Trading Wheels Technical Documentation

An Incom Systems Application

Incom Systems Team

Zac Williams - S3431670

- *Product Owner*
- *Android Developer*

Shane McLaws - S3436830

- *Scrum Master*
- *Front End Developer*
- *Java FX Developer*

Callum Pullyblank - S3378543

- *Back End Developer*
- *Front End Developer*
- *Storage Manager*

Sonia Varghese

- *UI Designer*
- *Homepage Developer*

Table of Contents

1. About the Product

Trading Wheels is a game developed by Incom Systems for desktop and android. It is designed to simulate the Australian Securities Exchange (ASX) for the purposes of helping people practice their skills in trading stocks on a real market, without any risk to their real life money.

Users can create an account so that their funds and stocks are saved between game sessions. This allows them watch the market, and to buy and sell stocks when they want to. Stocks can be searched for by their name, or their ASX trading code, allowing for quick navigation of the ~2000 different shares that exist in the current market. Users are also able to view a summary of their previous transactions, as well as a history of their account's value. On top of all this, users are ranked on a leaderboard, based on the total profit each player has made over the course of the game.

Administrator accounts also exist to moderate and control the game. Admins are able to view user's accounts, and modify them. Modification involves adjusting user's available funds, owned stocks, as well as having the ability to delete user accounts all together. Admin's can also control the brokerage fee on transactions, adjusting both the flat fee and percentage fee components where they see fit.

2. Application Operational Details

2.1. Preparing Stock Data

The end user application is written in Java, exported and run as a .Jar file. On launch, the application retrieves a CSV list of ASX codes from the Amazon S3 Storage (ASX-JSON-HOST), and references this list to download share data. The application then starts 9 threads to download the ASX data. Each thread handles 250 stock codes, with the last one handling whatever is left over. At time of writing, the total number of shares being retrieved was 2063. When retrieving stock data, only the most recent file is retrieved. Once a file has been retrieved, the program grabs the last JSON entry in the file, and adds it to a JSONObject ArrayList.

The amount of time this process takes is dependent on the user's internet connection, but on a machine with a 60Mb/s connection, this took roughly 14s to complete. After the data has been downloaded, a bubble sort is applied to sort the stock information within the ArrayList by the ASX Code.

While the download and sort processers are happening in the background, users are able to register new accounts on log in. If a user logs in before the download has completed, the ASX Codes will still appear in their account summary and in the stock listing, but stock information (value, growth etc.) won't appear until that particular stock has been loaded into memory.

2.2. User Login and Registration

The actual login and registration processes are handled by a server module running on an AWS EC2 instance. Details on how the user data is stored are explored more in Section 6. 'Server Operations'.

When a user logs in, the game hashes the email address and password entered, and sends these to the server. The server will either respond with a 401 error (Unauthorised) if the details don't match an existing account, or will respond with the user data saved in storage. Once logged in, the game will display the user's profile summary, displaying the stocks they own, including the quantity of each they own, and their current trading price.

When a user registers a new account, the game takes in the users email address, desired password, first name and last name. These details are then sent to the server, which creates the user listing in storage. If the user is successfully created, the server responds with 200. (OK). The application then logs the user in with the details entered in registration, using the same method as described above. If the user could not be successfully registered, the server will return a 500 error code (Internal Server Error).

2.3. Retrieving and Displaying User Data

Details on how the user data is stored are explored more in Section 5. 'Storage Class Details'. When the user successfully logs in, the server returns the user's data in the form of a series of JSON strings. The first string represents the user's general data (Name, email, current funds, owned stocks etc.). This string is then followed by the keyword 'transaction', followed by the user's transaction history in the form of a series of JSON strings. This is then followed by the keyword 'value', followed by the user's value history in the form of a series of JSON strings.

When this information is received, the application takes the user JSON string and creates a new player object using the details in the JSON. The user's transaction history and value history is stored in JSONObject ArrayLists.

