# Analysis

---

## Project Outline

This is a project aimed at helping groups of people manage money, secured with digitally signed transactions. It also provides a way to quickly and easily settle chains of debt [1]. To interact with the final product, a simple, easy to use command line interface will be provided.

**Features**

- Cryptographically signed transactions guaranteeing security and integrity of your transactions
- Simplify chains of debt in your group
- Command Line Interface (CLI)
- Client / Server Model
- Database

**Non-Features**

- None of the user's money is ever put into the app. This is simply a tracker, you cannot settle debts through the app
- No policing of people who do not pay their debt - this is a problem for people in the group to deal with as they choose

## Background to the Problem

A common problem for many young people is that of money. Specifically, keeping track of who owes who how much money in a group of friends. Arguments about how much money is owed are commonplace. This is something I often see amongst my own group of friends. I know one person in particular (the end user) feels as though he is never paid back, and would like to see a solution to the money tracking problem.

In order to arrive at a solution, what is needed is a reliable, trustworthy way to track money. People who use the tracker will need some sort of guarantee that people cannot 'hack' the app, changing people's debts. Since I am the creator, and likely a future user of this app, my friends also need confidence that I will not be able to write off all of my debts. Hence, that is problem number 1 - **secured transactions**.

A problem inherent to a money tracker is that it is just that - a tracker. Since no money flows through the software, long chains of debt may form. This may result in people needing to make many small transactions to different people in order to pay what they owe. This seems a shame, when it is possible to have the software simplify the debts into a minimal number

of transactions per person, to ensure money flows as efficiently as possible through the network. This idea was presented to me by the end user, but I anticipate it to be an integral part of the project, hence the brief mention here. This makes problem number 2 - **efficient settling of group debt**.

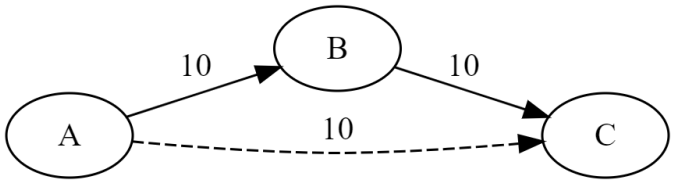
# The End User

The end user that I had in mind is the friend who inspired this project. I think he fits perfectly within the target market - 13-18 years old, fairly sociable, not too technologically minded, and just wants an easy way to keep track of his transactions in his group of friends.

My interviewee was keen to point out that people may abuse this system. They may just use it to rack up debt, and then never pay anyone back. He wanted to know if there was a way that I could stop this occurring. To this I replied no, not really. They can do the same thing without the app. It is your decision whether to lend to them. With the app however you can see exactly how much they've taken - it provides indisputable accountability. It does however assume that people are willing to pay up. It is up to the user to deal with the eventuality that they don't.

Another problem that he identified was that of chains of debt. He (rightly) pointed out that if you owe people who owe people, it's sometimes easier to cut out the middle man and turn two discreet transactions into one smaller one. This idea naturally extends to a group of friends, who may all have varying levels of debt between them. Thus, instead of making the group do a large number of transactions with money going back and forth frequently between the same hands, I will aim to let a group settle in the easiest way possible.

A valid concern with this plan is the fact that some may end up owing people they didn't before the simplification. Consider a simple case where A owes B £10, and B owes C £10. Two transactions could be reduced to 1 if A were to pay C directly. However, in a larger group, people may not like giving money to people who they do not directly owe on the will of my program. Hence, an important constraint is that no one owes someone that they didn't owe before settling occurred (see image below.)




Even though the dashed edge would reduce transactions, it should not be added.

The end user suggested that I keep the interface as simple as possible. He maintained that he didn't want lots of unnecessary frills - just a simple, functional interface. To this, I suggested the use of a CLI. I was a little concerned that most people would not have used one before and may not know what it is. However, once I explained the concept, he seemed to come round to the idea. It's main benefit over a graphical user interface (GUI) is that it is unambiguous. It is also, arguably, easier to do things with a CLI once you become comfortable using one.

The final main worry that my interviewee brought up was that of guaranteed security. I had talked to him when I had the idea for this project, before I had learned about asymmetric encryption and cryptographic signatures. He wanted to know how a transaction coming from him could be verified as his, and no one else could pretend that they are, say, owed lots of money. He also didn't trust me, and said that if our group of friends started using this product, he would suspect that I would "code away my debts".

---

In short, the problems identified here are as follows:

- How to make sure that users trust the integrity of the transactions, making sure that users know that no one can tamper with their debts.
- How to settle debt across large graphs efficiently (here meaning few transactions per person)
- How to make a CLI that is as simple as possible

This is not an exhaustive list - it leaves out all the technical problems I will likely face, which are discussed in the next two subsections

---

# Research of existing solutions

Currently, on the market, there are a few products similar to that which I am proposing. Having surveyed a few options, I decided to look at Evenfy and Splitwise in more detail.

Both work on the same premise that I have outlined: an intuitive way to track who owes who in a group.

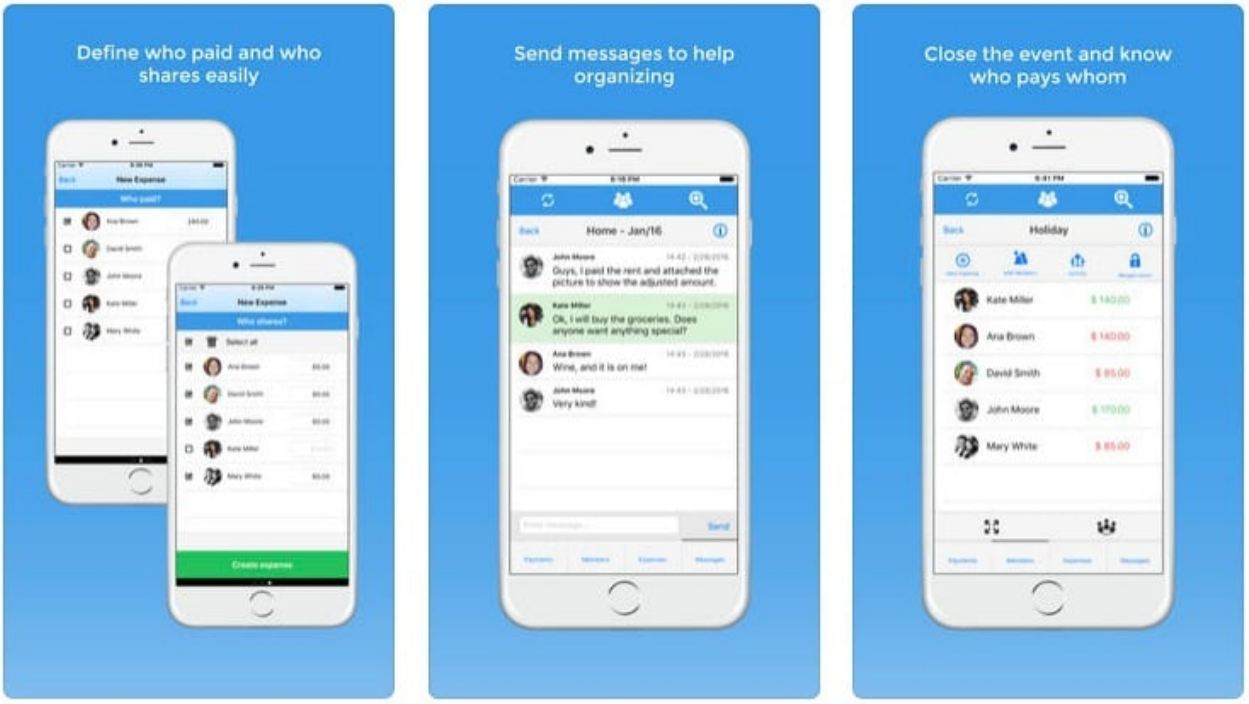
This is all accurate at time of writing.

