as we do not allow users to view the leaderboard or send gifts without setting a username, and we don't allow the recipient of a gift to be the empty string, so the fact it is not unique never poses a problem.

The exception to this document structure is that we have one document in the users collection called "usernames", where every field is a string, with each field name corresponding to the uid of a user, with it's value corresponding to the username of that user. We'll discuss in the core algorithms section why it is necessary to include this document.

It's worth noting that this means that the username is stored in two places in the database, which isn't typically good practice. We do this because the implementation using a usernames document is intended only for the prototype produced for this coursework, and we intend that eventually the usernames document will be deleted, leaving only the copy of each users username in their document.

Gifts collection

The purpose of this collection is to store the details on gifts, with each gift sent being stored in its own document and assigned a unique id automatically by Firestore. This collection is necessary to allow for a bonus feature I implemented where users are notified of who they have received gifts from. This necessitates storing the details of each gift as a user may be logged out when they receive a gift, so we need to keep a record of the gift to show them when they next sign-in. Each document consists of the following 3 fields.

Name	Type	Purpose	Notes
amount	Number	The amount of the currency the user has been gifted	
currency	String	The type of currency the user has been gifted	
senderUid	String	The uid of the user that sent the gift so that we can inform the user who sent them the gift	We could have also stored the sender's username instead, but I opted for the uid as the sender may change their username before the recipient opens the app and receives the gift, but by storing the uid we can always get the senders current username, so the recipient will always be able to identify who the gift is from.

Note that there are no default values for any of these fields, as when a gift is created all these values are known.

Users_gifts collection

This collection has a document for each user, which corresponds to their uid, and each document records the ids of gifts in the gifts collection that have been sent to the user that have not been shown to them. This information could also have been stored in the users collection, but I opted to make it its own document as for showing the gifts to the user in real-time we make use of a document snapshot listener, so by storing the gifts in a separate document the listener isn't triggered every time a user collects a coin or exchanges gold, which would lead to unnecessary queries of the Firestore.

Name	Type	Default value	Purpose
gifts	List of Strings	<i>Empty list</i>	Store the list of gift ids that the user has been sent but has not been shown yet.

SharedPreferences

In order to enable offline play, we needed to make a local copy of everything in the database, for this we used a SharedPreferences file. We keep a local copy of all the fields in the users document, and whenever we change a value in the Firestore, or we are shown an unseen gift, we update the local values in the SharedPreferences file to reflect the change.

One of the key fields we store in the SharedPreferences file that we don't in the Firestore are deltas. We have a delta for all values that can be incremented or decremented whilst offline, and whilst the user is offline we change these deltas rather than the currencies' values in the SharedPreferences file. We do this as the value of coins can become out of sync whilst the user is offline, for example if they receive a gift whilst offline, the coins value in the database will be updated, but it won't be updated locally. When the user comes back online, the values locally and in the database are updated with the deltas, and then the deltas are set to 0.

Name	Type	Default value	Purpose
DOLRDelta	Number	0.0	Keep track of changes in DOLR whilst offline.
GOLDDelta	Number	0.0	Keep track of changes in GOLD whilst offline.
PENYDelta	Number	0.0	Keep track of changes in PENY whilst offline.
QUIDDelta	Number	0.0	Keep track of changes in QUID whilst offline.
SHILDelta	Number	0.0	Keep track of changes in SHIL whilst offline.
coinsRemainingTodayDelta	Number	0	Keep track of changes in coinsRemainingToday whilst offline.
lostConnectionDate	String	<i>Empty string</i>	Keep track of the when coinsRemainingToday last changed, so we know whether we need to update coinsRemainingToday with the coin delta when we come back online. As if the day has changed then we don't need to, as the limit will have been refreshed, so the delta is no longer applicable.

Description of Core Algorithms

Writing to the Firestore

There are a lot of cases where we read a value from the Firestore, change it, then write the new value back to the Firestore. We have to be cautious when doing this, as the program flow of android apps is asynchronous, meaning that in rare cases we may be trying to update a field in two places in the code at the same time, which could cause the value in the Firestore to become inaccurate. For example, consider the below:

1. A user collects 5 SHIL, which we want to add to the value in the Firebase, so first we need to read its value.
2. If after we read its value but before we write the new value back, a gift comes in, then the value of SHIL has changed in the Firestore.
3. This means that when we write the new value back, it will not take into consideration the fact that a gift has come in, meaning the amount added for the gift will be lost.

We resolve this by using transactions to write to the Firestore. In a transaction if the document changes before reading the value and writing it back, then the transaction is restarted. This means that it solves the risk of potential errors occurring from concurrent writes.

Writing to the SharedPreferences file

As we're keeping a copy of the values in the Firestore in the SharedPreferences file, we can run into similar problems when writing this file. For example, if a gift is seen just as we're in the middle of collecting a coin of the same currency as the gift, then this can lead to a similar error in the SharedPreferences file as we saw could happen with the Firebase.

Solving this problem is more complicated than with Firestore, as SharedPreferences files do not have transactions. So instead of this I opted to use threads and stamped locks to solve the problem. Stamped locks are implemented for this exact purpose, as only one part of our program can hold the lock at any point, and if another part of the program reaches a point where it needs the lock, then it waits until the other part releases the lock. Their behaviour is also particularly desirable as there is a queue to acquire the lock, meaning if it reaches midnight, and the map changes, but we are in the process of collecting coins, then the tasks to collect coins will be ahead of that for updating the map in the queue, which ensures the coins are collected before the map is set to the next day.

One important thing to note is that in order to use locks, in a lot of cases we will need to run the code which requires the lock on a new thread. This is because waiting to acquire the lock on the activity thread will lead the app to freeze, which is undesirable. The flow of a section of code that edits the SharedPreferences file takes this basic structure:

1. The task to edit the SharedPreferences file is started on the new thread.
2. When the task is run, it tries to acquire the lock, if another thread has the lock, then the thread joins the queue to acquire the lock.
3. Once it has acquired the lock it runs its code editing the SharedPreferences file, assured that nothing else will edit the SharedPreferences file while it has the lock.
4. Once it has done editing the SharedPreferences file and there is nothing else to do, it will release the lock, so the next task can acquire it.
5. If there is more to write to the shared preference file outside the task, then we callback to the class that created the task, via a callback through an interface implemented in that class, to which we pass the lock.

6. Once we have finished changing the shared preferences file outside the task, we release the lock, so the next task can acquire it.

Note that this is also why we use StampedLocks as opposed to ReentrantLocks, so that the lock can be released on a separate thread to which it was acquired.

Updating the map with today's map

1. Whenever the day changes, or the user reopens the MapFragment, the checkForMapUpdate is run, which creates a task on a separate thread to check if the map requires an update.
 - a. If the MapFragment is opened and there is an existing map in the SharedPreferences file, then we display that, so that if it turns out we have the most up to date map, then a map is already being displayed
2. The task acquires the lock or joins the queue to acquire it if another thread has it.
3. Once the task has the lock, it checks if the lastDownloadDate is before today, if it is not then the lock is released, and we do no more.
4. If lastDownloadDate is before today, then we need to download a new map, so we callback to the MapFragment, passing the lock back to it
 - a. Note that we do not check if the day has changed outside the task, as it could be the case that we are in the middle of updating the map already, so if we checked outside the task before we had the lock we would end up downloading the new map twice.
5. The MapFragment clears the map and markers.
6. It then checks whether there is an internet connection, if there isn't then it marks that the map should be updated the next time we connect to the internet, releases the lock, and then informs the user through a toast the map will be updated when we next connect to the internet.
 - a. We inform the MainActivity that we need to update the map, which is keeping track of when the internet connection changes, so next time there is an internet connection and it sees that we are waiting to download the map, it will call checkForMapUpdate, to start this process of updating the map again.
7. If there is an internet connection, then determine the url of the GeoJSON we need to download using today's date, and then create a DownloadMapTask, like the one discussed in lectures, that attempts to download the map from the server. We also pass this task the lock.
8. If the download fails then the lock is released, and we do no more.
 - a. This is acceptable as although this will be an annoyance to the user, it causes no serious issues, as once they close the app or close and reopen the MapFragment, this process will be tried again.
9. If task is successful then we take the downloaded GeoJSON and store it as a string (called map), which we then pass back to the MapFragment along with the lock through the DownloadCompleteRunner
10. We take note of the date for the value of the lastDownloadDate and write this along with map string to the Firestore, updating the users document.
 - a. We do not need to worry about using a transaction here, as only the users app can write these fields, and no other method can write these fields without holding the lock.
 - b. Whilst doing this we also reset coinsRemainingToday to 25, because as we have downloaded a new map, it must be a new day.
11. If the update fails, then we release the lock and do nothing more.
12. If the update is successful then we write the updated values to the SharedPreferences file too, and call a final method to update the markers, passing it the lock and the map string.
13. We try to parse the map string as a JSONObject, if this fails then the stack trace is printed, the lock is released, and we do no more.

14. If parsing it is successful then we extract the “features” (which stores the markers) from it as a JSONArray, meaning we have an array of the JSONObject’s of each marker in the map.
15. We iterate through the array and extract the id, coordinates, colour, value, and currency from each marker’s JSONObject in the array.
16. We add each marker to the map at the position specified with the specified colour, and set that when it is clicked on, a title is displayed showing its value and currency. We also store the MarkerOptions object that created each marker, along with a HashMap that relates each MarkerOptions object to its respective id.
17. We then release the lock, and we have successfully updated the map.