The transaction history ArrayList is referenced to display the user's transaction history on their profile. This shows time time/date of a transaction, what stock was traded, the quantity traded, and the value of the stock at the time of transaction.

The value history ArrayList is used to generate the graph on the user profile that shows the total value of the User's account over time. The total value is calculated as the total funds on the users account, plus the total value of all stocks owned by the user. The value history is calculated at the close of the trading day.

2.4 Displaying ASX Share Data

Real time ASX Data shown in the summary table is retrieved when the application starts. Each entry features stock code, stock name, and current trade price.

Clicking on a stock will show that particular stock in more detail. This includes more of the information retrieved at application start (such as Day High, Day Low, Change(%) etc.), as well as a graph showing the stocks worth over the course of the last month. This history of the stock is retrieved through a server call. The application tells the server which stock code it wants history for, and the date range it wants, and the server returns a series of JSON strings for the stock in that date range. Each JSON string contains the stock code, the date, and the stock value at the end of that day.

2.5 Making Transactions

Users are able to buy shares by selecting a share in either the all shares table, or by selecting a share they already own in their user profile. If the share selected is one the user already owns, the user has the option to also sell their shares. When they select one of these options, the application prompts them to enter the number of shares they wish to buy. This window dynamically updates to show the transaction total as the user enters the quantity, taking into account the current brokers fee. Once the purchase button is pressed, the application sends a message to the server telling it what stocks were bought, how many were bought, how much each stock was worth at the time, the current time and date, and which user made the purchase. The user's account is then updated in storage to reflect the transaction, and the entry is logged in their transaction history.

The transaction fee is dynamic and can be set by the Admin account (see section 3. Admin Functionality). When a user is making a transaction, the application sends a message to the server requesting the current broker's fee for the appropriate transaction type. If for whatever reason the server cannot be contacted or returns an error, the application will use the default value of $50 + 1% for buying stocks, and $50 + 0.25% for selling stocks.

2.6 Viewing the Leaderboard

Users are able to view the leaderboard so they can see who is winning the game. Scores are calculated as the total profits players have made playing the game. Total Funds + Total value of owned shares - 1,000,000 (starting money). Any score that resolves to less than 0 is displayed as 0. The leaderboard is stored on the server as a csv file. Every odd numbered entry is the user's unique id number; every even numbered entry is that user's score. The application requests this list when the user selects the view leaderboard button, and the results of the csv get displayed as a table.

2.7 Sending Messages

Users are able to send messages to each other. When messaging each other, users must specify a user to receive the message by their login email address. The sender can also specify a subject line if they wish, and of course the message body. Messages within a user's inbox can be marked as unread, and deleted. If a message is deleted, it is marked as deleted, but still exists in the user's storage. It won't appear in the list of user messages, unless the user specifies they wish to view deleted messages. A user can deleted a 'deleted' message, and this will remove the message from the user's storage. Messages are stored in the user's folder on the AWS storage (as described in section 5), and are retrieved, viewed, and interacted with through various server calls (as described in section 6).

2.8 Transferring Funds

Users are also able to send funds to each other. When sending funds, the sender must specify a user to receive the funds by their login email. They can then send as much money as they have available on their account. When they send off the funds, the funds are immediately removed from their account. The receiving user then receives a notification that they've received funds from another player. They can accept any amount up to the amount sent, with any remaining funds being transferred back to the sender. If a funds transfer goes unaccepted for 5 days, it is automatically reversed by an automatic script that runs on the server.

3. Admin Functionality

3.1. Load User Data/Unload User Data

In order for an admin to alter a user's data, the admin need to first load the user's data into memory. The getUserList function queries the server for a list of all users, and displays this list as a list of email addresses. The admin can then select a user from this list to load into memory.

3.2. Change User Balance

This function allows an admin to change the balance of the loaded player. Once the player's balance has been changed, the player's updated data is saved to the server.

3.3. Add/Remove Stock (from user)

This function allows an admin to add or remove a quantity of shares to a user account. Once the changes have been made, the player's updated data is saved to the server.