## Evenfy

Evenfy is an app that does exactly what I set out to achieve. It mainly focuses on group expenses, and can be accessed from a computer.

It has an interesting feature in that it allows for temporary groups to be created in order to track short term expenses, such as over the course of an event. Evenfy tries to learn about common expenses, and suggests who pays over time. This may be useful if you rent a house with a few others, is what Evenfy say.

Evenfy will also calculate the easiest way to settle the group; that is, ensure that everyone's debt goes to 0. This is an integral part of keeping an expense tracking app usable in my opinion, and in the app's reviews, users seem to agree.

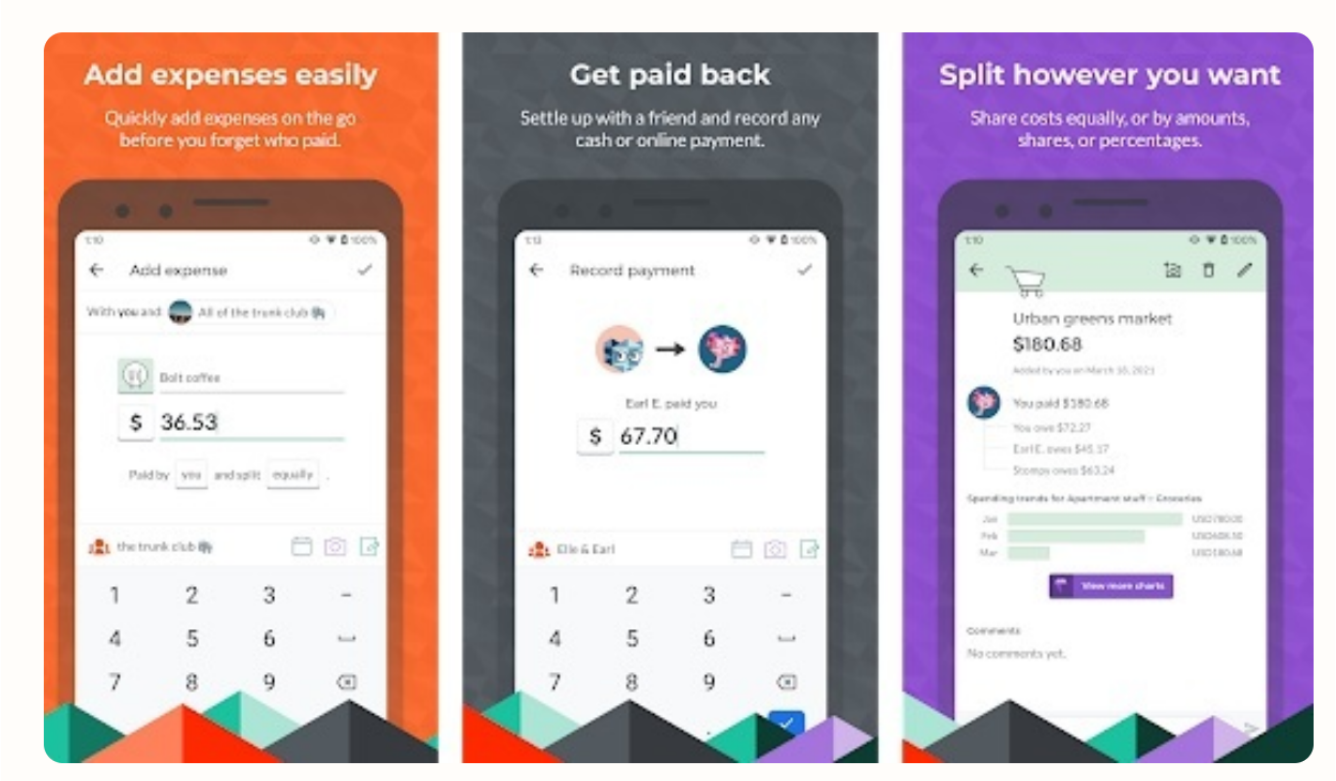



Evenfy also allows for a group to settle debts easily. This feature will be discussed more in the next product evaluation, as an identical feature appears there.

Evenfy is free to use for the first 6 months, but then requires monthly subscription of 99 cents if you want to track more than 10 expenses per month.

## Splitwise

Splitwise is similar to Evenfy in many ways. It allows the easy splitting of bills (by percentage or equally), and keeps track of who owes whom within the group.



After an account is created, you can create a group of people. Splitwise will then track all expenses in this group. Expenses can be referenced, and you can see at a glance exactly how much you owe.

In comparison with Evenfy, the overall experience and feature set is similar. Both are well put together and include features that I think are unnecessary for my target market. I do prefer Splitwise's UI slightly: it is easier on the eyes, and slightly less cluttered. Both have an excellent UX.

My only complaint is that I feel as though I have to search through a user interface for many quite simple tasks. The settings menu in particular feels like it obfuscates settings such as deleting your account, as well as updating user information. This is, however, common to many graphical user interfaces. As I do not plan on building a graphical interface, I do not think this is something that I need to be too mindful of. Instead, I will strive to make the CLI as straightforward as possible.

In terms of flaws, Splitwise requires a £2.99 per month subscription to unlock every feature (most core features are free to use). This is, in my opinion, better than Evenfy's payment model. However, the point is moot as I will not be charging for the use of this project.

The interesting part of these apps is in the debt simplification process. Since neither are open source, one cannot know for sure how the debt simplification is done. However, after much investigation, I found a few possible options.