Collecting coins

1. Whenever the MapFragment is opened, a location listener is created, which listens for changes in the location of the user.
2. If there is a location change, then the listener is triggered, which checks if the user is within 25m of any marker and stores a list of all markers that we’re within 25m of.
3. If we’re less than 25m from at least one marker, then we call the removeMarker method, passing it the list of these coins.
4. The remove markers method creates a RemoveMarkersTask which is run on a separate thread, which removes these markers.
5. In the task we acquire the lock, and if it is already held by another thread then we wait to acquire it.
6. Once we’ve acquired the lock, we load the map from the settings file and attempt to parse it as a JSON Object.
7. If we fail to parse it as a JSONObject then the stack trace is printed, the lock is released, and we do no more.
8. If we successfully parsed it, then we extract the features as a JSONArray, giving us an array of the markers.
9. For each marker in the JSONArray, we compare it against the ID of each marker being removed, to find the JSONObject each marker corresponds to.
 - a. When we find a match, we remove the JSONObject from the JSONArray, so that when we reload the map we won’t add that marker again.
10. We convert the JSONObject with the markers removed back to a string, and this forms our new value for map.
11. If we ended up removing at least 1 marker from the JSONArray, then we need to update the value of the map locally and in the cloud, and the currency values with the collected amounts.
 - a. If we ended up removing no markers from the JSON array, then we release the lock and do no more.
12. If there’s an internet connection then we proceed to do this via a transaction, reading the users document, extracting their current values for each currency and adding the amount just collected to each value in the Firestore, before updating the users document in the Firestore with the new values and the new map string.
 - a. If the transaction is successfully then we update the SharedPreferences file with these new values as well.
 - b. If the transaction fails then the lock is released.
13. If there’s not an internet connection, then we update the deltas with the new values in the SharedPreferences file, along with the new map string, and if we’re not already waitingToUpdateCoins, we inform the MainActivity that we are

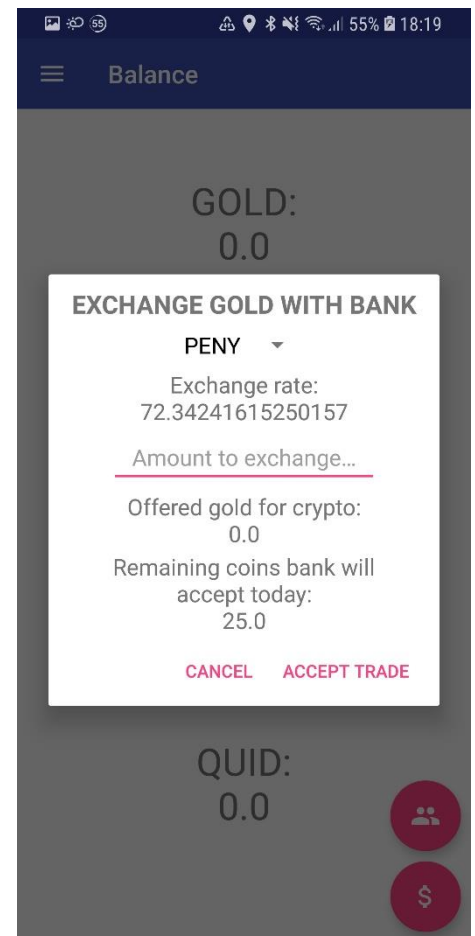
- a. This allows for functionality similarly to waiting to update the map, so that when we come back online, the MainActivity creates a task to update the Firestore with the coin deltas, along with the changed map.
14. Either way, once we have completed step 12 or 13, we callback to the MapFragment through the RemoveMarkersCallback interface, calling onMarkerRemoved.
15. In the MapFragment we remove the markers collected from the map and display a toast to the user stating what coins we collected, before releasing the lock.

Showing the users balance

We display the users coin balances in the BalanceFragment. As a copy is stored of each currency locally in the SharedPreferences file, we can just display the user these values whenever the BalanceFragment is opened, adding the delta for each currency whilst offline.

Exchanging cryptocurrency for gold with the bank

1. The user exchanges cryptocurrency with the bank through the BalanceDialogFragment, which is opened through the FAB with the dollar sign on the balance screen.
2. The user enters their desired exchange through a form, as shown in the screenshot to the right. Clicking on “PENY” will bring up a spinner which allows the user to select which of the 4 cryptocurrencies they’d like to exchange. When the user selects the cryptocurrency the TextView is automatically updated below to show the exchange rate for the respective cryptocurrency. The user enters the amount they want to exchange with the bank through the EditText below this, the contents are checked automatically as the user types the amount they’d like to exchange, so if they enter an amount that exceeds the amount of gold they can exchange today or the amount of the cryptocurrency they have, then the value in the field will be updated to the maximum they can exchange. As the user types the two TextViews below it are updated as well. The first shows the user how much GOLD the bank will give in exchange for the currency, and the second shows the remaining amount of cryptocurrency the user can exchange with the bank today after the proposed trade. The user can submit the trade by clicking the “Accept Trade” button. Or cancel it by clicking the “Cancel” button or clicking outside the dialog.
3. If the “Accept Trade” button is pressed, and the amount put forward for the trade is greater than 0, then we callback to BalanceFragment to execute the trade. This marks that we are executing a trade (so another can’t be started before the current completes) and creates a new ExecuteTradeTask, which is run on a new thread.
4. In the task we first acquire the settings write lock, if it is already held by another thread then we join the queue to acquire it.
5. Once we’ve acquired the lock, we update the currency values in the balance fragment to reflect the result of the exchange.



6. Next, we check if there is an internet connection, if there is we update the currency values in the database through a transaction, and also decrement the coinsRemainingToday value by the amount of crypto exchanged to the bank.
 - a. If this is successful, then we update the same values locally and then release the lock.
 - b. If this fails, then we revert changing the currency values in the balance fragment and release the lock.
7. If there is no internet connection, we update the coins remaining and currencies deltas locally. If we're not waiting to update coins, then we inform the MainActivity we are waiting. We then release the lock.
 - a. If we're waiting to update coins when we reconnect to the internet, then the same functionality is executed as describe is step 13a of collecting coins.
8. No matter the outcome in steps 6 and 7, we callback to the BalanceFragment with onTradeComplete.
9. In the BalanceFragment we mark that we are no longer executing a trade, run a method to update the balances being displayed, and show a toast to inform the user whether the exchange was successfully executed or not.

Creating / Changing Username

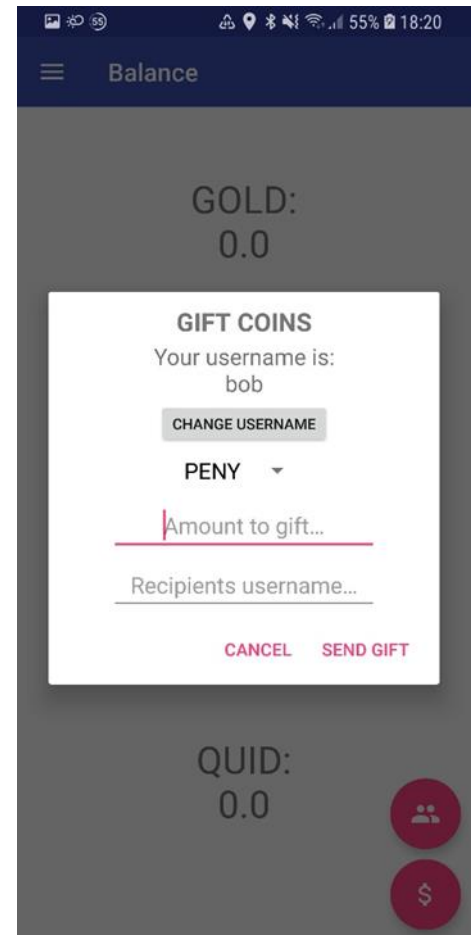
The process for creating/changing the username is the same, with the UI being the only big difference, so we describe them both at once.

1. First the user fills in their desired username and clicks the "Change Username" button or "Create Username" button.
2. If the desired username is not the empty string, then we check if their username is valid.
3. If the user is changing their username, rather than creating it, and the new username is the same as their old one, then they are warned that their new one can't be the same as their old one, and nothing else is done.
4. Otherwise we begin a transaction to check whether the username is in use, and if not update it with the desired one in the database.
 - a. First, we fetch the usernames document from the users collection and check if the username is listed in there. If so, we throw a Firestore exception which is caught by the onFailureListener, which then warns the user the username is in use, and nothing else is done.
 - b. Otherwise we update the users username in the usernames document and update the username in the users document.
 - i. Note that this does create redundancy, but we only intend to be the usernames document to be used in the prototype.
 - ii. It is necessary to include a usernames document as between getting the list of usernames and writing our new username to the database, another user could update their username with the same one. Using a single usernames document and transactions ensures that if any user updates their username mid-transaction, then it will fail and be retried, meaning there is no chance of two users having the same username. The downside to this is if lots of users trying to change their username at once, this could lead to the transactions failing entirely, hence why this functionality is only intended for the prototype.
 - c. If the transaction fails for a reason other than the username already being in use, then the dialog is dismissed, and a toast informs the user to try again.

- d. If the transaction is successful, then the value for the username is updated locally, the dialog is dismissed, and a toast is displayed informing the user of the success in changing their username.

Sending gifts

1. The user sends gifts to other users through the GiftExchangeDialog, which is opened by clicking the FAB with the people symbol on the balance fragment.
2. If the user presses the gift FAB but there is no internet connection, they are informed in a toast that sending gifts requires an internet connection, and nothing else is done.
3. If the user presses the gift FAB but has not created a username, they are first asked to create one. Otherwise they are presented with the GiftExchangeDialog, as shown on the right. At the top of the dialog we display their current username and a button they can click to change it, which will open the change username dialog. Below this there is an input form for sending gifts, allowing them to select the currency they wish to send and the amount. Both behave the same as in the exchange, with the only difference being the EditText only corrects the amount to gift if it exceeds the amount of currency the user has, as there is no limit on the amount that can be gifted. Finally the user enters the username of the recipient they wish to gift coin to and presses the send gift button. They can also cancel sending a gift by clicking cancel, or clicking outside the dialog.
4. If they click the send gift button, and the recipient box is not empty, and the amount to gift is greater than 0, then we query the Firestore users collection for the document where the username is the recipients.
5. If the result of the query is empty, then username is not in use in the Firestore, so we display in the dialog we couldn't find the recipient and do nothing more.
6. If the result of the query isn't empty, then the first result is the document corresponding to the recipients, as we enforce usernames are unique. So, we can just take this document, get its ID, and then we have the ID of the user.
7. Next, we try to send the gift to the recipient, which we do via a database transaction.
 - a. If the recipient does not have the same uid as the sender, then we subtract the value being gifted from the senders document and add it to the value in the recipients document.
 - i. Note we allow people to send themselves gifts as it doesn't break any core functionality and makes seeing the gift listener in action simpler.
 - b. Next we create the gifts document, placing in it the senders uid, the name of the currency gifted, and the amount gifted of that currency. We allow Firestore to assign an automatic id to each gift.
8. If the transaction fails, we inform the user something went wrong, dismiss the dialog, and do no more.
9. If the transaction is successful, then we dismiss the dialog, and display to the user a toast informing them the gift was sent successfully.
10. After displaying its success, we create a new CoinsUpdateTask, which waits for the lock, before writing the changes made in the database to the SharedPreferences file, afterwards releasing the



lock. After this, it calls back to the BalanceFragment with the updated currency values, and forces updates the displayed balances to reflect the changes.

Receiving gifts

1. Whenever we restart the app, we show the user any new gifts
 - a. We'll describe the procedure of "showing gifts" in the next section
2. We make use of a snapshot listener to listen for gifts, by listening to the users document in the users_gifts collection. This means that we receive gifts while using the app without having to restart it.
 - a. If there is a change to their users_gifts document then an event is triggered in the listener, which gets the list of gifts from the document. If the list is not empty, then we remove the snapshot listener, and show the user the gifts, otherwise we do nothing and continue listening.
3. If the internet is disconnected then the listener is removed, and we recreate it when the user reconnects to the internet.

