Introduction

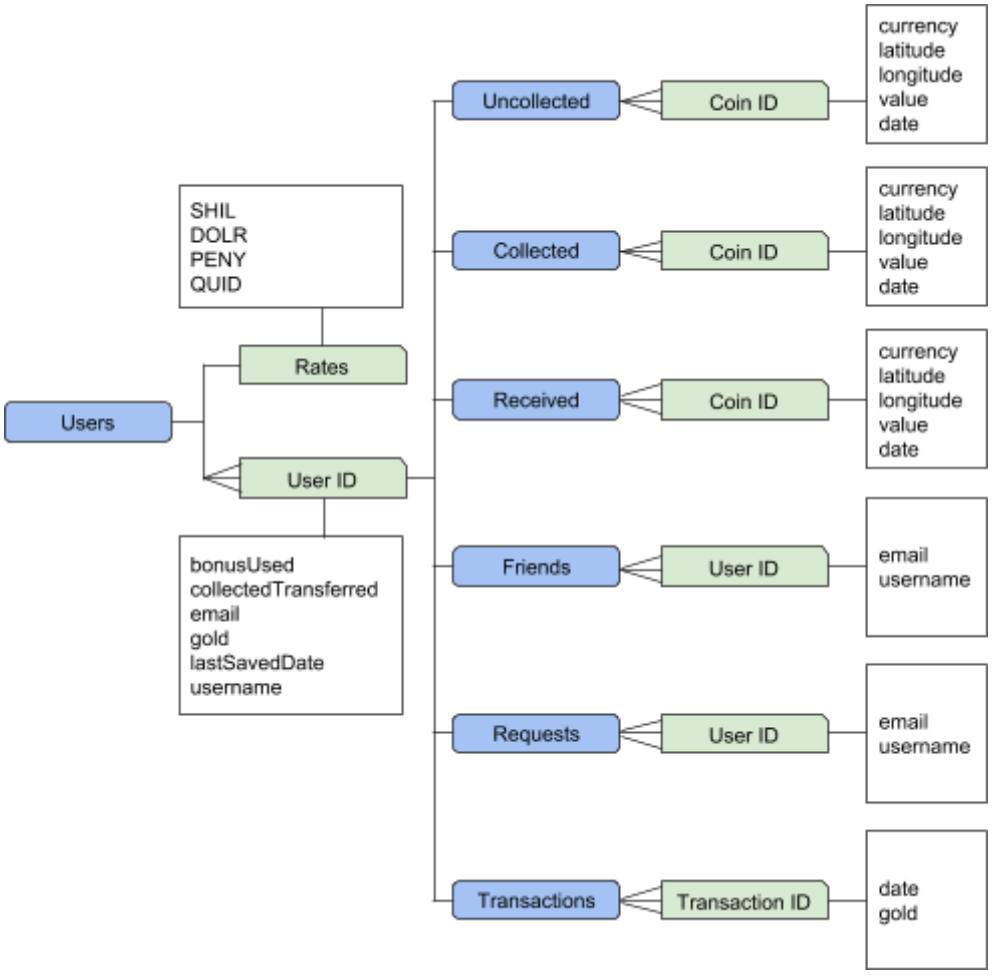
This report explains the implementation and design rationale of the Coinz application. The core functionality is examined, including the algorithms and data structures used throughout the software and the persistent storage of user data. Methods of robust testing for the application are also explored, along with improvements that could be made by a future team of developers. This application was developed using Android Studio with a Nexus 5X emulator running API 26. All classes in the codebase pass the IntelliJ code analysis.

JavaDoc documentation for the application can be found in Coinz/docs/.

Core Functionality

Data Storage

The application uses Firebase Cloud Firestore to store user data. Users are stored under the top-level collection “users”, where each user is stored as a document. The user document ID’s are the unique authentication ID’s which are created when a user registers an account with the application. Each user document contains several subcollections, which are shown in the diagram below.



The diagram above illustrates the structure of the database. The collections are shown in blue, and the documents in green. Relationships between collections and documents are depicted by the number of joining lines; a single line is a one-to-one relationship, and a triple line is a one-to-many relationship. The fields stored in documents are shown in the white boxes. The table below describes what each of the subcollections are used for.

Collection	**Description**
Uncollected	Coins the user has not yet collected on a given day
Collected	Coins the user has collected on a given day
Received	Coins the user has received from friends on a given day
Friends	The users friends list
Requests	The users friend requests
Transactions	A history of transactions the user has made to their bank account

The users collection also stores the exchange rates for each coin with respect to GOLD for a given day under the “rates” document. Each user document contains the fields detailed in the table below.