---

## Givers and Receivers

This, I believe, is the less likely of the two possible approaches, because it reduces down to a decision theory problem which is NP-Complete. It also takes a few passes of the data to get there. It is not particularly efficient. The steps are as follows

1. Calculate the net flow of each node
2. Categorize nodes into 'givers' (those who overall owe money) and 'receivers' (those who are overall owed), and those who owe/are owed nothing.
3. Settle any '1-1 transactions', where one 'giver' can completely remunerate a 'receiver'.
4. Settle any transactions where a receiver is owed a perfect subset of givers money (i.e. a receiver who is owed £5 could be settled by two givers, with £3 and £2)
5. Settle any remaining transactions by splitting money from givers in a greedy fashion to receivers.

The 'NP-completeness' of the problem starts at stage 4, when lots of computation has already been done on the data.

Stage 1 reducing all the transaction from Person A → Person B to a single number. This is then done for the whole group, and a weighted digraph is built, where edges represent money owed. A residual graph, wherein an edge in the opposite direction with a weight $\times = -1$ the initial weight is added.

A breadth-first-search, where each edge traversed from the current node (*u*) has its weight recorded (and summed when all edges from *u* are explored) is required to obtain the flow of the graph. This gives a time complexity of $\mathcal{O}(|V| + |E|)$.

The second step can be done very efficiently, in $\mathcal{O}(logV)$ time if a mergesort is used to sort the nodes in descending order of money they have (assuming those owed have negative amounts of money). Then the list of nodes can be split into two subsets at the point where 0 would be inserted into the list (those with a net debt of 0 can be removed before splitting).

Step 3 could be done by walking through the sorted givers list. For each node *g* in the givers list, you would walk down the receivers list (*r* in receivers), settling any transactions where *g*[money] = r[money]. (Any nodes which are not 'filtered' in this way move to a new stage of processing). This would give a time complexity of $\mathcal{O}(G \cdot R)$ , where G is the number of 'givers,' and R is the number of 'receivers'. This can be made much more efficient by using sorted lists, and remembering how far the receivers list was walked down in previous runs.

In pseudocode

```
for giver in givers:
        for receiver in receivers:
                if giver.money > receiver.money:
                        // giver has more money than any receiver in receivers
                        giver passes filter
                else if giver.money < receiver.money:
                    // receiver is owed more money than any giver in givers has
                        receiver passes filter
                else:
                        // giver and receiver have the same amount of money so a 1-
1 can happen
                        settle(giver, receiver)
```

Step 4 becomes a sum of subsets problem, and is NP-Complete, i.e. there is no algorithm which can solve this problem in quicker than exponential time. It is at this stage that this approach becomes infeasible for the real world, and thus, A better heuristic must be considered.

Another problem with this approach is that it contradicts one of my initial high level requirements. It does not preserve any sense of who owes whom past step 1. Thus, it cannot guarantee that no one will owe anyone that they did not previously owe.

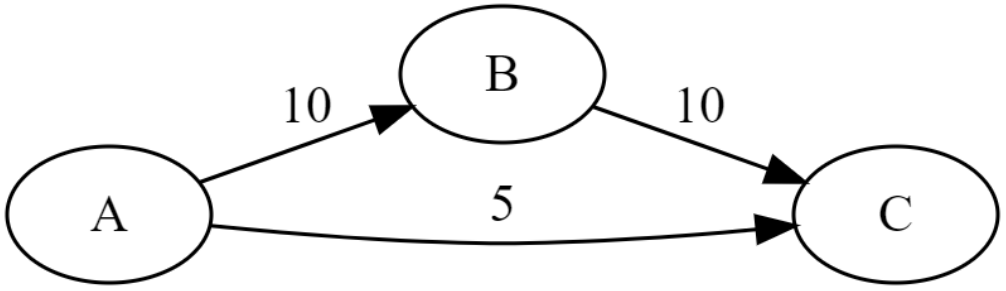
For these two reasons, I shall elect to not use this method.

---

## Max Flow

I think that this problem can be modelled as a problem of flow. The question 'how can we minimise the number of transactions one person has' can be reduced to 'how do I maximise the amount of money I send to one person'. If the amount of money sent to someone is maximised such that no one ever has to pay more than they owe, then people will, on average, have fewer transactions to make after settling.

Consider the simple case of three people, Alice, Bob and Charlie. Let Alice owe Bob £10, Bob owe Charlie £10, and Alice owe Charlie £5. This can be represented as a weighted digraph




Notice that there is a chain from A -[10]→ B -[10]→ C.

This chain shows that all £15 from Alice will eventually end up with Charlie, even though some of it goes via Bob. Thus, this can be simplified by cutting out the middle man, and instead letting Alice owe Charlie £15.

I believe that this functionality can be achieved through a slightly unusual application of the Edmonds-Karp Max Flow Algorithm.

## Fulkerson-Ford Max Flow

The Edmonds-Karp Max flow algorithm is really a combination of two separate algorithms: a breadth first search, and the Ford-Fulkerson max flow algorithm.

Ford-Fulkerson aims to answer the question: how much flow can one push along a network, without exceeding the capacity of any edge? The algorithm works on flow graphs.

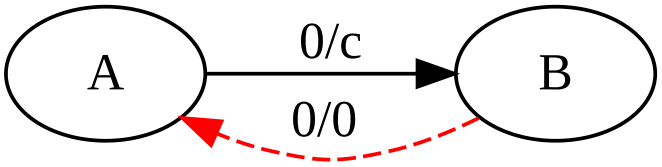
The way in which the algorithm works is simple:
1) Find an augmenting path from source node to sink node (through the residual graph)
2) Augment flow down path
3) Repeat until no more augmenting paths exist

To define some terms:
**Flow Graph**
A flow graph is a form of weighted digraph, in which edges have a flow and a capacity (as opposed to a single weight). Each edge has the notion of remaining capacity (remaining capacity := capacity - flow). Each edge is initialised with flow = 0. This value changes during the course of the algorithm.

The flow of an edge is never allowed to exceed its capacity. (i.e. edge will never have a negative remaining capacity).

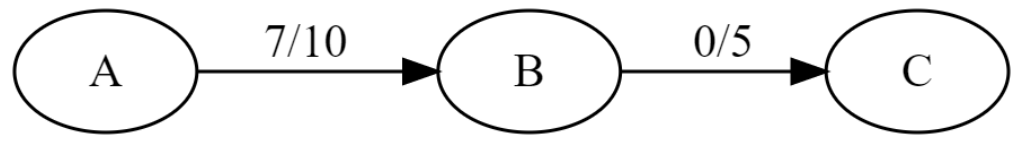
**Augmenting Path**
The augmenting path is a path of edges in the residual graph, where each edge has a remaining capacity > 0. The path is from two specified nodes - a source node $s$ and a sink node $t$.