Showing gifts

Showing gifts is the process of the user's app receiving the gift, updating the local values, and showing a toast to the user notifying them of the gift.

1. The first step to showing a gift is to get the usernames document from users, so we can match the senders uids to their usernames.
2. Next, we start a transaction to get the gifts from the database and remove them.
3. First, we get the users document in users_gifts and extract the list of gift ids.
4. For each gift id in the list, we fetch the corresponding document from the gifts collection and extract the currency and value of the gift, along with the senders uid. We then get the field in the usernames document corresponding to the uid, giving us their username, which we then store along with currency and value of the gift in an array detailing the each of the gift's values.
5. Once we've stored all details of the gifts in the array, we delete the gifts from the gifts collection, and we set the value of the list of gifts in the users document in the gifts collection to the empty list.
6. At the end of the transaction we return the array of the gifts details, to pass to the success listener.
7. If the transaction is successful, then we set up a new listener for the users document in the users_gifts collection, with the functionality described in 2a of the receiving gifts section.
8. Next, we inform the user of each gift in a toast.
9. If we haven't just logged in, then these gifts have come in since we last updated the local values with the values from the database, which means we need to update the local values with the changes made by the gifts using a CoinUpdateTask.
10. The CoinUpdateTask behaves the same as described in step 10 of sending gifts, and if we received the gift while viewing the balance fragment, it ensures the balance is updated in the UI to reflect the change in balances.

Updating the leaderboard

The same flow of code is executed when the user first opens the LeaderboardFragment or hits the refresh button.

1. First, we check if the internet is available, if it isn't, we inform the user through a toast that we can't update the leaderboard without an internet connection, and do nothing more.

2. If there is an internet connection, we check if the user has set a username, if not we ask them to create one by directing them to the create username dialog.
3. If the user has an internet connection and has set a username, then we begin the process of updating the database.
4. First, we set all rows in the leaderboard to invisible, as the number of rows we show can vary between updates.
5. Next, we get all documents in the username collection, and from each that relates to a user (so not the usernames one), we extract the username of the user and their amount of GOLD, storing this in an object, which we then store in an array. We then sort this array into descending order by the value of GOLD.
 - a. Note that this method of ranking all users is inefficient, as sorting has a worst-case time complexity of $n \log n$, so if we sort them every time the user hits refresh, this quickly becomes a very inefficient. This method is suitable for the prototype as we will have a small userbase, so time complexity is not a large concern as n is small. But for the implementation it would make more sense to have a ranking of users that is stored in the Firestore, which is updated at intervals using cloud functions.
6. Then we search the sorted array to find the position of the user in the rankings, and we take note of their rank and amount of GOLD.
7. If there is less than 10 users in the database, we display all users, scores, and make the rows we fill visible.
8. If there is more than 10 users in the database, then we display the top 10, and show all rows.
9. If the users rank is outside the top 10, then above the leaderboard we show the users rank and how much gold they've collected.

Screenshots of app in use

Logging In

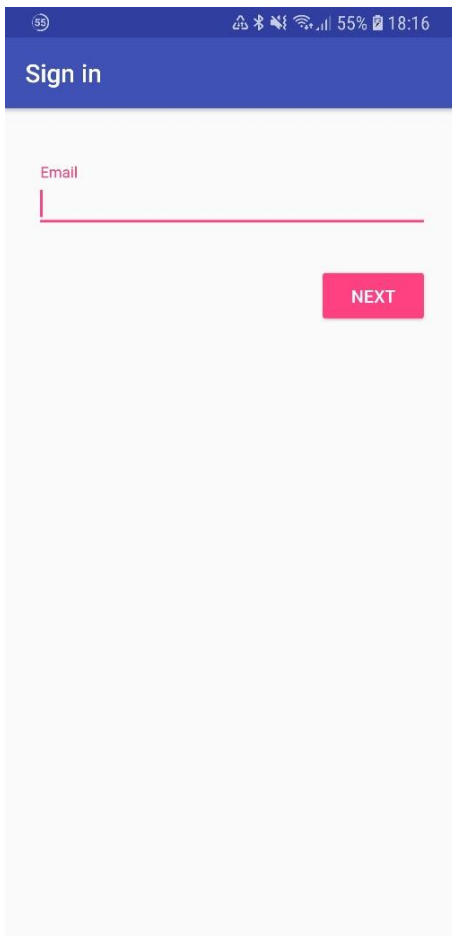


Figure 1 (left) - Sign in with email screen

The screenshot shows a mobile app interface with a blue header bar containing the text "Sign in". Below the header, there is a light gray background. A pink label "Email" is positioned above a pink input field. To the right of the input field is a pink button labeled "NEXT". The status bar at the top shows a battery icon, signal strength, 55% battery, and the time 18:16.

Figure 1 (left) - Sign in with email screen

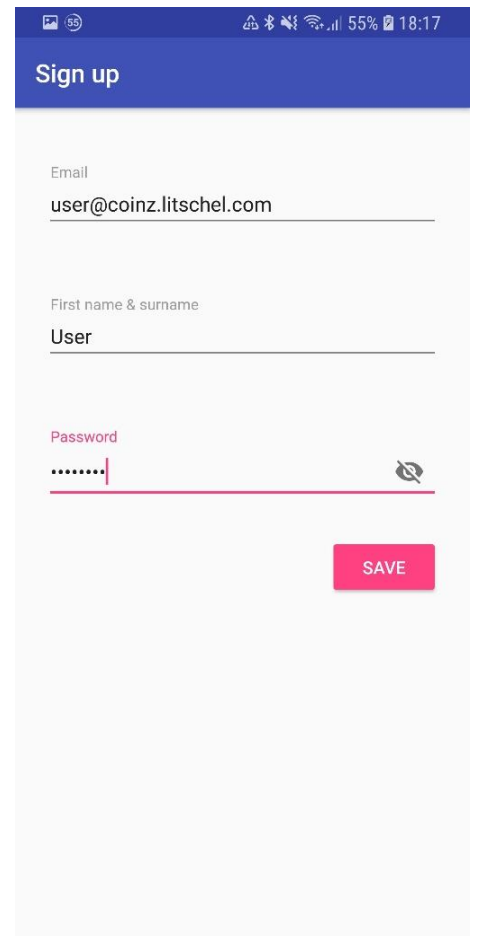


Figure 2 (right) - Sign up screen, displayed after user has pressed next on the "sign in with email" screen, and the email address has not been used in the app before

The screenshot shows a mobile app interface with a blue header bar containing the text "Sign up". Below the header, there is a light gray background. There are three input fields: "Email" with the value "user@coinz.litschel.com", "First name & surname" with the value "User", and "Password" with a masked password ".....". To the right of the password field is a pink button labeled "SAVE". The status bar at the top shows a battery icon, signal strength, 55% battery, and the time 18:17.

Figure 2 (right) - Sign up screen, displayed after user has pressed next on the "sign in with email" screen, and the email address has not been used in the app before

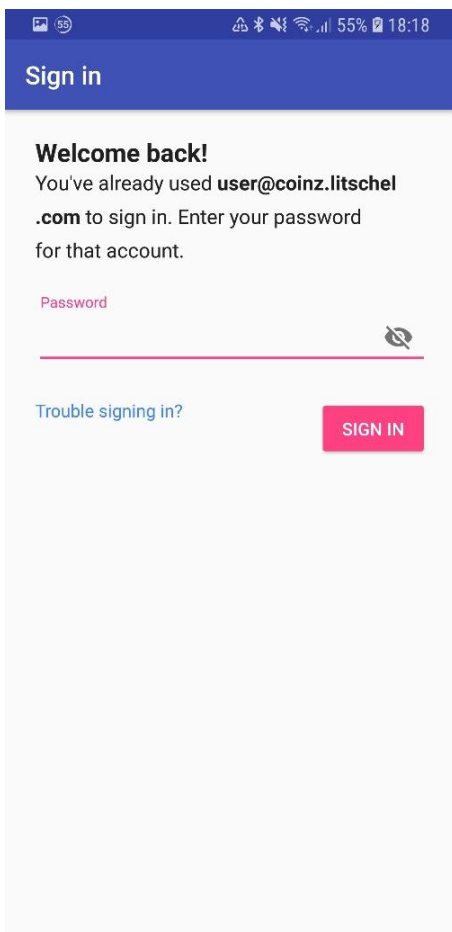


Figure 3 (left) - Enter password screen, displayed after user has pressed next on the "sign in with email" screen, and the email address has been used in the app before

The screenshot shows a mobile app interface with a blue header bar containing the text "Sign in". Below the header, there is a light gray background. The text "Welcome back!" is displayed, followed by "You've already used user@coinz.litschel .com to sign in. Enter your password for that account." Below this text is a pink label "Password" above a pink input field. To the right of the input field is a pink button labeled "SIGN IN". A link "Trouble signing in?" is located below the input field. The status bar at the top shows a battery icon, signal strength, 55% battery, and the time 18:18.

Figure 3 (left) - Enter password screen, displayed after user has pressed next on the "sign in with email" screen, and the email address has been used in the app before

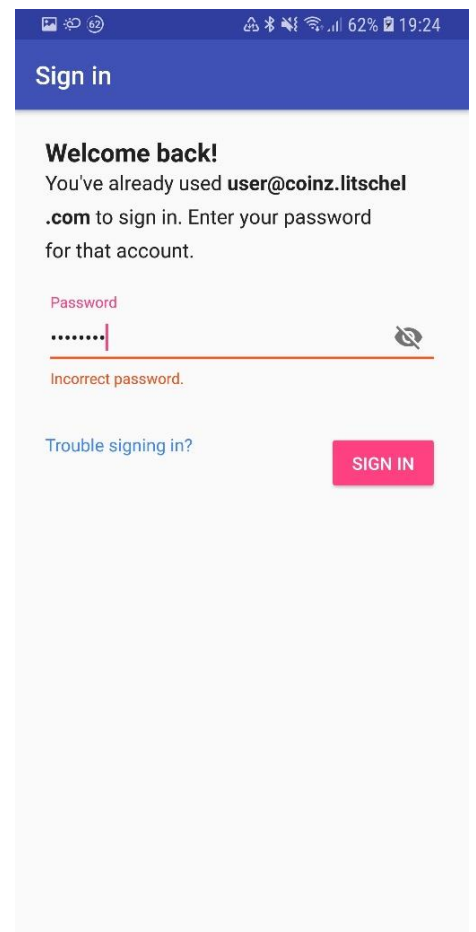


Figure 4 (right) - Enter password screen showing result of entering incorrect password

The screenshot shows a mobile app interface with a blue header bar containing the text "Sign in". Below the header, there is a light gray background. The text "Welcome back!" is displayed, followed by "You've already used user@coinz.litschel .com to sign in. Enter your password for that account." Below this text is a pink label "Password" above a pink input field. To the right of the input field is a pink button labeled "SIGN IN". A link "Trouble signing in?" is located below the input field. Below the input field, the text "Incorrect password." is displayed. The status bar at the top shows a battery icon, signal strength, 62% battery, and the time 19:24.