Field	**Type**	**Description**
bonusUsed	Boolean	Indicates if the user has used their daily bonus for the given day
collectedTransferred	Integer	Number of coins the user has transferred from their collected section to their bank account on a given day (used to enforce the 25 coin daily bank transfer limit for collected coins)
email	String	Users email address
gold	Double	Amount of Gold coins user has
lastSavedDate	String	Last date the user has retrieved the latest map. Used to download new map if a new day has begun. If current date is not equal to lastSavedDate, then retrieve new map data from Informatics server. Otherwise, retrieve all data from firebase.
username	String	Users username

To manage all of this user data, I created a Data class consisting of all static methods and fields. The benefit of this is that no two activities or fragments need to communicate directly with each other to share data, rather they refer to a single point of access to store and retrieve data. Furthermore, the Data class handles all uploading and downloading to Firebase. This abstraction of the backend functionality means all activities and fragments are much cleaner and easier to read, making them easier to debug and maintain.

This design is somewhat similar to the Singleton design pattern, however I chose to create a class with static methods as the Data class is not maintaining any state, it is just providing global access to methods. Furthermore, static methods are much faster than a Singleton class because of static binding during compile time[1]. However, perhaps a more suitable design choice would be to use a Singleton class as it is more object-oriented, which may make the backend easier to unit test.

The application uses the objects User, Coin and Transaction to store data about entities. The Uncollected, Collected and Received collections are stored in the Data class as ArrayLists of Coin objects. The Friends and Requests collections are stored as ArrayLists of User objects, and the Transactions subcollection is stored as an ArrayList of Transaction objects.

ArrayLists were a good choice for these collections due to their dynamic storage. However, perhaps a more suitable data structure for the coin collections would have been a HashMap or HashSet. This is because there is no need to preserve ordering in the coins, and checking if a coin exists in the data structure by ID would take $O(1)$ time rather than $O(n)$.

The Friends, Requests and Transactions collections do require an ordering i.e. friends and friend requests are sorted alphabetically by username and transactions are sorted by date. For the current scale of the application, sorting these values using the Java List.sort() is sufficient as it takes $O(n\ log(n))$ time.

A potential performance improvement would be to use a TreeMap or TreeSet for these objects, as they would be sorted upon insertion. These data structures would need to be converted to Lists to be used with Recycler Adapters for the UI, however this conversion takes $O(n)$ time, which is an overall performance improvement. Furthermore, checking the existence of a friend, friend request or transaction in a TreeMap or TreeSet is guaranteed $O(log(n))$ time, rather than $O(n)$ time for an ArrayList. However, adding an element to a TreeSet or TreeMap would take $O(log(n))$ time rather than constant time with an ArrayList.

The exchange rates are stored in a HashMap as (String, Double) pairs which represent the currency and its current exchange rate with GOLD. This is a suitable data structure as no ordering is required and retrieving an exchange rate takes constant time.

The performance improvements mentioned above could be easily implemented by a future team of developers due to the design of the software. The majority of the changes required would only be in the Data class, and the activities and fragments that use the data would only require minimal changes.

The application updates the data in firebase as soon as any changes occur, for example when collecting a coin or adding a friend. In addition, all data is retrieved directly from Firebase upon bootup of the application (this is discussed in the Boot Up Sequence section), rather than through Androids Shared Preferences. The benefit of this is that all user data is constantly being backed up, ensuring that the user will never lose any data (for example their collected coins) if they decide to log in to another device.

The downside, of course, is that the application always requires an internet connection to function properly. A potential improvement here would be to make use of both Firebase and Shared Preferences, so that if the user loses internet connection, their data will at least be backed up locally on the device they are using.

The backend methods for the Data class are described in the JavaDoc documentation. All of the backend methods take the interface OnEventListener as an argument. This interface contains the methods onSuccess() and onFailure(). Once the asynchronous task for the backend method finishes, one of the methods are invoked (dependent on a success or failure) which is implemented by the caller of the backend method.

This design is consistent throughout the application, and means that the UI never has to wait on an asynchronous task to finish before it continues with inflating its view. Once the asynchronous task completes, the UI can then be updated with the most up to date data. The result is a fast loading UI, creating pleasant user experience.

All of these backend methods within the data class could be extensively tested using Firebase Test Lab. These tests could be combined with instrumentation tests to extensively validate the full infrastructure of the software. For example, an instrumentation test could run through the full cycle of two users becoming friends on the application. This would involve one user sending a friend request via email and the other user accepting the friend request, and Firebase Test Lab could verify every step along the way.

Registration and Login

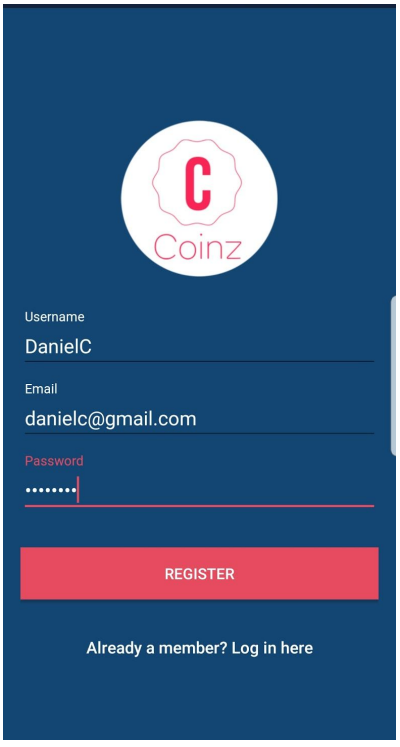
To create the registration and login functionality, I followed a tutorial from Android Hive[2], which details how to register a user with an email and password using Firebase Authentication. I also included a username field for the users, as I feel this is more appealing on a friends list or leader board than an email address. The registration process works as follows:

1. User enters username, email address and password
2. Validate the entered details according to the following criteria:
    a. No fields are empty
    b. Password length is at least 6 characters
3. Verify that the username entered does not already exist (Firebase Authentication verifies that the email address does not already exist)
4. Create user account with email and password using Firebase Authentication
5. Add user data to database with the fields described on page 2

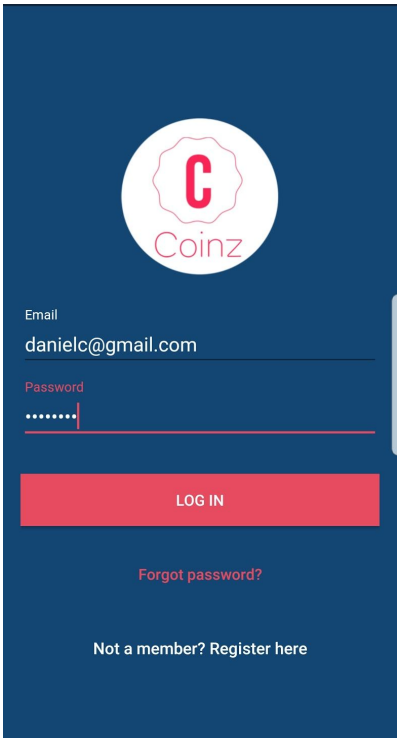
To verify that the username does not already exist, the application retrieves all existing user documents from the database and compares the entered username against the existing usernames. I realise this process may be slow if the application was ever scaled to be used by a large number of users. A more efficient method would be to have a single document that stores all existing usernames, and then the process would only involve retrieving one document rather than all user documents. However, it is also a one off process to create a user account, so it would not affect the day to day use of the application.

The login activity requires that the user enters their email and password, which will be verified by Firebase Authentication. The user will remain logged in on their device so long as they don't log out, therefore they will not have to enter their login details every time they open the application.

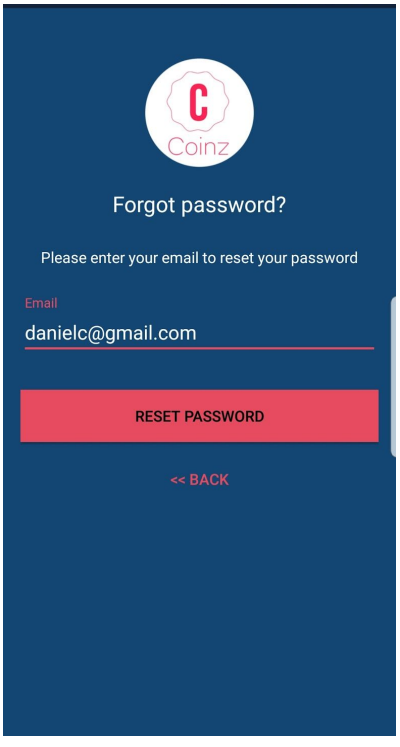
The application also allows the user to reset their password, which is handled by Firebase Authentication. The screenshots below show a typical registration, login and password reset with the application. Users can log out by selecting the log out option from the overflow menu on the toolbar. Automated tests of this process can be found under the directory mullan.sean.coinz (androidTest).



Registration



Login



Reset Password

Boot Up Sequence

The entry point to the main application (following log in or registration) is the MainActivity. The application uses Bottom Navigation View. Therefore, I used fragments for each of the application sections (friends, wallet, bank etc) which are hosted by the MainActivity, and the MainActivity simply loads the correct fragment when an item is selected on the Bottom Navigation View. This removes the need to have every application section implement the Bottom Navigation View and Action Bar. I followed an Android Hive tutorial as a guideline.[3]

When the MainActivity is created, it begins the boot up sequence for the application. This sequence ensures that the application retrieves the correct data for the user. The boot up sequence begins as follows:

1. Fetch the user document from Firebase using the User ID
2. Set the users value of GOLD in the Data class
3. If lastSavedDate != currentDate (i.e. a new day has begun), then update the following fields in Data and Firebase:
    a. Set bonus used to false
    b. Set collected transferred to 0
    c. Set the new lastSavedDate to the current date
4. If lastSavedDate == currentDate, then initialise the values of bonusUsed and collectedTransferred in the Data class with the current values in the user document
5. Load Map Fragment
6. Invoke populateData()

The populateData() method executes the process of retrieving data to populate the users collections in the Data class. Whether or not a new day has begun, the application needs to fetch all existing data from the users subcollections.

If a new day has not begun, then the Data class is populated with all the existing data i.e. the users Uncollected, Collected and Received coins, Friends, Requests, Transactions, and the Exchange Rates. When all of the Uncollected coins have been retrieved, the Map Fragment is prompted to update its map with the new uncollected coins data.