**Residual Graph** and **Residual Edges**
The residual graph is the combination of the flow graph , and residual edges. For each original edge from $u$ → $v$ , with capacity $c$, there exists a residual edge from $v$ → $u$, with capacity $0$.

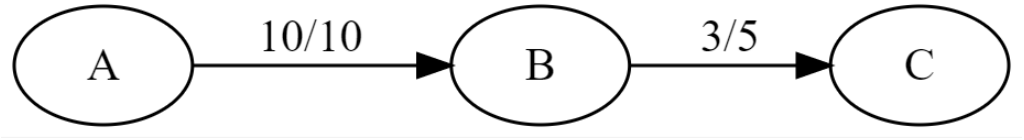



Residual edges are valid edges to consider when looking for an augmenting path, given that they have unused capacity (the above example's residual edge has an unused capacity $0$, as $0 - 0 = 0$).

**Augmenting the flow**
The act of pushing as much flow as possible along an augmenting path.
The amount of flow pushed down the path is equal to the bottleneck value of the path. The bottleneck value is given by the edge with the smallest amount of unused capacity




In the above case, the bottleneck value is 3. This is because A →B has 3 units of unused capacity, and B → C has 5 units of remaining capacity. Thus, the maximum amount of flow that can be pushed through this augmenting path is 3 units. After augmenting the flow, the path looks like this




When flow is augmented, it is important to keep the net flow through the node the same. This is done through the residual edges, which are updated so that they have a flow of $-c$. In this example, the residual edge from C → B would have a capacity of $-3$.

An example of the algorithm working is provided in the next subsection

poOnce no more augmenting paths can be found, the bottleneck values of each of the augmenting paths are summed. This value is the max flow through the given graph from the source node to the sink node.

### Complexity

Finding an augmenting path is completed in $\mathcal{O}(E)$ time (where $E$ is the number of edges in the graph). In the worst case, 1 unit of flow is added every iteration. This makes the overall time complexity of the Fulkerson-Ford Max Flow $\mathcal{O}(E \cdot f)$, where $f$ is the max flow of the graph.

This is not ideal, as the time complexity is heavily dependent on the flow through the graph. This is improved upon in the strongly polynomial Edmonds-Karp Max Flow algorithm.

## Edmonds-Karp Max Flow

Edmonds-Karp Max Flow differs from Fulkerson-Ford in the finding of augmenting paths. Fulkerson-Ford does not specify how an augmenting path should be found, whereas Edmonds-Karp finds the shortest [2] augmenting path from $s$ to $t$.

This is ensured by using a Breadth First Search (BFS) to find augmenting paths.

A short augmenting path is favourable, as the longer the augmenting path is, the higher the chance of an edge with very little unused capacity. This could lead to edges reaching capacity in more iterations, giving a considerably slower runtime. As aforementioned, the worst case is that every path has a bottleneck of 1 unit of flow. Since Edmonds-Karp uses a BFS to find augmenting paths, we are guaranteed the shortest (in terms of number of edges traversed) path from $s$ to $t$.

This detail gives Edmonds-Karp a much more favourable time complexity, of $\mathcal{O}(EV^2)$. This is a product of the time complexity of a BFS, $\mathcal{O}(V^2)$ (when using an adjacency matrix to represent the graph) and the Fulkerson Ford time complexity, $\mathcal{O}(E \cdot f)$. However, flow does not appear in the time complexity of Edmonds-Karp.

A property of BFS is that, when it finds a path from a source node $s$ to a target node $t$, that path is guaranteed to be the shortest path ($P_n$) from $s$ to $t$. A corollary of this is that $P_{n+1}$ is guaranteed to be a longer path than $P_n$. This reduces the upper bound run time of one iteration of Edmonds-Karp to $\mathcal{O}(E)$ versus the original $\mathcal{O}(E \cdot f)$. A more complete proof of time complexity is provided here [3]].

Since Edmonds-Karp's runtime is independent of flow, its input, it is classed as a strongly polynomial algorithm, making it perform much better than the Fulkerson-Ford max flow algorithm. Thus, this is the algorithm that I will be implementing to simplify debts across a group.

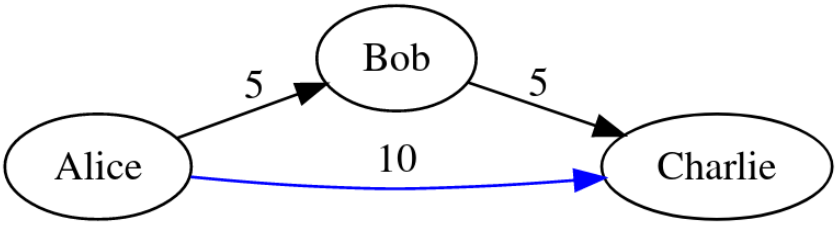
# Settling a graph using a Max Flow algorithm

Having explored various max flow algorithms, the question now becomes how to settle an entire graph's worth of debt. This is a fairly challenging problem since max flow algorithms only work on a source node and a sink node.

After researching, I found a solution which proposed the following.

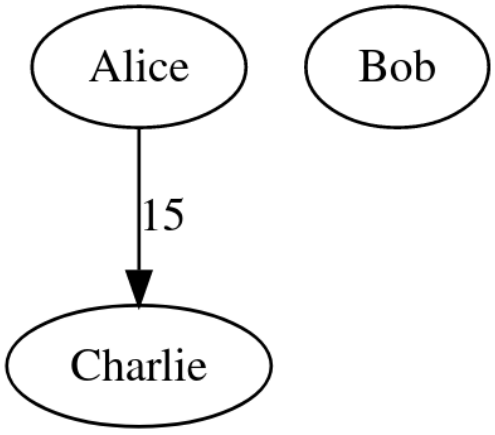
```
// initial graph is the initial network of debts

clean_graph = WeightedDigraph()
for edge(u, v) in initial_graph:
         if flow := max_flow(u, v):
                 //  append an edge to the new graph from u -> v with weight flow if
flow > 0
                 clean_graph.append(u, v, flow)
                 // remove edge that has been maxflowed
                 initial_graph.remove_edge(u, v)
```



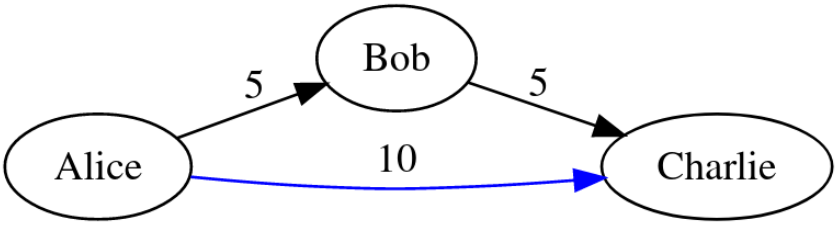

In prose, a new weighted digraph is generated (with no edges, but all the same nodes) for the cleaned edges.