Figure 4 (right) - Enter password screen showing result of entering incorrect password

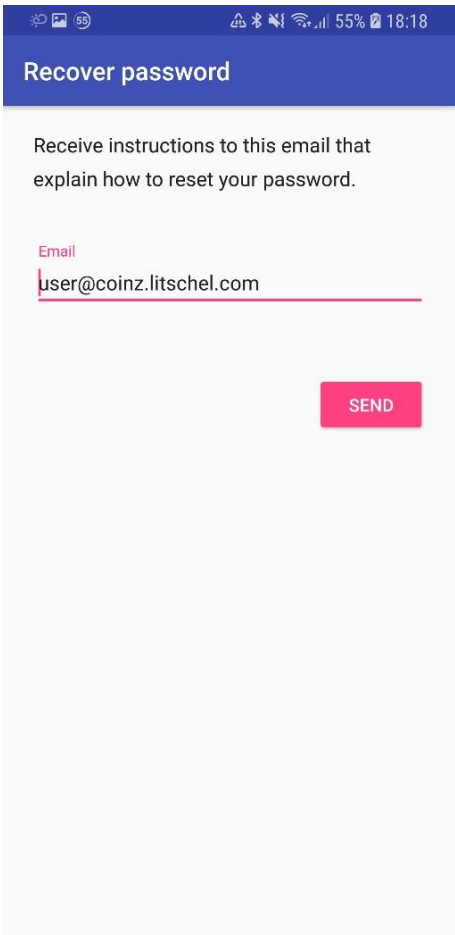


Figure 5 - Recover password screen, shown if "Trouble signing in" is clicked on "password sign in screen"

Map

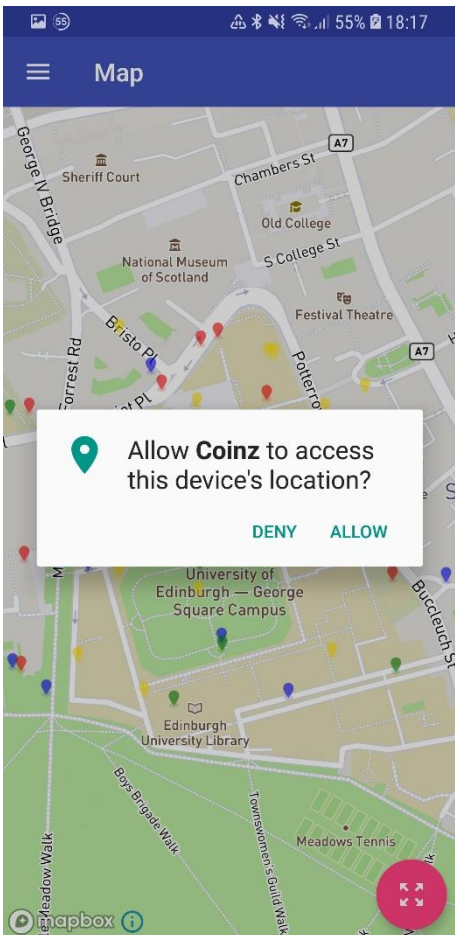


Figure 6 (left) – If it's the users first time installing the app on this device, they're asked to give permission to access their location. If they deny it the app still works, but they will not be able to collect coins or see their location.

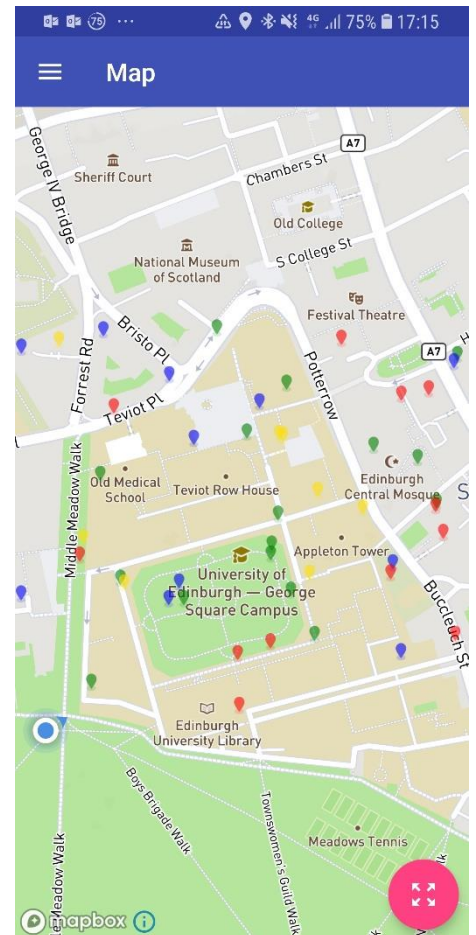


Figure 7 (right) – The view of the map, which shows the position of markers and the user's location, the zoom out FAB at the bottom left resets the view of the map to the default displayed here

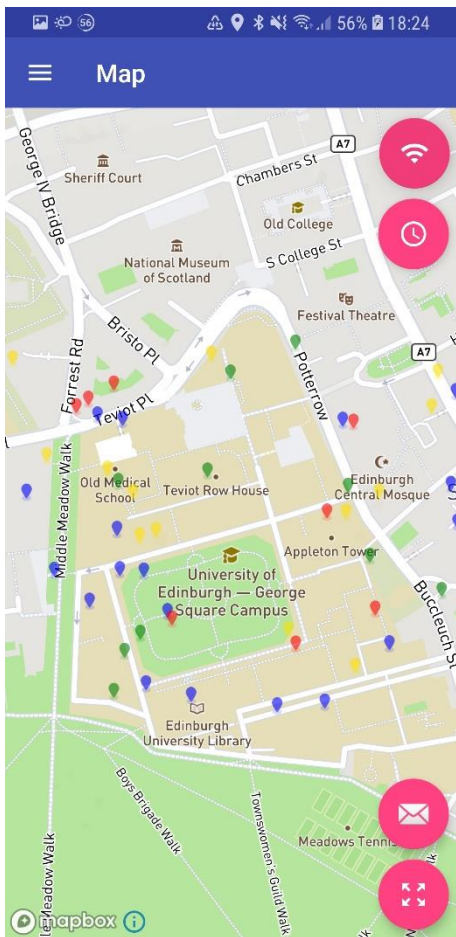


Figure 8 (left) – The view of the map testers see on log in. Note that there are 3 extra FABs, the one with the wifi symbol simulates the internet being turn on/off when pressed, and the one with the clock simulates the day changing to the next when pressed, and the one with the mail symbol simulates collecting a coin when pressed

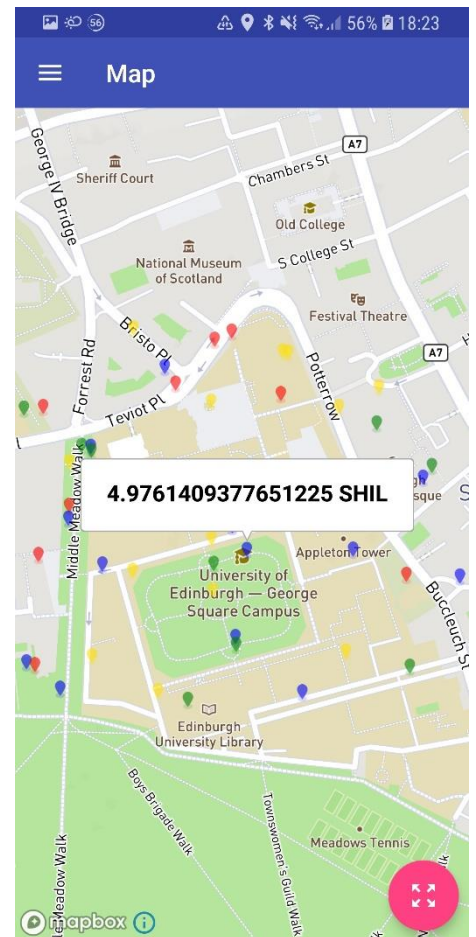


Figure 9 (right) – You can click on any marker on the map, and the value of that marker and currency are displayed

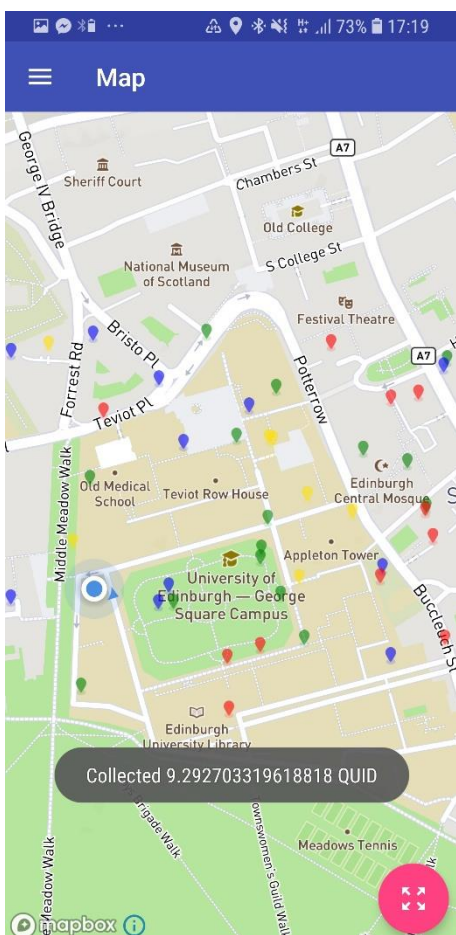


Figure 10 – When a coin is collected, a toast is shown showing what currency was collected and how much