If a new day has begun, then the retrieved data is used to identify which documents need to be deleted in Firebase (i.e. for deleting old map data and clearing the users local wallet). If this is the case, then the following steps are executed:

1. Clear uncollected coins so that they can be replaced with new coins
2. Clear 'spare change' i.e. collected coins and received coins (check the date that each coin was received from a friend, and keep the coin if it was received on the present day, otherwise delete it)
3. Retrieve the new map data from the Informatics server
4. Parse the received data
5. Update the Data class fields and firebase with the new data
6. Prompt the Map Fragment to update its map with the new data

To new map data is retrieved by sending a HTTP GET request to the following informatics server: http://homepages.inf.ed.ac.uk/stg/coinz/. The date is appended to the URL to retrieve the map data for the current day. The server returns the exchange rates and map data as a JSON file.

The exchange rate data is extracted from the JSON and a HashMap is created which is used to set the exchange rates in the data class. Each of the coins are then processed individually, by creating a Coin object from the extracted data and passing it to the Data class for processing. The Data class adds the coin to it's uncollected coins ArrayList and uploads the coin to the users Uncollected subcollection on Firebase. To ensure a fast loading of the coins on the map, the Map Fragment is prompted to update its data once all of the Coin objects have been created - it does not wait on all of the coins to be uploaded to Firebase.

The boot up sequence could also be tested extensively using Firebase Test Lab. It can be separated into two main cases: 1) A new day beginning or 2) Using the app on the same day. Both of these cases could then be further divided into sub cases to ensure that every individual field and document gets updated appropriately.

Map Rendering and Coin Collection

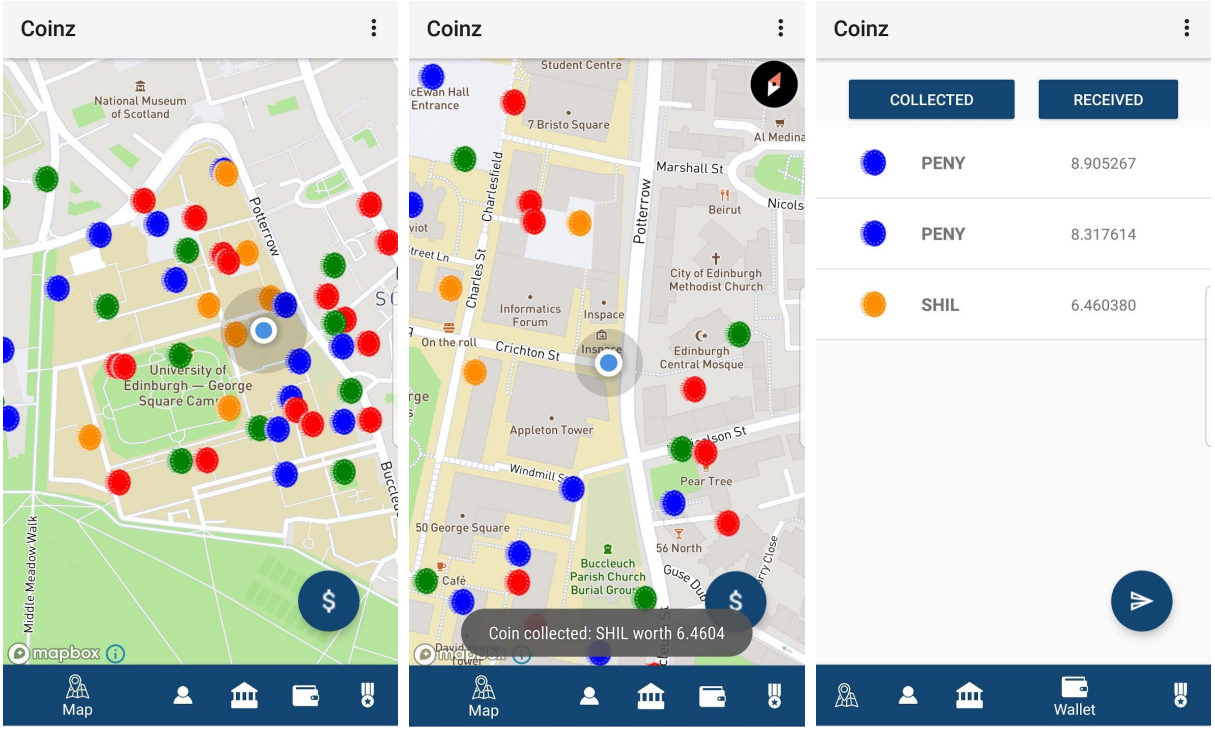
The map is rendered using the MapBox API, and the user's location is detected using the MapBox Location Engine. Coins are added as Markers on the map, which get updated any time the uncollected coin data changes. The MapFragment contains two data structures: the ArrayList of uncollected coins provided by the Data class, and a HashMap containing (Coin, Marker) key-value pairs. When a Marker object is created from a Coin, the Marker and Coin are added to the HashMap and the Marker is rendered on the MapBox map. An example of a rendered map is shown on page 8.