Starting on node $G$, we would therefore run a max flow from $G \to B$ . If the max flow from $G \to B > 0$, then an edge with the weight of the max flow from $G \to B$ is added to the new weighted digraph. The edge from $G \to B$ in the flow graph is deleted. This process will happen again from $G \to D$. After having explored all neighbours, the BFS continues, until every edge in the graph has been settled. In the above example, a valid settling could look like this [4]



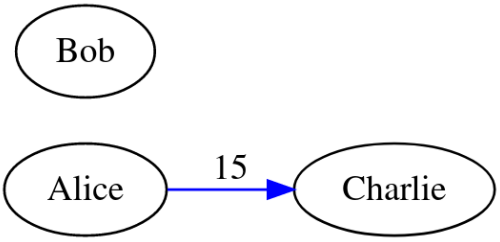

However, while hand tracing the aforementioned algorithm, I discovered that it was not a correct algorithm.

Take the original, unsettled graph, and select the edge in blue first

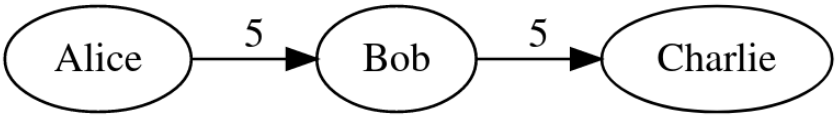



Running Edmonds-Karp along this network with Alice as a source and Charlie as a sink gives a max-flow of 15. Thus, add an edge to the new graph from Alice to Charlie with a weight of 15, and remove the Alice - Charlie edge from the initial graph

This gives the new graph:

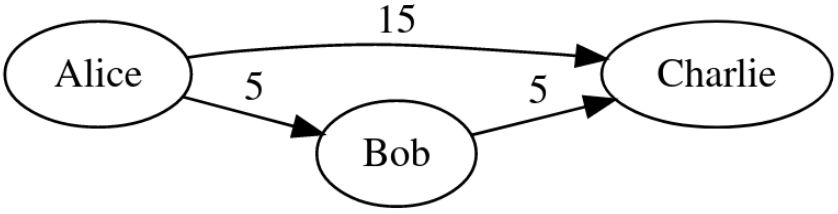



And the 'initial' graph




Repeating this process for the remaining two edges gives us max-flows of
Alice → Bob: 5
Bob → Charlie: 5 (after removing Alice → Bob)

The 'clean' graph that is generated will therefore look like this




Notice that initially, Alice owed the group £15. Now she owes the group £20. Similarly, Charlie was owed a total of £15 pounds by the group, and is now owed £20.

Thus, this algorithm is incorrect.

The reason why is that it does not account for how max-flow is generated. In the case of the first edge we just considered, we calculated a max-flow of 15. This was achieved by pushing 5 units of flow down the Alice → Bob → Charlie path, and 10 units of flow directly from Alice → Charlie.

Since all of these paths become saturated, it should be the case that **no more flow can be pushed through the graph**. This algorithm only removes the considered edge, instead of all the edges saturated by the max-flow algorithm.

The approach I decided to take was to modify the initial graph in place. After a max-flow is run, and a new edge is added to the clean graph, the 'initial' graph is restructured. The restructuring replaces the capacity of each edge $(u, v)$ to its original unused capacity. Like this, any saturated edges are removed, and future iterations of the graph are constrained to only produce results based on outcomes of previous iterations.

You may assume that this step is unnecessary, as it should be the case that the max-flow algorithm does not consider saturated paths. However, the max-flow runs on the residual graph, and thus may augment flow in a way that 'removes' money from a previous transaction. Since the transactions are solidified in another graph, this creates a discrepancy, and was where the error in the algorithm I found online lay.

Hence, I implemented the algorithm as follows:

```
clean = FlowGraph()

 for edge(u, v) in graph:
         if flow := max_flow(u, v):
                // append an edge to the new graph from u -> v with weight = flow
                clean.add_edge(u, v, flow)

                // restructures edges as described above
                graph.reduce_edges()

                // for loop now runs on until no more edges in graph;
                // thus stops when no edges left in initial graph
```

In the worst case scenario, only 1 edge is removed from the graph each time `graph.reduce_edges()` is called. In this case, 1 max-flow would happen per edge of the graph, giving a time complexity of $\mathcal{O}((EV^2) \cdot E) = \mathcal{O}((EV)^2)$ . However in practice, more than one edge will become saturated with each max-flow, reducing the expected time complexity.

---

Note: **Implications to security**
Since the server that is settling a group of transactions is creating and destroying new transactions that haven't happened in the real world, the signatures that transactions were initialised with will no longer be valid after settling.

Thus, the user should resign any transactions in the group that they are involved in after the settling has happened. This allows the solution to remain secure, and also lets the user see exactly what has happened to their debt.

# High Level Objectives for the Solution

After careful consideration of the end user, and existing systems, I can arrive at my high level and low level requirements

On a high level, my objectives are as follows:

1. An RSA implementation that will allow the signing, and verification, of transactions
2. A way to settle the debts of the group in as few as possible (heuristically speaking) monetary transfers
3. A server-side component of the application which can verify transactions, and store / retrieve them from a database
4. A client-side component of the application that will have a simple user interface (CLI)
5. A database that should be able to store user and transaction information

# Low Level Requirements

For the purposes of testing, these are low level requirements that I would like to fulfil. The first three sections are very low level, and as such may be hard to understand in context of the project. Section D paints a picture of how achieving my aims in sections A, B and C will apply to my project.

## RSA Implementation (A)

1. A reliable interface to a hashing module
2. RSA Key Handling:
    1. Be able to load RSA public/private keys in PEM format from files / STDIN
    2. Be able to validate the format of these keys
    3. Be able to parse these keys extracting all necessary numbers for RSA decryption
3. Signing/Verification
    1. Have a valid RSA encryption scheme (encryption with public key)
    2. Have a valid RSA decryption scheme (decryption with private key)
    3. Have a valid RSA signing (sig) scheme (signing with private key)
    4. Have a valid RSA signature verification (verif) scheme (verify with public key)
4. Object Signing
    1. Algorithm to convert an object to a hash in a reproducible way, minimising the chance of hash collisions
    2. Ability to sign a class of object with RSA sig scheme
    3. Ability to verify a signed object with RSA verif scheme, raising an error if signature is invalid