3.4. Delete User

This function allows an admin to delete a user's account from the server. This action is permanent, and when this function is invoked, the admin is asked to enter their email address and password to confirm the action.

3.5. Set Buy/Sell Fee

This function allows the admin to set the buy and sell fees that get charged when players make a transaction. The admin can set both the flat fee and the percentage fee with this call.

3.6. Message All Users

This function allows the admin to send a message to all users that play the game. The admin can specify the subject line and the message body, and then that message is sent to all users immediately.

4. Java Class breakdown

4.1. Admin.java

This class holds the functions available to an admin account. These functions consist of a number of player functions, such as loading a user's data into memory, printing that user's data, unloading a user's data from memory and viewing a list of users. Once a user is loaded into memory, the admin can change that user's balance, add or remove shares from the user's account, and deleting the user's account. Functions to change the broker's fees, as well as sending a message to all user accounts exist in this class as well.

4.2. AsxGame.java

This class has the main function that is invoked when the application is launched. It holds all of the global variables needed to track local game state, such as the Player object of the logged in user, connection information to the game server, and various lists referenced by other classes. The main function invokes the LoadASXData class to start downloading ASX data, and initializes the UI.

4.3. AsxPull.java

This class holds the functions required to retrieve ASX Data from storage, and format it for use within the application. As each data point is retrieved, it is read from the JSON Object, and a new 'Stock' object is created, and then added to an ArrayList in AsxGame.java.

4.4. AsxPullThread.java

This class implements 'Runnable' and runs as a thread that manages the downloading of ASX data through the functions found in AsxPull.java. When this thread is invoked, it is given a start point and endpoint, which defines which block of the stock list it is responsible for downloading.

4.5. Creds.java

This is an abstract class that holds the AWS Credentials needed to connect to the AWS storage to retrieve data being stored.

4.6. Game.java

This class holds functions relating to playing the game. These functions are: buyStocks, sellStocks, login, logout, registerPlayer, saveActivePlayer, loadPlayer, getValueLeaderboard, getStockCurretPrice, getStockHistory, calcBrokersFeePurch, calcBrokersFeeSale, sednMessage, getMessageList, getDeletedMessageList, getMessage, getUnreadMessageList, deleteMessage, markUnread, sendFunds, acceptFunds, getFundsList, playerDeleteSelf. The basic operations of these functions are covered in Section 2. 'Application Operational Details'.

4.7. LoadASXData.java

This class implements 'Runnable' and runs as a thread that manages all of the threads (AsxPullThread.java) used to split up the downloading of ASX data, and keep track of progress. Once all of the AsxPullThread's have finished, this class then implements a bubble sort on the list, to sort the share data by ASX Code alphabetically.

4.8. MessageCheck.java

This class implements 'Runnable' and runs as a thread that checks the logged in user's mail box, for any new messages or a funds transfer. This thread checks the user's mail box every minute, and if a new message is found in the mailbox, generates a notification in the UI.

4.9. Player.java

This class holds the constructor to create the player object, and all of the variables that describe a player whilst they're logged in. It also holds functions needed for the user account to react to changes, such as making transactions. The functions in this class only affect the actual player object at run time. The function generateDataSave takes the run time player data, and creates the string that will be saved to the server when the saveActivePlayer function in Game.java is called.

4.10. Stock.java

This class holds the constructor to create the stock objects that are used at run time. These stock objects are placed in an ArrayList to be accessed by other classes.

4.11. Utilities.java

This is an abstract class that holds a few functions that help with running the program. errorToLogFile is called to log any errors to a file to then be viewed later. asxErrorToLogFile is used to log any errors found when downloading asx data to a file for later review. sendServerMessage is used to send requests to the server application. The server's response is returned as a String.