The Map Fragment overrides the onLocationChanged method, which is invoked by the Location Engine Listener. The Location Engine is configured to invoke this function at least every second, with a preferable interval of every 5 seconds. When the onLocationChanged function is invoked, the algorithm to detect if a coin should be collected begins.

This algorithm is a simple linear search over the GPS coordinates of the uncollected coins ArrayList, with the distance being calculated between the location of a coin and the user's current location. If the distance between the two sets of GPS coordinates is less than or equal to 25 metres, then the coin is collected. Clearly, this algorithm takes $O(n)$ time.

To collect a coin, the corresponding Marker is removed from the map using the coin as the key for the HashMap, and the coin is removed from the local uncollected coins ArrayList. Then, the Data class is prompted to remove the coin from the uncollected subcollection and upload the same coin to the collected subcollection.

The screenshots below show rendered map and the process of collecting a coin. Each coin currency is colour coded, with DOLR as green, PENY as blue, SHIL as orange and QUID as red. The user is purposely not shown the value of the coin before collecting it - this is to add an element of mystery to the game. When the user gets within 25 metres of the SHIL coin, it is removed from the map and placed in the users wallet. As shown in the screenshot on the right, the SHIL coin sits in the users Collected section of their wallet. The implementation of the wallet is described in the Wallet section.



This feature could be tested by placing the application into a testing mode so that it always downloads and renders a specific map. A gpx file could then be used to create a route for the location of the device to follow, and Firebase Test Lab could be used to verify that the coins which should be collected along the route are removed from the Uncollected subcollection and placed in the Collected subcollection.

I attempted to create an automated test for this feature, however I was unable to create an automated way of mocking the location of the device without having to manually input a gpx file. However, I did create a way to put the application into a test mode so that it will always download a specific map, and when running the gpx file I was able to manually verify that all the coins which should have been collected are successfully processed.

Friend Adding

The Friends Fragment is separated into friends and friend requests. The UI is achieved using two Recycler Views (one for each) and buttons at the top of the fragment to display the appropriate view. The views are loaded using the ArrayLists of Friends and Requests currently stored in the Data class. The lists are sorted alphabetically by usernames, using the Java List.sort() method which takes $O(n\ log(n))$ time as described in page 3.

Then, in the background, the Data class is prompted to update its Friends and Requests data. Once this process is complete, the Friends Fragment has its callback method invoked and it updates the UI with the most up to date data. This design ensures a fast initial loading of the UI, and then the most up to date lists are displayed once they have been retrieved from Firebase.

Users can add each other via email. The process of becoming friends with another user is described below:

1. Daniel sends Anna a friend request via her email address
2. Anna's document is located on Firebase using her email, and a document is placed in Anna's Requests subcollection. The document ID is Daniels User ID, and the fields contain Daniel's username and email.
3. Anna accepts the friend request from Daniel
4. Daniel is removed from Anna's Requests subcollection and placed in Anna's Friends subcollection
5. Anna is placed in Daniel's Friends subcollection

Now, when both users check their friends list, they will have each other as a friend. If Anna had declined the friend request, then Daniel would simply be removed from Anna's Requests subcollection. The following criteria must be met for a friend request to be sent:

- The user is not sending themself a friend request
- The user is not trying to add someone they already have as a friend
- The user is not trying to add an account which doesn't exist
- The user is not trying to add someone who has already sent them a friend request

In the case that a user attempts to send a friend request multiple times to another user before they have accepted or declined the request, it will still remain one request as Firebase does not allow duplicate document ID's.

A User object is created for each friend, which stores the friends ID, email and username. As mentioned on page 3, a potential improvement here would be to use a TreeMap or TreeSet to store the friends and friend requests as they would preserve ordering. I have created an automated test for the process of two users becoming friends, which can be located under mullan.sean.coinz (androidTest) as the 'addFriendTest'.

The storyboard on the following page shows two users becoming friends.


Coinz
FRIENDS
REQUESTS
seanmullan1997
seanmullan1997@gmail.com
Friends
1. Daniel sends Anna a friend request via email
Enter their email address
anna@gmail.com
CANCEL
ADD
2. Anna accepts the friend request from Daniel
DanielC
danielc@gmail.com
ACCEPT
DECLINE
3. Daniel is now in Anna's friends list
anna
anna@gmail.com
4. Anna is now in Daniel's friends list

Local Wallet and Bank Account

A user's local wallet is separated into coins they have collected themselves and coins which they have received from friends. The UI is designed and operates in a similar way to to the Friends Fragment. That is, the UI is inflated with the currently stored data, and the Data class is prompted to update its Collected and Received coins in the background. Once this process completes, the UI is updated with the most up to date data.

The local wallet provides two main functions to the user: sending spare change (collected or received coins) to a friend, or transferring spare change to the users bank account to be converted to GOLD. A user can only transfer a maximum of 25 collected coins to their bank account per day, but they can transfer an unlimited amount of received coins to their bank account per day. There is no limit on the number of coins the user can send to friends, and all spare change will get cleared at the end of the day.