Navigation

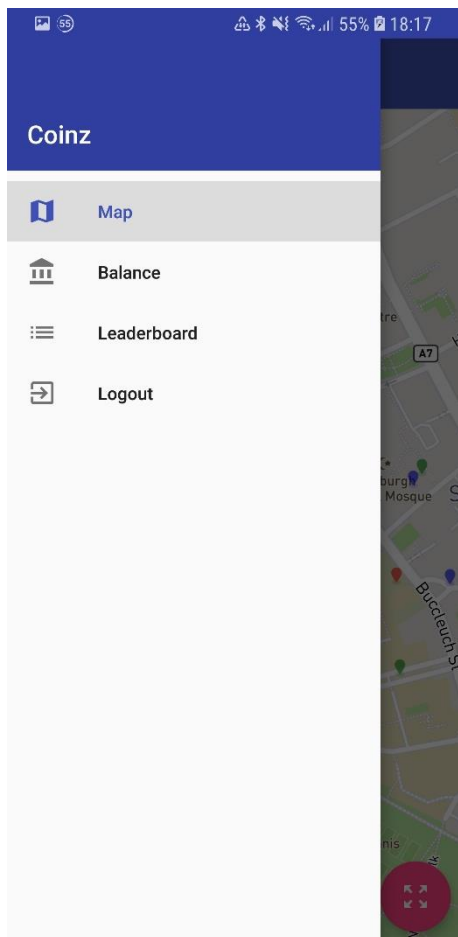


Figure 12 – The navigation drawer can be opened on any screen through clicking the button at the top left, and allows navigation between screens

Balance

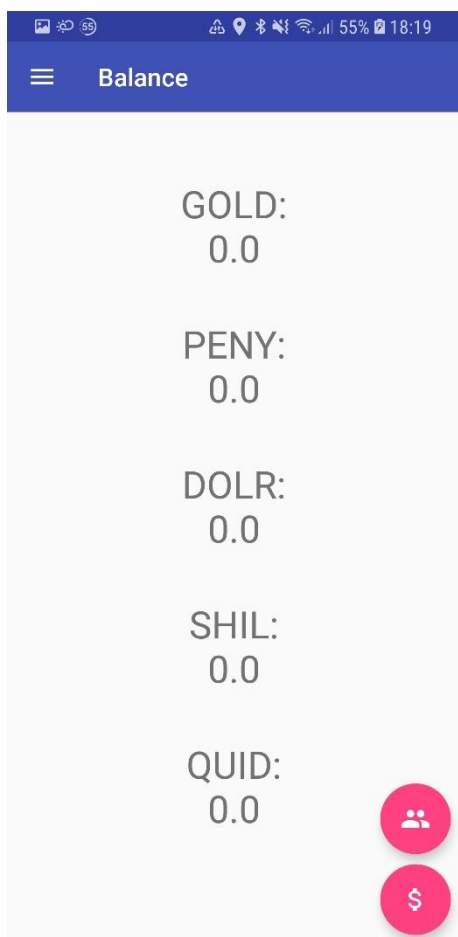


Figure 13 – The balance shows the amount of each coin the user has and will update automatically whenever a gift is received or sent, or an exchange is made, even when the balance is being displayed. From here you can open the exchange by pressing the FAB with the dollar symbol and can reach the gift sending screen by pressing the FAB with the people symbol.

Exchange

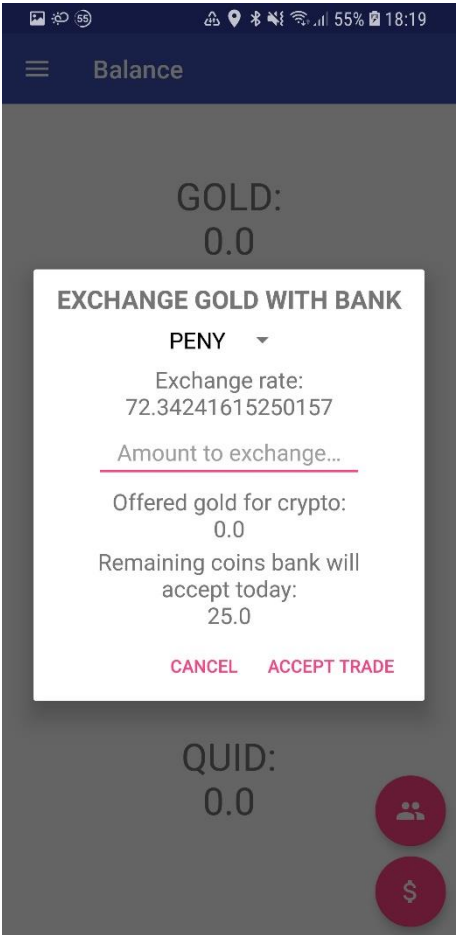


Figure 14 – This dialog fragment is displayed when the exchange FAB is clicked. The figure shows the default screen for exchanging with the bank.

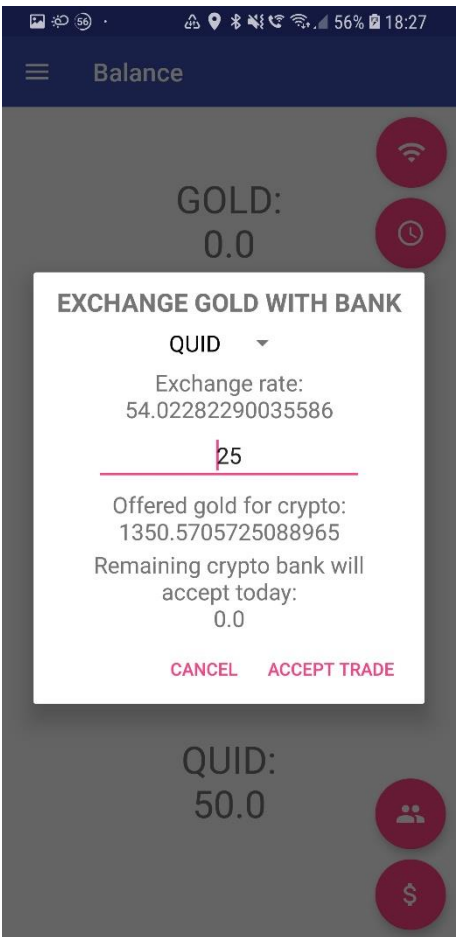


Figure 15 (left) – Shows a filled in exchange form. (Note that the two tester FABs are visible here at the top right as these screenshots were taken in the espresso tests, where a tester profile was used)

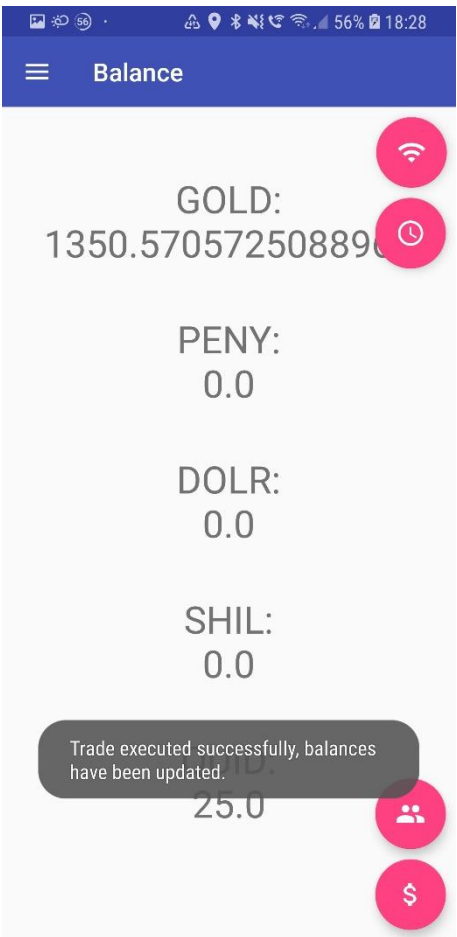


Figure 16 (right) – Shows the result of clicking accept trade. The value for GOLD has been increased by the amount the bank offered, the value for QUID has been decreased by the amount the user offered, and a toast is shown to inform the user the trade was executed successfully.

Creating a Username

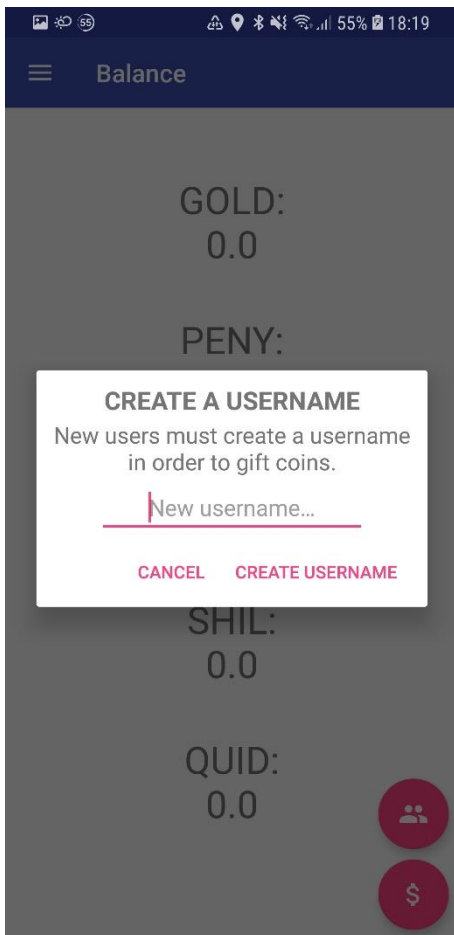


Figure 17 (left) – If the user clicks the gift FAB but they have not created a username, the following dialog will be shown, prompting them to create a username.

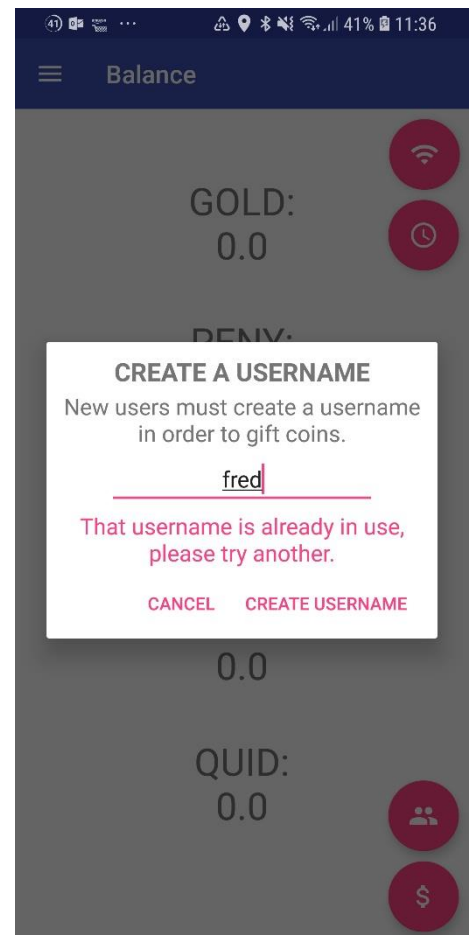


Figure 18 (right) – If the user enters their desired username, but it's already in use, then the username will not be set, and a warning will be returned informing the user the username is in use.