## Debt Simplification (B)

1. A reliable digraph structure, with operations to `transactions.graph.GenericDigraph`
    1. Get the nodes in the graph `nodes()`
    2. Check if an edge exists between two nodes
    3. Nodes can be added
    4. Nodes can be removed
    5. Edges can be added
    6. Edges can be removed
    7. Neighbours of a node should be easily accessed (neighbours for the purposes of a breadth first search)
2. A reliable flow graph structure
    1. All the operations listed in B.1.1
    2. Adding an edge should have different functionality: edge should be able to be added with a capacity, and edges should have a notion of flow and unused capacity
    3. Be able to return neighbours of nodes in the residual graph (i.e. edges, including residual edges, that have unused capacity)
    4. A way to get the bottleneck value of a path, given a path of nodes
3. A reliable recursive BFS that works on flow graphs
4. Implementation of Edmonds-Karp
    1. Way to find the shortest augmenting path between two nodes
    2. Way to find bottleneck value of a path
    3. Finding max flow along a flow graph from source node to sink node
5. Simplifying an entire graph using Edmonds Karp, using the method laid out in Settling a graph using a Max Flow algorithm.
6. Be able to convert a list of valid transactions into a flow graph
7. Be able to convert a flow graph into a list of transactions, signed by the server
8. Be able to simplify a group of transactions, having each transaction individually verified before settling

## Client / Server Structure (C)

1. The server should be accessible to the client via a REST API
2. The client should be relatively thin, only dealing with input from user and handling error 400 and 500 codes gracefully.
3. Client and Server should communicate over HTTP, using JSON as an information interchange format
4. The client should have a clear, easy to use command line interface

## 'Integrated' requirements for how the end system should behave (D)

1. Ensuring the validity of transactions
    1. If a transaction is tampered with in the database, it should be classed as unverified
    2. A user should not be able to sign an already signed transaction
    3. A user should not be able to sign a transaction where they are not one of the listed members
    4. A user should not be able to sign a transaction with a key that is not associated to their account
    5. A user should not be able to sign a transaction without entering their password correctly
    6. Every time a transaction is pulled from the database and sent to the user, it should be verified by the server using the RSA sig/verif scheme from section A
2. Ensuring that the debt simplification feature works
    1. All transactions in the group being settled should be verified upon being pulled from the database
    2. It should not be possible to simplify a group if there are unverified transactions in the group
    3. If the transaction structure of the group does not change, the user should be notified
    4. The simplification should accurately simplify a system of debts such that no one is owed / owes a different amount of money after simplification
    5. The simplifying process should result in unverified transactions being produced, able to be signed by the user
3. Ancillary features
    1. Users should be able to register for an account, providing name, email, password and a PEM formatted private key
    2. Users should be able to create transactions where they are the party owing money; these transactions should be created as unsigned
    3. Users should be able to create a group with a name and password
    4. Users should be able to join a group by group ID
    5. Users should be able to mark a transaction as settled; transactions should only be marked as settled when both parties involved mark the transaction as settled
    6. Users should be able to see which groups they are a member of
    7. Users should be able to see all of their open transactions
    8. Users should be able to see all the open transactions in a group (whether they are part of the group)
    9. Users should be able to see the public key information of any user on the system
    10. Users should be able to see individual transactions by passing in a transaction ID

## Database Architecture (E)

1. User information

    1. User ID
    2. Contact info
    3. A hash of the user's password
    4. Associated Groups
    5. Public Key (provisions for one or more)
2. Transaction Information
    1. Transaction ID
    2. Payee
    3. Recipient
    4. Transaction reference
    5. Amount (£)
    6. Payee's signature
    7. Recipient's signature
    8. Whether transaction has been settled
3. Group information
    1. Group name
    2. Group password
    3. People in the group
    4. Transactions in the group

# Project Critical Path

The order in which I will carry out the project in 6 distinct phases

1. Cryptography
2. Debt Simplification
3. Combination of 1 & 2 into a fully encompassing transaction object
4. Database setup and design of SQL statements to retrieve data
5. An API designed to allow communication between client and server
6. User Command Line Interface design

In more detail (Key: Black boxes with coloured text are labels)


# Design

## High Level System Design

Briefly, my goal for the system is to be able to create transactions, and digitally sign / verify them. Then, the system should be able to simplify chains of debt among people.

The notion of signing and verifying means that the project will require a sizeable cryptography aspect, especially as is my intention to write the RSA encryption / decryption myself.

I will also need a mechanism with which to be able to simplify large groups of debt. As discussed in the analysis section, this will be done through the use of flow graphs.

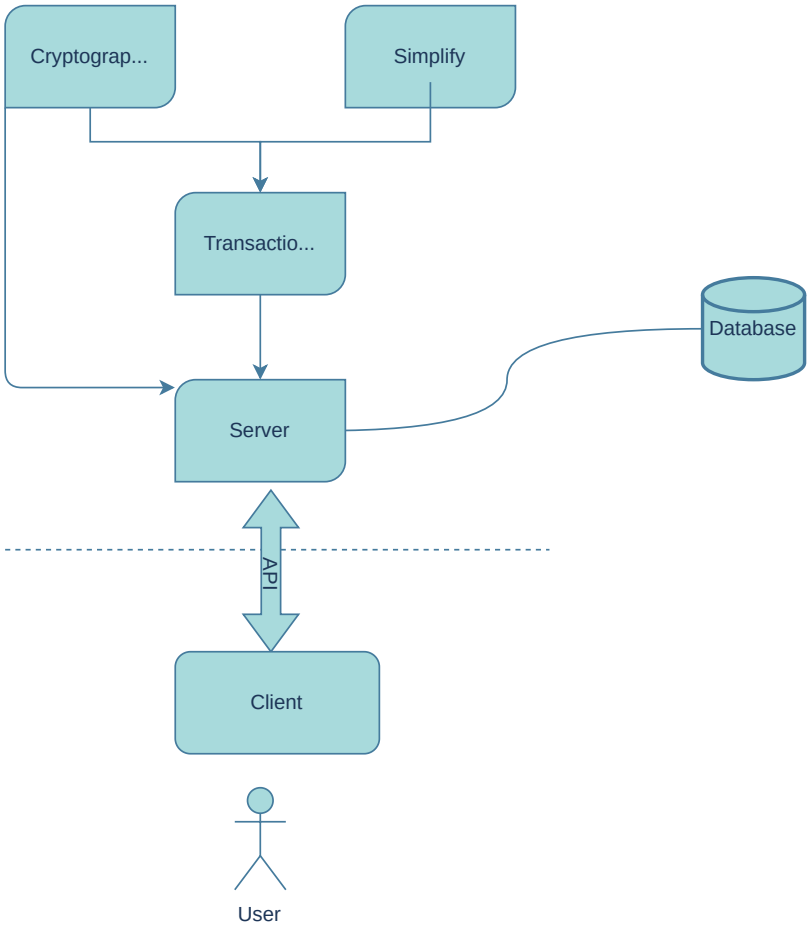
Next comes the transactions themselves. Only at the transaction level should the concepts of cryptography or the graph processing start being used.