Sending coins to a Friend

The user can select which coins they would like to send, either from the collected or received section of their local wallet. Once the user selects the coins, they are presented with a list of their friends and can select who they want to send the coins to. The CoinAdapter detects which coins have been selected by their position in the RecyclerView, and sets a flag in the corresponding Coin objects which mean that they have been selected.

When the user hits send, an ArrayList is created for the coins which have been selected. The Data class is then prompted to place each coin in the friends Received subcollection on Firebase, and each coin is removed from the appropriate subcollection in the user's document (either from Collected or Received). When placed in the Received subcollection, the Coin document is allocated a random ID rather than using the Coin ID. This is to prevent duplicate ID's in the case where two players send the same coin to another player. The current date is added to the coin data - this is so the boot up sequence can identify when it was received, and can delete the coin if it was not received on the present day.

To prevent two transfers occurring at the same time, a flag is set when a transfer begins, and cleared when the transfer ends. To detect when a transfer has completed, a variable is incremented after each coin has been processed on Firebase. Once this variable is equal to the size of the ArrayList of selected coins that had been created, the transfer is considered complete, and the transfer in progress flag is cleared.

Transferring coins to Bank Account

This works similarly to sending coins to a friend. The CoinAdapter detects which coins have been selected in the wallet, and an ArrayList of the selected coins is created. If the selected coins are from the Collected section, then the following equality is checked to ensure that the user does not exceed the 25 coin daily transfer limit:

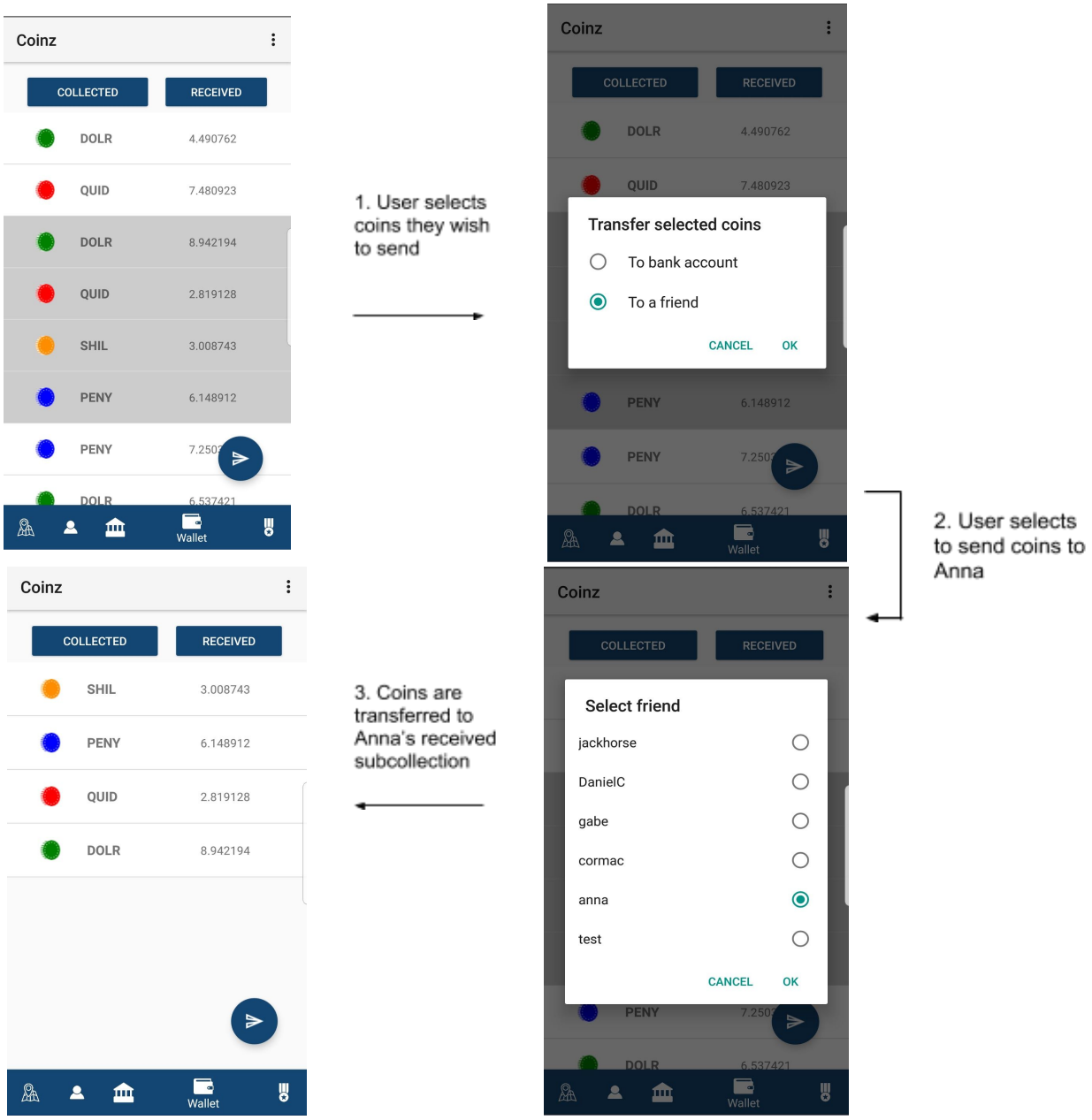
```
if (selectedCoins.size() > (25 - Data.getCollectedTransferred())
        // Prevent transfer
```

A bank transfer works similarly to a friend transfer in that a flag is set while the transfer is in progress and cleared when it finished, with a variable keeping track of how many coins have been processed on Firebase so far.