5. Storage Class Details

5.1. ASX Data Management and Access

ASX data utilised in trading wheels is hosted in an Amazon S3 storage bucket, with new data being uploaded to the bucket every 20 minutes, between the hours of 10:00AM and 5:40PM on weekdays. This data is retrieved using an automated java application being run on an Amazon EC2 instance, which makes use of the Yahoo Finance API (http://www.marketindex.com.au/yahoo-finance-api). At the start of day (9:55AM), an initializer PHP script downloads a csv list of every tradable stock on the ASX (http://www.asx.com.au/asx/research/ASXListedCompanies.csv) and reformats the file so that it only contains the ASX trading codes in csv form. This formatted list is saved locally to the EC2 instance, and to the S3 bucket for later use. The S3 copy is saved as 'companies.csv'.

The main java application runs throughout the day and references this CSV list so that it knows which ASX codes to download information for. This script splits the task of downloading all ~2000 entries up into separate threads to speed the process up. Each thread downloads roughly 500 entries. Once the information has been downloaded, the script checks to see if the company is still alive. If it finds the "name" field is "N/A", the script discards the data. Otherwise, the script saves the stock code to an ArrayList; "stockCodes" for later use. Stock information is saved as a JSON file. Each JSON file is saved as *date*.JSON where the date is formatted as YYYYMMDD. Inside the file, there is one JSON line for each time that information is retrieved throughout the day, therefore a typical day will have 21 entries in the file. This file is then stored in a folder, which is named for the ASX code the data within it refers to. For instance, JSON data for Commonwealth Bank of Australia, for the 6th of April 2017, would be stored as /CBA/20170406.JSON. This naming convention allows for a specific days data for a specific company to be retrieved though the use of the AWS S3 API's. The JSON data within a file is formatted as follows:

{ "Time", "Name", "ASX Code", "Ask Price", "Bid Price", "Last Trade Price", "Last Trade Time", "Change", "Change(%)", "Opening Value", "Day High", "Day Low", "Previous Close", "52 Week Range", "52 Week High", "52 Week Low", "Dividend/Share", "Ex-Dividend Date", "Dividend Pay Date", "Dividend Yield"}

The java application makes use of the Amazon S3 API to help manage this data.

Once all ASX data has been retrieved and all threads have been merged, the script then performs a bubble sort on the list of stock codes. After the list has been sorted, it then overwrites the locally saved companies.csv and the companies.csv file stored in the S3, with the sorted list. That way, only companies that are alive are stored for reference later on.

5.2. User Data Files

The trading wheels game makes use of a back end User Management Server to manage user interactions with potentially sensitive data being stored in the S3 bucket. User data is stored in a few different ways in the S3 bucket. Within the User bucket, there are two folders, ‘data’, and ‘credentials’.

In the ‘credentials’ folder, each registered user has a file that contains a hash of their password, and their unique ID number. This file is name *emailHash*.rec, where *emailHash* is the hash function of the email they signed up to the game with.

Within the ‘data’ folder, each user has a folder called *ID* where *ID* is the ID number found in their credentials file. Within this folder, each user has three files, and a folder. Data.json, purchaseHistory.json, valueHistory.json, and the folder ‘mailbox’.

Data.json contains all of the user’s information, set out as follows:

{“Name”, “Surname”, ”Email”, ”Balance”, ”Shares”, ”Score”, ”Rights”}

Balance refers to how much money as user has in their account, with the default value on signup being $1,000,000.00. Shares refer to which shares a user currently owns, and the quantity owned for each. Score refers to a user’s score, which is calculated as a user’s total cash, plus the total value of all shares the user owns, subtract the initial starting $1000000. Any score that drops below 0 is set to 0. IE, it is impossible to have negative score displayed; however the actual calculated value could drop below 0.

PurchaseHistory.json contains a running log of all transactions made by a user. Individual entries are separated by new line feeds, and are set out as follows:

{‘Date’, ‘Time’, ’ASX Code’, ‘Qty’, ‘Price’, ‘Type’}

Type can be given as either purchase or sale. Price refers to the price per share at the time of the transaction.