Finally, I will assemble everything together at the server level, although the simplification module should never need to be directly used. The cryptography module will require light

use for the hashing of passwords.

To store these transactions, along with user data and supporting infrastructure, a fairly complex database system will be employed.

As a diagram




Note: the client needs a way to generate hashes of passwords. I intend to use the cryptography module that I will write. This is not necessary however, as any SHA3_256 hash will be sufficient.

On a high level, the system is decomposed into 5 separate modules, each accounting for a main component of the system (as above). These are `crypto`, `simplify`, `transactions`, `client` and `server`. Three of these modules, `crypto`, `simplify`, and `transactions` will all have a folder of unit tests associated with them.

The modules in the final design should be as decoupled as possible. `crypto` and `simplify` should not have any dependencies on any other modules (as will become apparent in class diagrams above)

## Expected File Structure

```
.
├── README.md
├── requirements.txt
├── settle_db.sqlite
```

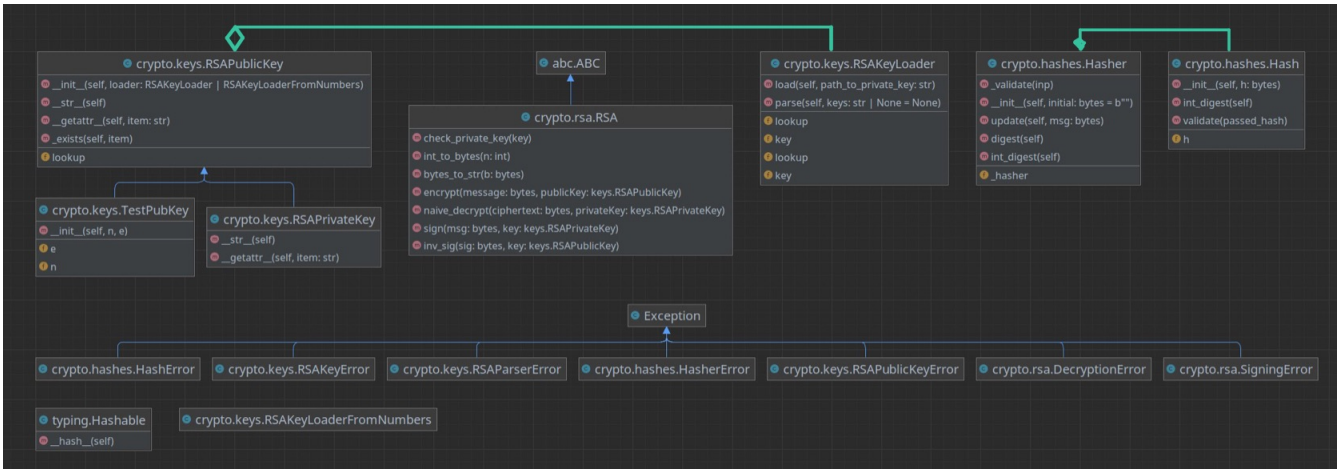
```
├── setup.py
├── src
│   ├── client
│   │   ├── client.py
│   │   ├── cli_helpers.py
│   │   ├── cli.py
│   │   └── __init__.py
│   ├── crypto
│   │   ├── hashes.py
│   │   ├── __init__.py
│   │   ├── keys.py
│   │   ├── rsa.py
│   │   └── sample_keys
│   │       ├── d_private-key.pem
│   │       ├── d_public-key.pe
│   │       ├── m_private-key.pem
│   │       ├── m_public-key.pe
│   │       ├── t_private-key.pem
│   │       └── t_public-key.pe
│   ├── __init__.py
│   ├── server
│   │   ├── endpoint.py
│   │   ├── __init__.py
│   │   ├── log
│   │   ├── models.py
│   │   ├── processes.py
│   │   ├── resources.py
│   │   ├── schemas.py
│   │   └── setup.py
│   ├── simplify
│   │   ├── base_graph.py
│   │   ├── flow_algorithms.py
│   │   ├── flow_graph.py
│   │   ├── graph_objects.py
│   │   ├── __init__.py
│   │   ├── path.py
│   │   └── weighted_digraph.py
│   └── transactions
│       ├── __init__.py
│       ├── ledger.py
│       └── transaction.py
└── tests
    ├── test_crypto
    │   ├── __init__.py
    │   ├── test_hashes.py
```

```
    │   ├── test_keys.py
    │   └── test_sign_verify.py
    ├── test_settling
    │   ├── __init__.py
    │   ├── test_flow.py
    │   ├── test_graph.py
    │   └── test_Path.py
    └── test_transactions
        ├── __init__.py
        ├── mock_db.csv
        ├── test_ledger.py
        └── test_transaction.py
```

The interactions of different classes are shown diagrammatically below. If I were to provide an entire class diagram for the system, it would be unreadable. Hence, I have broken diagrams down by module, referencing objects from other modules where appropriate.

# Class Diagrams by Module

## Cryptography (`crypto`) Module




Key for this, and all following class diagrams: red *m* indicates a method, yellow *f* indicates a field. Method signatures are included, with type hints of parameters where relevant to aid in highlighting relationships. Protected variables are denoted through name - they start with an underscore

Above is a class diagram for the `crypto` module.

The primary objective of this module is to handle all the security needed by the application. This mainly involves a consistent way to ensure the validity of transactions, as well as their origin. It also will take care of hashing the passwords of users and groups so that they are not stored in plaintext. This is a more minor role, however.

The `RSAPublicKey` and `RSAPrivateKey` objects are lightweight. Their only field is a dictionary, which stores parts of the key, and an identifier. Since an RSA key needs three components to work (when implemented, as it is here, with modular exponentiation

encryption/decryption), each component is stored separately in this dictionary. The `__getattr__` method will be overwritten from the `object` parent class so that it is possible to access parts of the key as one would access an attribute (i.e. for the modulus `RSAPublicKey.n`)

## How RSA Works

RSA is an asymmetric encryption algorithm which works by taking the plaintext (encoded by an integer), raising it to a very large number, and then taking a modulus of a different very large number

$$c = p^e \mod n \quad (1)$$

where $c$ is the ciphertext, $p$ is the plaintext, $e$ is the public exponent, and $n$ is the modulus. How the these numbers are generated is beyond the scope of this project. To decrypt, a similar relationship is used

$$p = c^d \mod n \quad (2)$$

where $d$ represents the private exponent.

In public key cryptography, everyone has access to the modulus and public exponent in a key, but it is very important that no one except the owner has access to the private exponent. For this reason, I will ensure that the private exponent is never sent across the network.

To create digital signatures, the process is similar. First, a hash of the message is generated. This is done by the `Hasher` object above. Then, equation $(2)$ is used, with $c$ representing the hash of the message in integer form, as opposed to the ciphertext. Similarly, $p$ now represents the digital signature instead of the plaintext. The signature is then appended to the message.