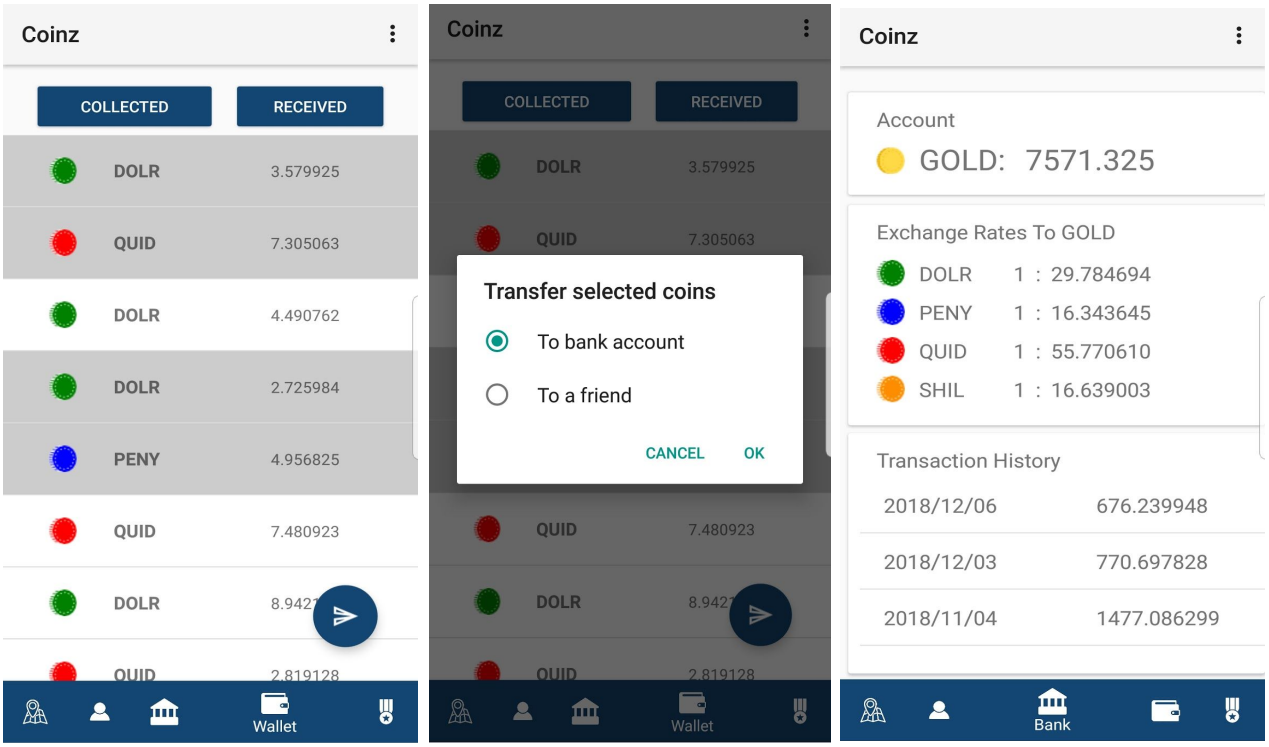
For each coin, the value of the coin is multiplied by the exchange rate for the coins currency to get the coins value in GOLD and the coin is removed from the appropriate subcollection on Firebase. These values of GOLD are summed up across the coins to calculate the total amount of GOLD all the selected coins are worth.

Once all coins have been processed, a Transaction object is created which contains the total value of GOLD that the coins were worth and the date of the transaction. The Transaction data is then uploaded to Firebase and the user's GOLD amount is increased by the transactions value. If the coins were transferred from the Collected section, then the quantity of these coins is added to the collectedTransferred variable.

Screenshots of sending coins to a Friend



Sending coins to Bank Account



As shown in the above screenshot, the users Bank Account displays their current amount of GOLD, the current days exchange rates and a history of their transactions sorted by date. It is clear that given the coins selected and the given exchange rates:

$$(3.6 * 29.8) + (7.3 * 55.8) + (2.7 * 29.8) + (5.0 * 16.4) = 677.1$$

Using the more accurate decimal place values we see that a transaction has been added with a value of 676.24 on 2018/12/06. As mentioned on page 3, a potential performance improvement here would be to use a TreeMap or TreeSet to store the Transaction objects as these data structures would preserve natural ordering.