ValueHistory.json contains a running log of the players score, as calculated at end of day. Individual entries are separated by new line feeds, and are set out as follows:

{‘Date’,’Score’}

The user’s mailbox folder contains any messages from other users. Each message has its own json file set out as follows

{‘Date’, ‘Time’, ‘Sender’, ‘Unread’, ‘Type’, ‘Contents’}

There is also a file for tracking the leaderboard in the bucket. This file is called ‘leaderboard.csv’ and is formatted as follows:

{‘ID’,’Score’}\n{‘ID’,’Score’}\n{‘ID’,’Score’}etc

This file is used to keep track of the order of players, by descending score, for purposes of showing a leaderboard in the user application.

6. Server Operations

6.1. Trading Wheels User Management Server

The User management server is designed to act as a gateway between the front end program and the user storage bucket hosted on S3. This server is always running on an EC2 instance, and requests can be sent by opening a socket connection to the EC2 server, on port 28543. The front end program makes requests to the server, which then responds with the appropriate data.

- Login Request (login\nemailHash\npasswordHash)

When the user wishes to login, the user program needs to make a login call to the server. This is done by sending the word 'login' followed by a new line feed (\n), followed by the hash of the entered email address, followed by a new line feed, followed by the hash of the entered password. The server program takes these inputs, and checks to see if a) the hash of the email matches an existing credentials file, and b) if the password hash received matches the password hash in the credentials file that matches with the email hash received. If these details match up, the server program returns the valid user's data file, the users transaction history, and the users value history. If the entered details don't match up, the server returns a '401: Unauthorised' http code.

- History Request (history\nemailHash\ntype)

The history request is used to retrieve the data within one of the history files belonging to a user. The user program sends the word 'history', followed by a new line feed, followed by the hash of the users email address, followed by a new line feed, followed by the type of history to be returned, where type can be 'transaction' or 'value'. The server will check to see if the email hash is a valid hash, and if it is, it will return the contents of the applicable history file to the user program, based on which was requested. If the email hash isn't valid, or if a history file is missing, the server returns a '500: Internal Server Error' http code.

- Register Request (register\npasswordHash\nfirstName\nsurname\nemailAddress)

When the user wishes to register for a new account, the user program needs to make a register request to the server. This is done by passing the word 'register' followed by the following fields, in the following order, separated by new line feeds: Hash of password entered, First Name, Surname, Email Address. The server then creates a hash of the entered email address, and checks if a credentials file matching that has already exists. If it does, the server throws a '500: Internal Server Error' error code. If the account doesn't already exist, the server generates a new unique ID for the account, and then creates the four files outlined in the previous section. It will also add the user to the running leaderboard file, with a score of 0. Once this is done, the server then returns a '200' http code.

- Save request (save\nemailHash\nnewJson\ntransaction)

When the user does something that requires their information to be updated, the user program needs to make a save request to the server. This is done by passing the word 'save' followed by a new line feed,

followed by the hash of the users email, followed a new line feed, followed by the new JSON string to be saved, followed by an optional new line feed and transaction listing. The server will check to see if a user account associated with the email hash exists, and if it does, it will rewrite the data file with the new json string passed to the server. The server will then read the new score value in the JSON line, and update the players position on the leaderboard. If a transaction was also passed in, the server will append this line to the end of the user transaction history file. If the save process was successful, the server will return a '200' http code, otherwise it will return a '500: Internal Server Error' http code.

- Leaderboard request (leaders\ntopVal\nnumVals)

When the user wishes to view the leaderboard, the user program needs to make a leaderboard request to the server. This is done by passing the word 'leaders' to the server, followed by a new line feed, followed by an integer representing the 0 indexed position of the top value to be returned, followed by a new line feed, followed by the total number of results to be returned. For instance, if the user wishes to view the top 10 people, the topVal must be 0, and numVals must be 10, so the request string would look like: 'leaders\n0\n10'. If the server was able to successfully complete the users request, the server will return a single string of users and scores in the following format:

{'Name','Score'};{'Name','Score'};{'Name','Score}; etc.