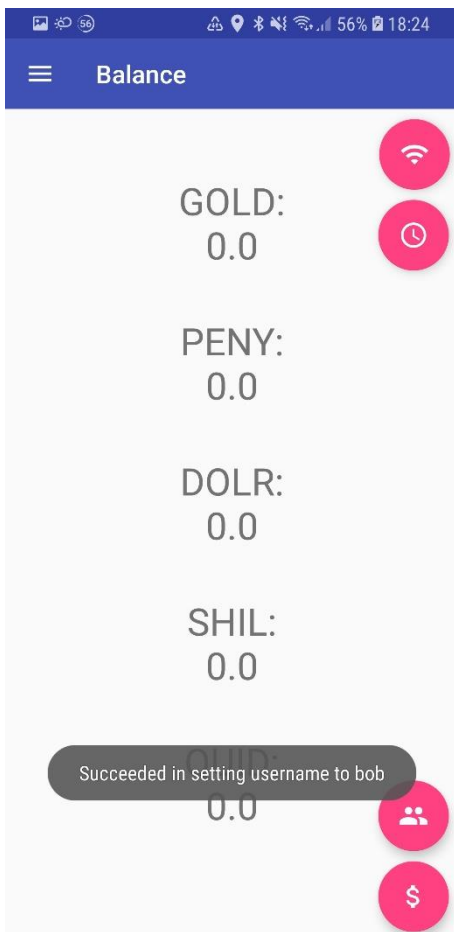


Figure 19 – If the user enters a username not in use into the form, and clicks the create username button, the dialog will disappear, and shortly return a toast to confirm their username was changed successfully.

Gifts

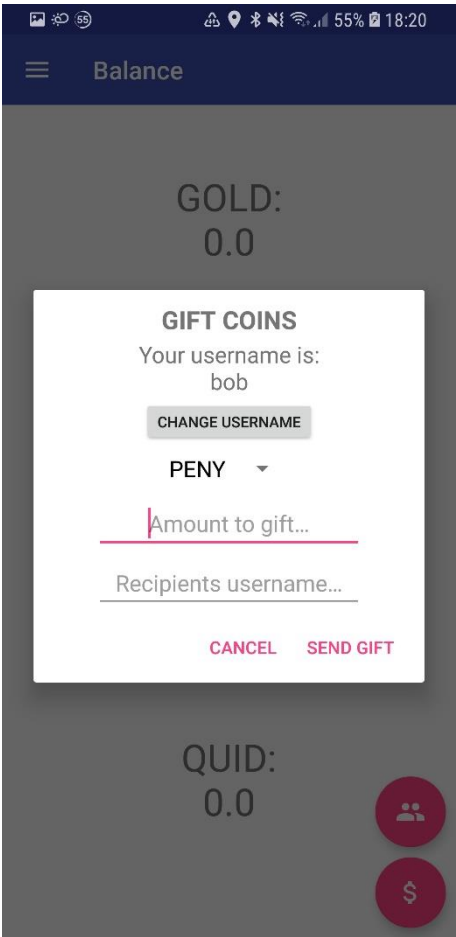


Figure 20 (left) – If the user clicks the gift FAB and have already set a username then they will be presented with the following view of the default gift form.

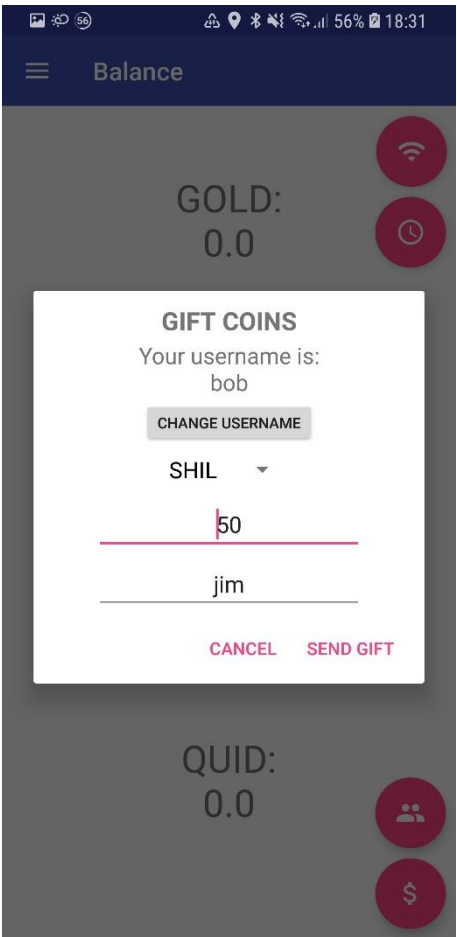


Figure 21 (right) – Shows a filled-out gift form.

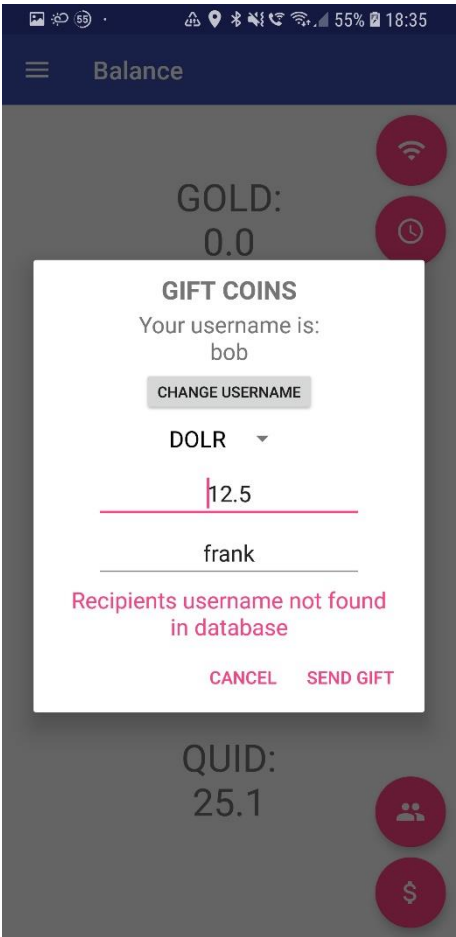


Figure 22 (left) – If the user enters a recipient who does not exist and then clicks “Send gift”, then the gift is not sent, and a warning appears.

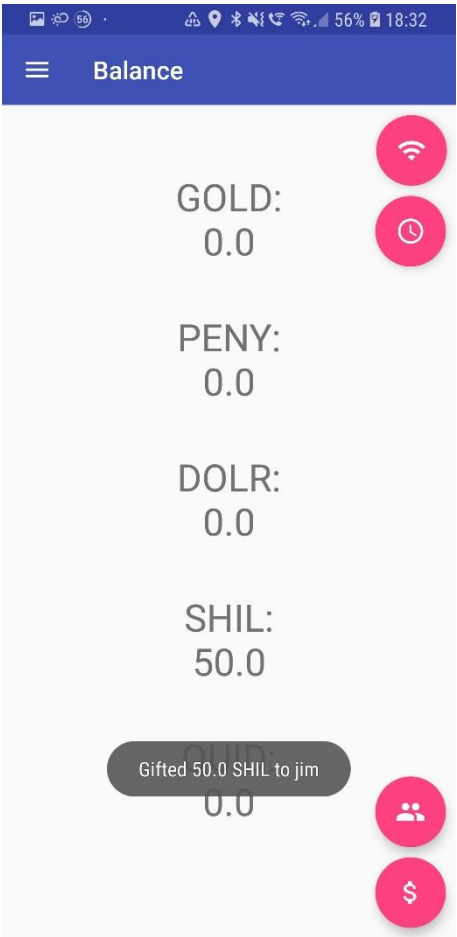


Figure 23 (right) – If the user enters a valid recipient for the gift, and clicks send, then the dialog disappears, and shortly after a toast informs them the gift was sent successfully. At this time the crypto they gifted is subtracted from their balance (note in this case we had 100 SHIL before sending the gift)

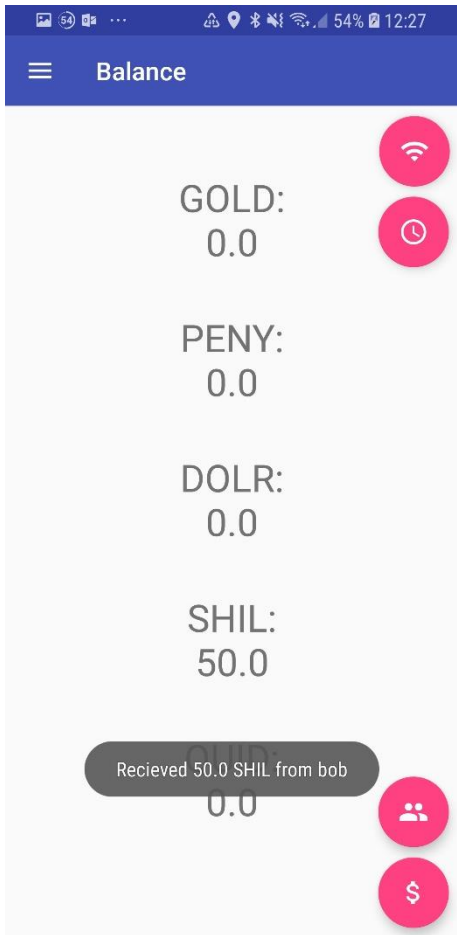


Figure 24 – Shortly after the toast informs the sender the gift has been sent, a toast appears on the recipient’s device showing them they have received a gift. If they are not using the app when they receive the gift, the notification is shown on login, and if they’re not connected to the app when they receive the gift, they’re shown the notification when they reconnect. When they are notified of the gift their local balances are updated to reflect receiving the gift.

Change username

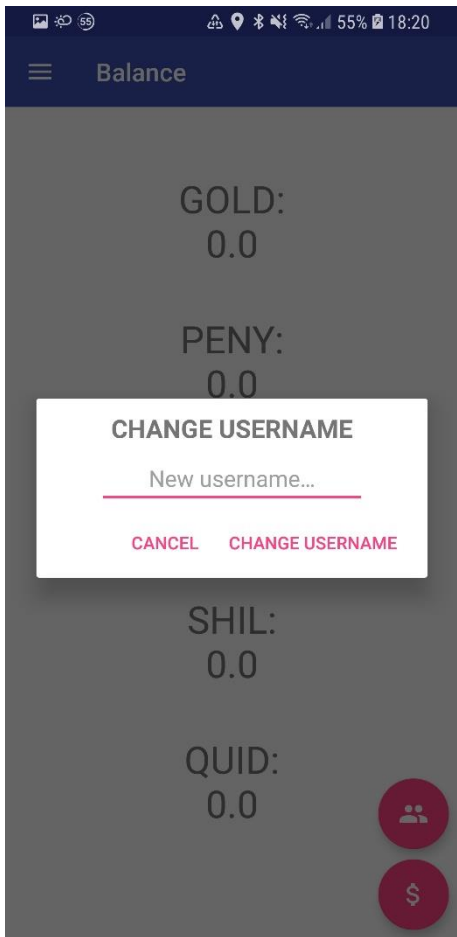


Figure 25 (left) – If the user clicks the change username button on the gift dialog, then the following dialog is displayed. This behaves the same as the create username dialog.