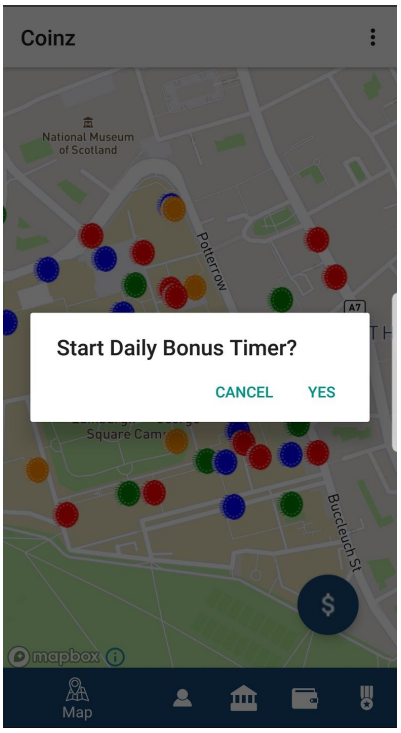
Both the types of transfer (to bank account or to a friend) could be tested using Firebase Test Lab to ensure that every step in a transfer is completed correctly.

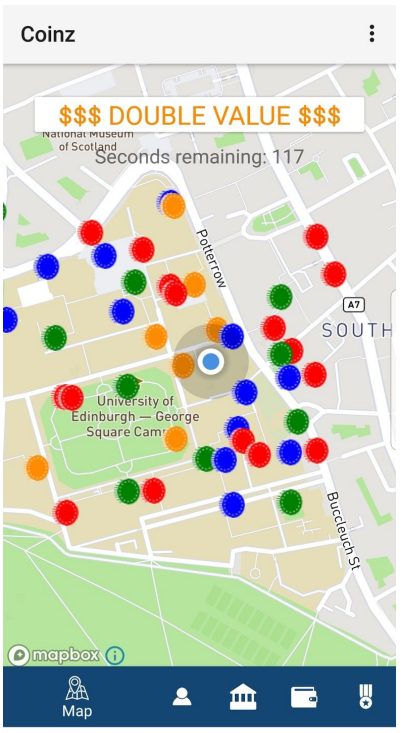
Bonus Features

Daily Bonus

The daily bonus feature is available to each player once a day, and allows players to double the value of the coins on the map for 120 seconds only. Once a coin is doubled in value, it retains this value when transferred to friends or to the users bank account.

A flag is used to indicate if the user has used their daily bonus or not (this gets set when they start the daily bonus timer, and cleared if a new day has begun). Another flag is used to indicate if the timer for the daily bonus is active. When a coin is collected, a check is made to see if the bonus is active, and if so the value of the coin is doubled.

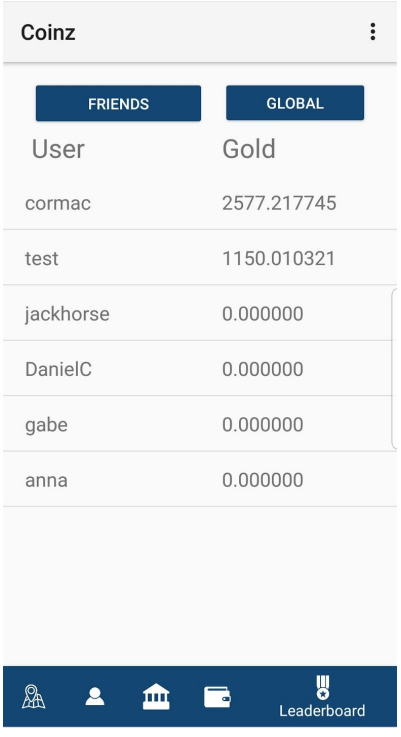




Leaderboard

The Leaderboard fragment operates similarly to the Wallet Fragment and the Friends Fragment by initially loading the UI with the current data, then updating in the background. Users can view a leaderboard of their friends, and also a global leaderboard.

To load the global leaderboard, all user documents must be retrieved to get their GOLD amounts, which is inefficient. A more efficient way to do this would be to store a single document that contains all usernames as keys and their corresponding GOLD amounts as values. Then, only one document has to be retrieved to fetch the global leaderboard.



All features described in the Project Plan have been successfully implemented. However, there are a number of performance improvements that can be made which I have discussed throughout this report. I have summarised the suggestions below:

- Change Uncollected, Collected and Received coin ArrayLists to HashMaps
- Change Friend, Requests and Transactions ArrayLists to TreeMaps
- Combine use of Firebase with Shared Preferences to ensure a local backup is made if the user loses internet connection
- Create a single document that stores all existing usernames
- Create a single document that stores every users GOLD amount

There were no major differences between the original design and the implementation, apart from the use of the subcollection “Uncollected”. Originally, I had designed the application so that uncollected coins would be stored in the users local storage. However, I realised that if the user logged on to another device, the uncollected coin data would be incorrect. Overall, the implementation closely followed the original design.

Acknowledgments

[1]
https://javarevisited.blogspot.com/2013/03/difference-between-singleton-pattern-vs-static-class-java.html

[2]
https://www.androidhive.info/2016/06/android-getting-started-firebase-simple-login-registration-auth/

[3]
https://www.androidhive.info/2017/12/android-working-with-bottom-navigation/