If the request could not be complete, the server will return a '500: Internal Server Error' http code.

- Get User request (getUser\nemailHash)

This function will be reserved to admin accounts, and the front end application will restrict access to this function. When requested, the admin can either get a specific user by passing in the hash of the users email address as the 'emailHash' flag, or can pass in an asterisk (*) to retrieve all users. If all users a requested, a list formatted as firstname, surname, email in JSON format will be returned. If a specific user is requested, the user's data JSON and transaction history will be returned.

If the request could not be complete, the server will return a '500: Internal Server Error' http code.

- Stock History request (stockHistory\nasxCode\nstartDate\nendDate)

This function allows the user application to get a basic history for a particular stock code over a period of time. The user application sends an ASX code, followed by the start date (in the format yyyymmdd), followed by the end date (in the format yyyymmdd). The server will respond will "200" followed by a JSON array as a String containing the history data. Each JSON object in the array is in the format of {ASXCode, Date, Ask Price}. The function returns results for all dates between and including the start and end dates supplied. The Ask Price given is the ask price at close of trading (5:40PM) or for the last data point available in the case of the current day being requested.

- setBuy/setSell request (setX\nflatFee\npercentageFee)

This function will be reserved to admin accounts, and the front end application will restrict access to this function. When requested, this allows the admin to alter the brokers fee on sales and purchases. The flat value represent the flat fee on a transaction, and the percentage value represents the percentage fee on a transaction. Brokers fee is calculated as 'transaction value' * 'percentage fee' + 'flat fee'.

- getBuy/getSell request (getX)

This function returns the current brokers fees values to the client program, relevant to the transaction type requested. These values are stored in the server program as a global variable.

- sendMessage request (sendMessage\nsenderEmaiLHash\nrecipientEmailHash \ncontents)

This function allows users to send messages to other users, or even to themselves. The sender and recipient fields must the hash of the email addresses and the contents field is the actual message itself. If the message is successfully posted to another user's mailbox folder, this server call returns 200. Otherwise it returns 500.

- getMessageList request (getMessageList\nuser)

This function allows a user to retrieve a list of all messages in their mailbox. The user attribute sent must be a hash of the users email address. If successful, the function will respond with a comma separated list of the ID's of all messages in the user's mailbox. If a user's mailbox is empty, the function will return 204. If unsuccessful, the function will return 500.

- getDeleted request (getDeleted\nuser)

This function allows a user to retrieve a list of deleted messages in their mailbox. The user attribute sent must be a hash of the users email address. If successful, the function will respond with a comma separated list of the ID's of all messages in the user's mailbox. If a user has no deleted messages, the function will return 204. If unsuccessful, the function will return 500.

- getMessage request (getMessage\nuser\nmailID)

This function allows the user program to retrieve a message from a user's mailbox. The given user field must be the hash of the users email address, and mailID must be the integer ID of the mail item being retrieved. If successful, the function will return the json object of the specified message item as a String. If unsuccessful, the function will return 500.

- deleteMessage request (deleteMessage\nuser\nmailID)

This function allows a user to delete a message in their mailbox. The user attribute must be the hash of the users email address, and mailID must be the integer ID of the mail item being deleted. Messages have a "Deleted" flag, if this flag is false when the function is called, then the flag is set to true. If the flag

is true when the function is called, the mail object is deleted from storage. If successful, the function will return 200, and if unsuccessful, will return 500.

- unreadMail request (unreadMail\nuser)

This function will return to the client, a comma separated list of mail ID's belonging to the given user, which are marked as unread. The user attribute must be the hash of the users email address. If unsuccessful, the function will return 500.

- markUnread request (markUnread\nuser\nmailID)

This function allows a user to mark a previously read message as unread. The user attribute must be the hash of the users email address, and mailID must be the integer ID of the mail item being marked unread. If successful, the function will return 200, and if unsuccessful, will return 500.

- getID request (getID\nuser)

This function allows the front end application to convert a user's email hash into their unique ID. If successful, the function will return the unique ID of the user being requested, and if unsuccessful, will return 500.