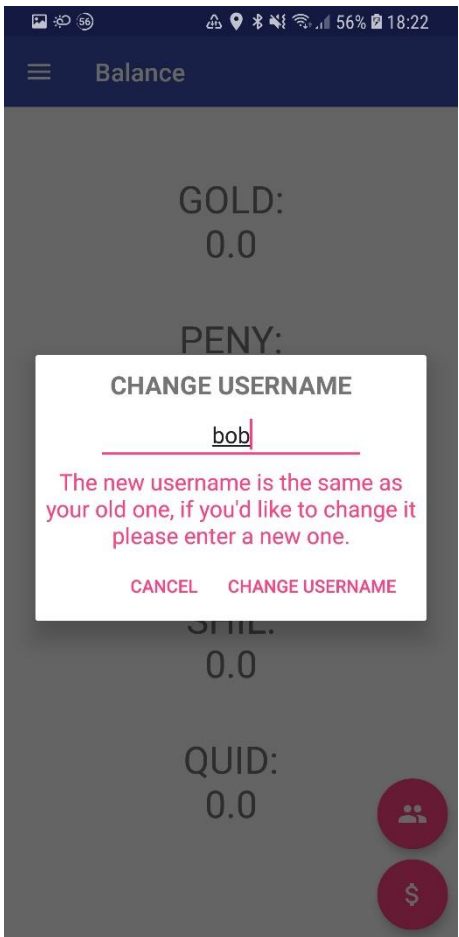


Figure 26 (right) – In addition to warning user when they try to change their username to one already in use, we warn them if their username is the same as their old one.

Leaderboard

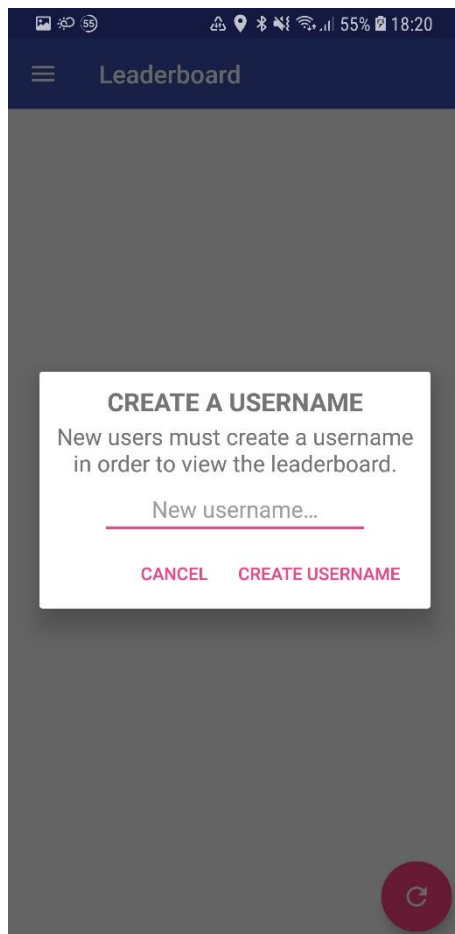


Figure 27 (left) – When the user opens the leaderboard, if they have not set a username, they will be requested to set one, displaying the create username dialog, and the only thing displayed will be the leaderboard refresh button. If they click the refresh button without having set a username, the same dialog will appear, and nothing else will be displayed.

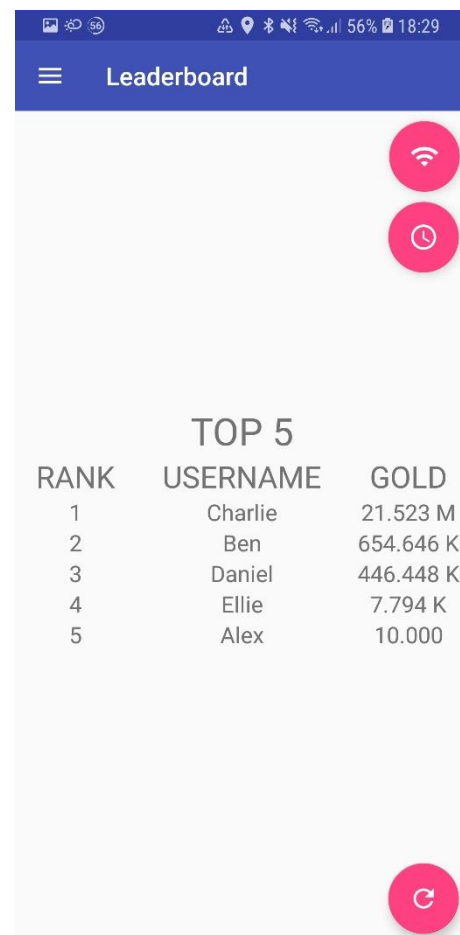


Figure 28 (right) – The leaderboard displays the amount of gold each user has collected, abbreviating million, billion, and thousand, as M, B, and K respectively, all amounts are rounded to 3 decimal places, this is done to ensure balances fit on the leaderboard. If there are less than 10 people in the database with a username set, then the top “n” are displayed, with the title updating to reflect how many are displayed.

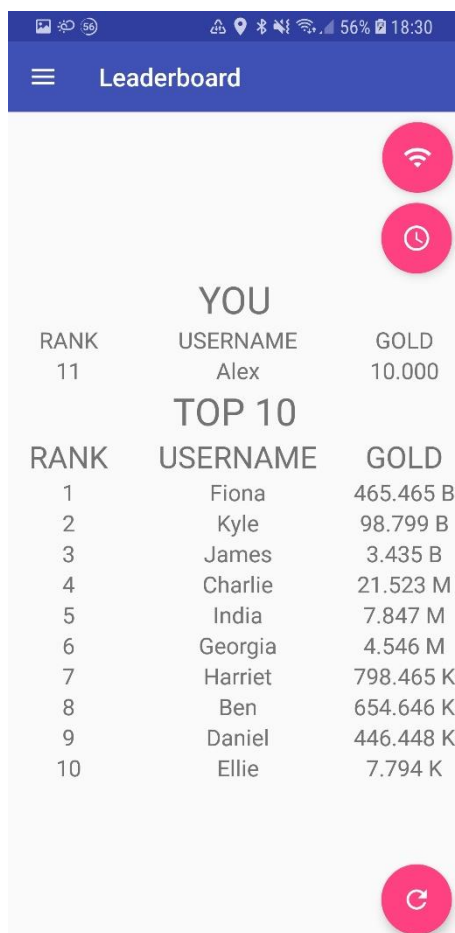


Figure 29 – If the user is outside of the top 10, then their rank in the leaderboard is shown above the top 10.

Offline Play

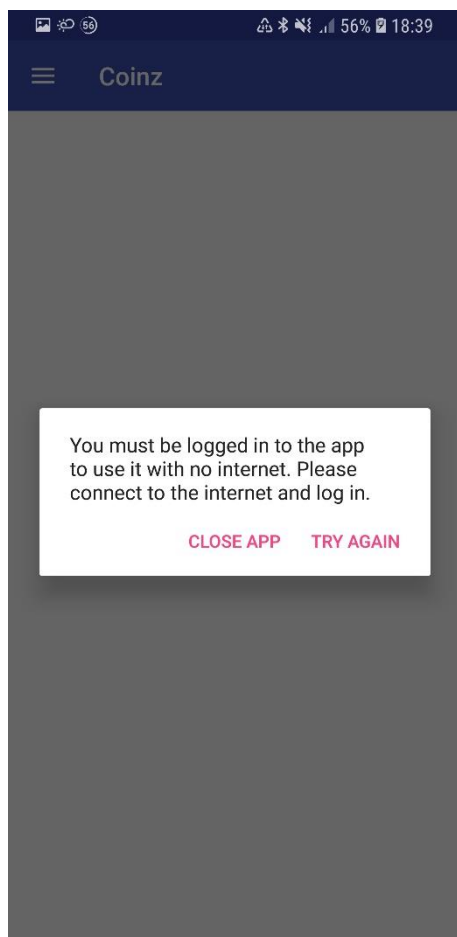


Figure 30 (left) – If the user is not logged into the app when offline, then when the method is called to log them in. The following dialog appears informing them they must be logged in to use the app, but can't log in as they're offline. This dialog can't be dismissed by clicking outside of it. The only way to get rid of it is to click "Close app", which closes the app, or "Try again", which will try and sign them in again, and if there is still no internet the dialog will reappear.

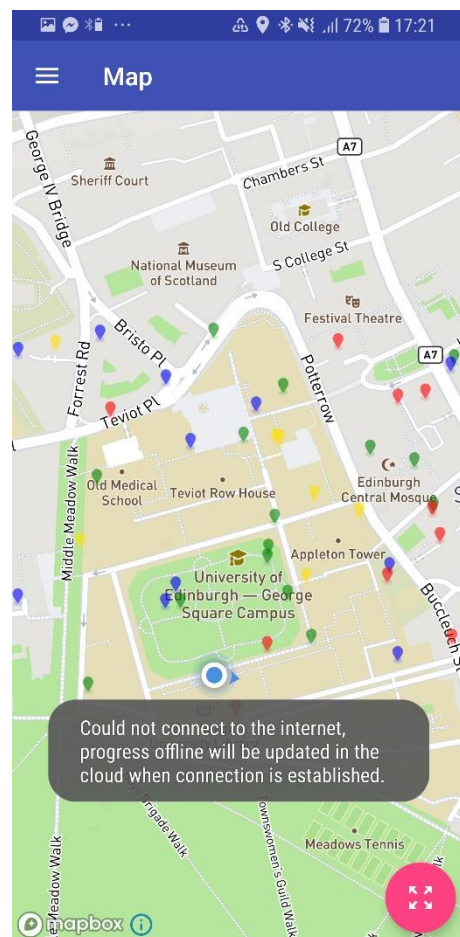


Figure 31 (right) – The first time a coin is collected offline a toast will be displayed to inform the user, followed by the toast to inform the user how much they collected and the specified currency.

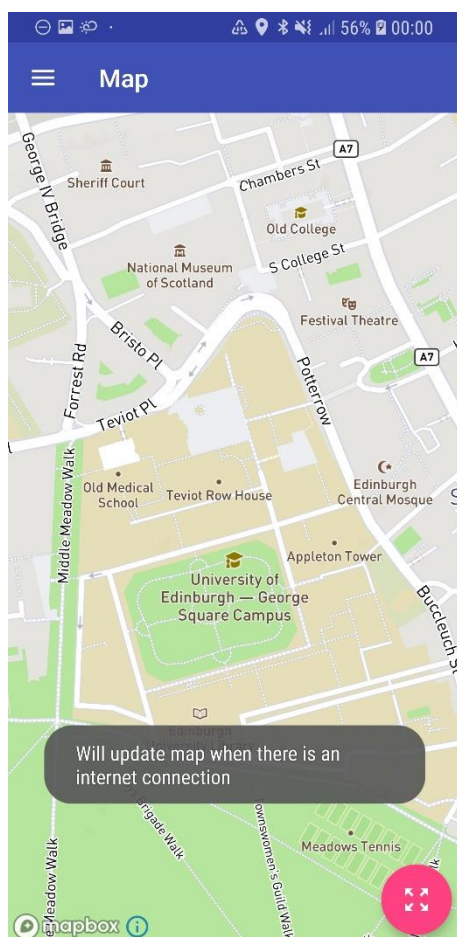


Figure 32 (left) – If the day changes while the user is offline, then the map is cleared of markers and a toast is displayed to the user informing them it will be updated when they next connect to the internet. If they're not viewing the map, then this happens the next time they open it.

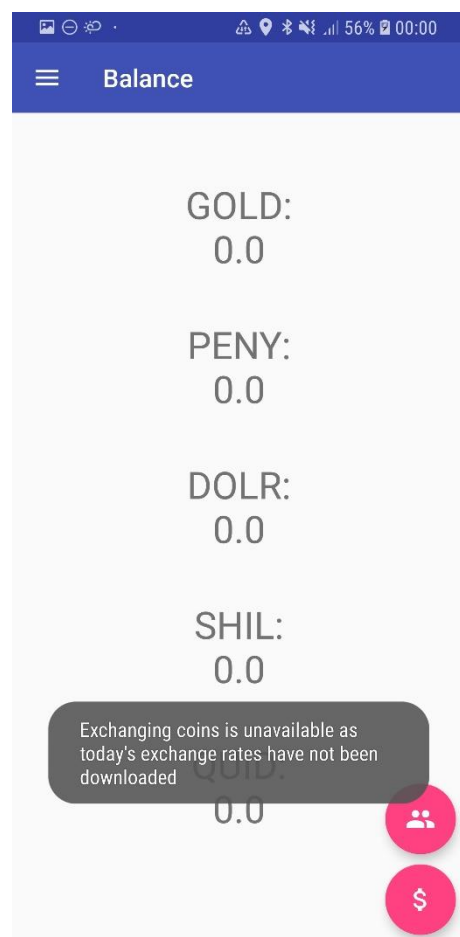


Figure 33 (right) – If the user tries to open the exchange after the day has changed but the new map has not been downloaded, then it is not opened, and a toast appears explaining that the currencies have not been downloaded for today (as they are attached to the JSON which hasn't been downloaded). Note that if the day changes but the user hasn't opened the map yet, the user can still trade on yesterday's exchange rates. This should be simple to correct, but it was not a critical bug and I didn't have time to fix it, so I left this bug in the submission.



Figure 34 (left) – If the user opens the exchange whilst offline, but the currency exchanges are downloaded, then the exchange will open as normal. But after clicking “Confirm trade”, they will be presented with the following toast, informing them their progress will be updated when they next connect to the internet.

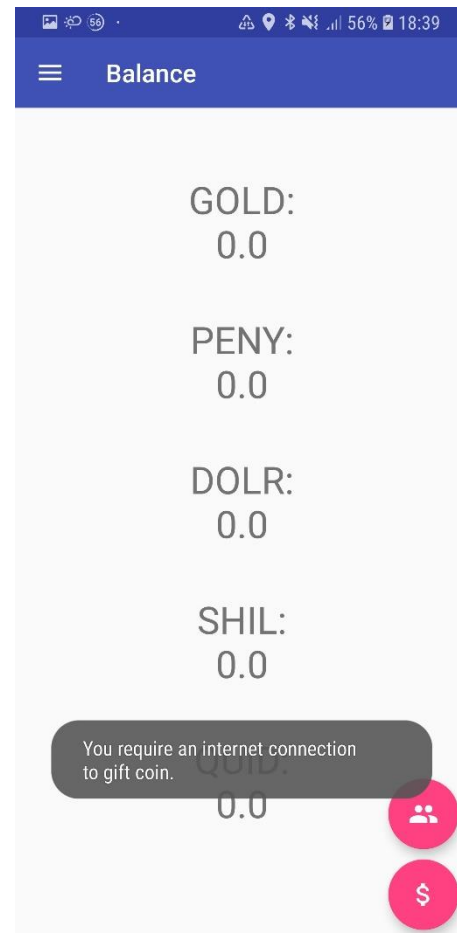


Figure 35 (right) – If the user tries to open the gift dialog when offline, then a toast will appear informing them they can’t gift coins while offline. We enforce this as we have no way to validate a recipient username is valid whilst offline.

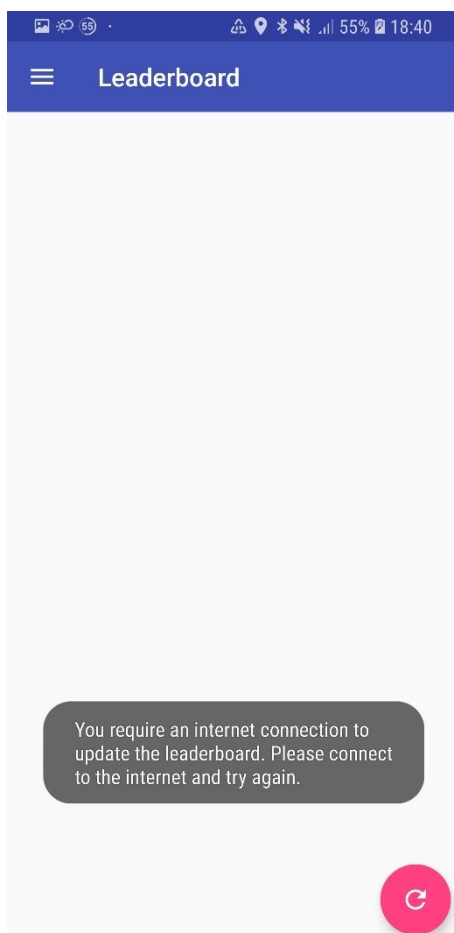


Figure 36 (left) – If the user opens the leaderboard or click the refresh button on the leaderboard whilst offline, they will be presented with an empty leaderboard, and a toast will inform them they must be online to update the leaderboard.

Acknowledgements

As I was working, I recorded this table of acknowledgements, along with the commit id of the point I first submitted code related to the source. In addition to those I listed, I made use of Mapbox tutorials for setting up the map and how to determine and display the user's location. I also made use of the documentation of Android, for things like creating alert dialogs.

Note that you can use the commit ID to view the code committed in that commit, allowing you to see how I used the source, to do this you can go to the following link, replacing in it "commit_id" with the commit you want to view.

https://github.com/KieranLitschel/Coinz/commit/commit_id

Commit ID	What	Source
2a1786f9f3b1c7af93092e321cae47ec4e5175f	How to add map to main activity	https://www.mapbox.com/install/android/
2a1786f9f3b1c7af93092e321cae47ec4e5175f	Added line "xmlns:mapbox= http://schemas.android.com/apk/res-auto " to activity_main.xml to fix unbound prefix error	https://stackoverflow.com/questions/25986242/error-parsing-xml-unbound-prefix-for-mapbox
cb4ce3e72cf55487132b5c6146564dd9eeb83b77	Used to fix render bug in android studio	https://stackoverflow.com/questions/49292487/failed-to-find-style-coordinatorlayoutstyle-in-current-theme/50851755
b2a675515afa57196c26c9cd23e5217b83140884	Display users location	https://www.mapbox.com/help/android-navigation-sdk/#display-user-location
c3020c867b88769059ae8d063518430b1780ec79	Input stream to String	https://stackoverflow.com/questions/9856195/how-to-read-an-http-input-stream
64c26838235109eec500e38e6c21c689905e73fc	Shared preferences listener (later removed from the implementation)	https://stackoverflow.com/questions/4997907/how-to-detect-if-changes-were-made-in-the-preferences
1fa2fc1380fec759c382999f370fac07c6d660bd	Marker template (first answer), "drawableTolcon" method (third answer)	https://stackoverflow.com/questions/37805379/mapbox-for-android-changing-color-of-a-markers-icon
e98dbf0077e703b5c2907be7379652ecce342dd5	Timers for events	https://stackoverflow.com/questions/10029831/how-do-you-use-a-timertask-to-run-a-thread
2602dfa7b3c3c0d3bb5f2f78a19fc10825e2c428	Method for detecting where there is an internet connection	https://stackoverflow.com/questions/4238921/detect-whether-there-is-an-internet-connection-available-on-android
24553ca0a5aea8fc3ab2dcc661159d50728a5dfb	DistanceCalculator class	https://www.geodatasource.com/developers/java
24553ca0a5aea8fc3ab2dcc661159d50728a5dfb	Sample code for LocationListener	https://stackoverflow.com/questions/42218419/how-do-i-implement-the-locationlistener
24553ca0a5aea8fc3ab2dcc661159d50728a5dfb	Sample code for how to do something when DownloadComplete	https://stackoverflow.com/questions/48212156/how-to-return-a-value-after-onsuccess-is-finished?noredirect=1&lq=1
a594de16b2cd601e0e688b64c375b54706236968	Used code as base for how to implement navigation drawer with fragments	https://guides.codepath.com/android/fragment-navigation-drawer
84ecd6ca32f84072f272466ecac89ac428624d33	Used to find how to run a toast on a separate thread	https://stackoverflow.com/questions/3134683/android-toast-in-a-thread

a2c33f90b61057e9b067645f8c796e9b05f991fa	Used idea suggested to make it so dialog didn't disappear when button pressed	https://stackoverflow.com/questions/2620444/how-to-prevent-a-dialog-from-closing-when-a-button-is-clicked
09f8721dca61f9a79d466239ed4a25b5ced7d5e2	Used 3 rd answer as inspiration for keeping track of losing and regaining internet	https://stackoverflow.com/questions/25678216/android-internet-connectivity-change-listener
92afdca7eaacab241d0aca21c25cdccbcd4691d6	Used answer for how to select item from spinner in espresso unit test	https://stackoverflow.com/questions/38920141/runtimeexception-in-android-espresso-when-selecting-spinner-in-dialog
24ef453071c652b7d60dc33b1f670a8ebffb4b42	Used answer for how to clear shared preferences in unit test	https://stackoverflow.com/questions/17791191/android-unit-test-how-to-clear-sharedpreferences