- deleteAccount (deleteAccount\nuser)

This function allows the user application to delete a particular account from the server. An admin can call this function, passing in the user's email address, whilst a user can only delete their own account. The function requires the user's email hash to be passed in, and if successful, all of the users data will be removed from storage, and their entry on the leaderboard will be removed. If successful, the function will return 200, and if unsuccessful, will return 500.

- sendFunds request (sendFunds\nsender\nrecipient\namount)

This function allows users to send funds to each other. Sender is the hash of the senders email address, recipient is the hash of the recipients email address, amount is the amount of money being send. If successful, the server will respond with 200, if unsuccessful, it will respond with 500.

- acceptFunds request (acceptFunds\nuser\mfundID\namount)

This function allows a user to accept either in whole, or a part of funds being sent to them. User is the hash of the email address of the person receiving the funds, fundID is the unique ID of the funds transfer, amount is the amount of money from the original transfer being accepted. If successful, the server will respond with 200, if unsuccessful, it will respond with 500.

- getFundsList request (getFundsList\nuser)

This function will return a list of message ID's that are funds transfers being sent to the user. The user flag is the hash of the user's email who wishes to see the list of funds transfers. If successful, the server will return a comma separated list of message ID's. If unsuccessful, the server will return 500. If the user has no pending funds transfers, the function will return 204.

6.2. Server calls summary

Call: login\nemailHash\npasswordHash

Successful return: userData.json\n'transaction'\npurchaseHistory.json\n'value'\nvalueHistory.json - as a string

Failure Return: 401

Call: history\nemailHash\ntype

Successful return: *type*History.json - as a string

Failure Return: 500

Call: register\npasswordHash\nfirstName\nsurname\nemailAddress

Successful return: 200

Failure Return: 500

Call: save\nemailHash\nnewJson\ntransaction

Successful return: 200

Failure Return: 500

Call: leaders\ntopVal\nnumVals

Successful return: leaders.json - as a String

Failure Return: 500

Call: getUser\nemailHash

Successful return: data.json\npurchaseHistory.json - as a String

Failure Return: 500

Call: getUser\n*

Successful return: a list of all users fName - sName - Email - as a String

Failure Return: 500

Call: setBuy\nflatFee\npercentageFee

Successful return: 200

Failure Return: 400

Call: setSell\nflatFee\npercentageFee

Successful return: 200

Failure Return: 400

Call: getBuy\n

Successful return: 200\nflatBuyFee\nperBuyFee

Failure Return: 500

Call: getSell\n

Successful return: 200\nflatSellFee\nperSellFee

Failure Return: 500

Call: sendMessage\nsenderEmailHash\nrecipientEmailHash \ncontents

Successful return: 200

Failure return: 500

Call: getMessageList\nuserEmailHash

Successful return: 204 - if specified user has no messages

Successful return: list of message ids, as CSV integers

Failure Return: 500

Call: getMessage\nuser\nmailID

Successful return: mail JSON object as a String

Failure return: 500

Call: getDeleted\nuser

Successful return: 204 - if specified user has no deleted messages

Successful return: list of message ids, as CSV integers

Failure Return: 500

Call: deleteMessage\nuser\nmailID

Successful return: 200

Failure return: 500

Call: unreadMail\nuser

Successful Return: CSV list of mail ID's of unread mail

Failure Return: 500

Call: markUnread\nuser\nmailID

Successful return: 200

Failure return: 200

Call: getID\nuser

Successful return: ID number of supplied user

Failure: 500

Call: deleteAccount\nuser

Successful return: 200

Failure return: 500

Call: sendFunds\nsenderEmailHash\nrecipientEmailHash\ntamount

Successful return: 200

Failure Return: 500

Call: acceptFunds\nuser\nfundID\amount

Successful return: 200

Failure return: 500

Call: getFundsList\nuser

Successful return: The entire list of message ID's of type 'funds' in users inbox

Failure return: 500