To verify the digital signature, equation $(1)$ is used. If the signature is valid, the output of this equation should be the hash of the message. Thus, one hashes the messages and compares it to the outcome of equation $(1)$. If the two match, then the signature is valid. However, if they don't match, either the message has been tampered with or the signature has been tampered with.

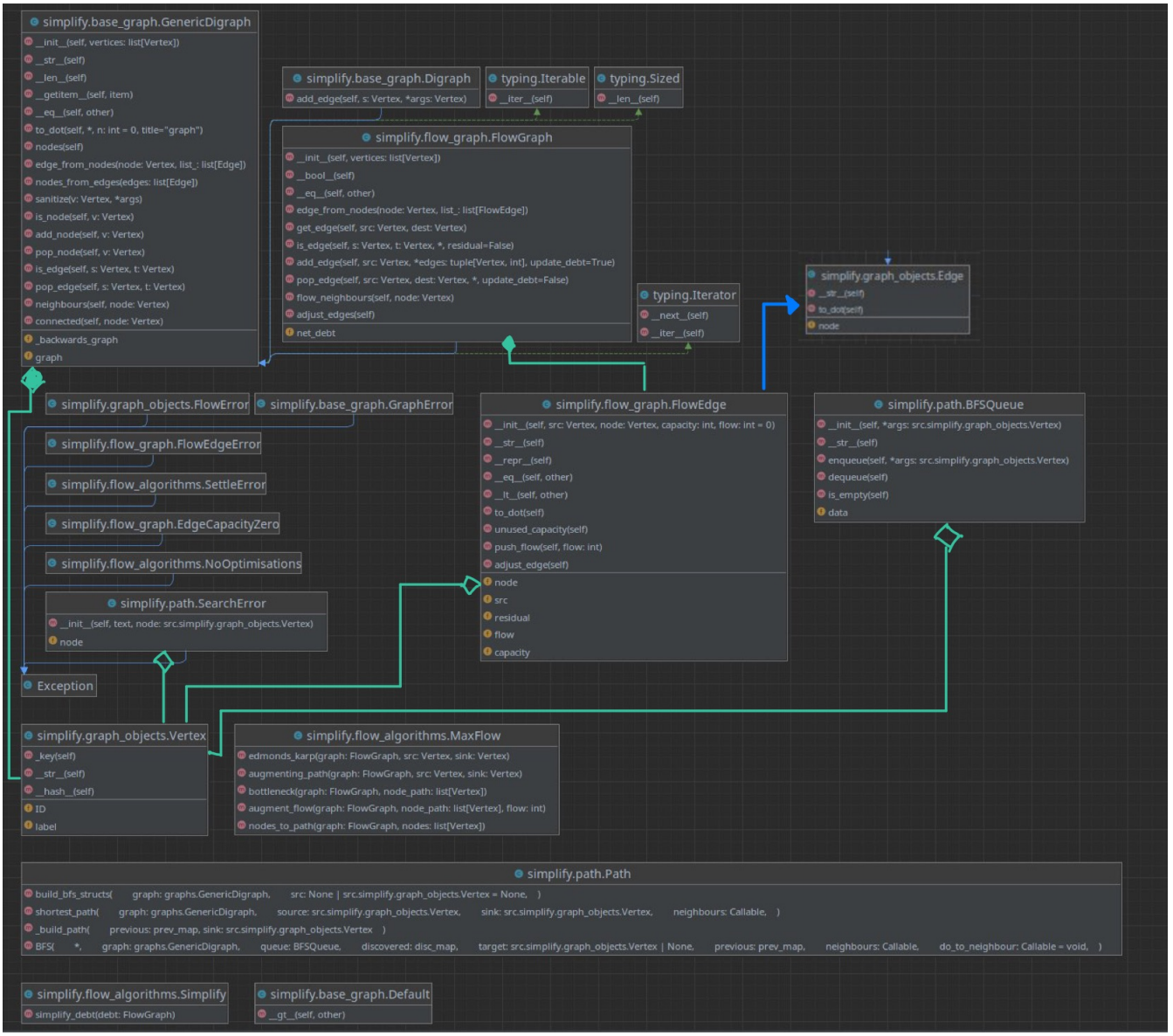
Thus, it is possible to ensure validity, and verify the origin, of messages.

I will implement this exactly in this way, using the Python builtin `pow()` to carry out the modular exponentiation, and the builtin `hashlib` library to generate hashes. I will, however, be writing my own interface to `hashlib` to add extra functionality, such as ensuring that all hashes that I generate are padded correctly so that the encryption / decryption works as expected.

I will also implement the loading of the RSA key in PEM format myself, using regexes to filter out only what I need, and package the resulting numbers into keys accordingly.

# Debt Simplification (`simplify`) Module

## Class Diagram




This is by far the most intricate module. It shows a complex object-oriented model (furthered in the `transactions` module). It also involves complex key data structures, as well as various complex algorithms - both well known and user defined.

The debt simplification module is dedicated to simplifying a graph using the algorithm specified in the analysis section. It does not have any dependencies on any other modules from this project.

The task of debt simplification is decomposed into four key areas: The flow graph data structure, graph search algorithms, graph flow algorithms, and the assembly of the main simplification interface.

### The Flow Graph Data Structure

As I outlined in the project critical path, I wanted to have a basic unweighted digraph data structure, which I could perform breadth first searches (BFS) on before I started to consider

flow. To do this, I started with just a `GenericDigraph` class. The graph has only two fields: the `graph`, and the protected field `_backwards_graph`, to aid with the deletion of nodes.

The graph is effectively represented as an adjacency matrix. I use a dictionary, which maps a node to a list of `edges`. `edge` objects contain a `node` field, which represents the destination node, i.e. where the edge is pointing.

The base graph then has various bookkeeping methods such as checking if nodes are in the graph, checking if a node is associated a list of edges (and vice versa). You can also add and remove nodes and edges.

I designed a `neighbours` function to return the neighbours of a node, and a `connected()` function, seeing if a node has any connections in the graph.

I mentioned that the `_backwards_graph` was necessary when it comes to deleting nodes. This is because there must be a way of traversing the graph backwards to find which nodes are pointing to the node that you want to delete. If you do not protect against this, then you will end up with edges pointing nowhere.

The `_backwards_graph` will be managed every time an edge is added / deleted from the graph. When an edge$(u, v)$ is added to the forwards graph, an edge$(v, u)$ is added to the backwards graph. Then, when deleting a node, all that needs to be done is look at the connections of the given node in the backwards graph, and delete the edges from those nodes in the forwards graph.

This should provide everything necessary to implement breadth first search. I opted to do this recursively. The function signature did not fit on the class diagram, so is included here

```
def BFS(
    *,  # star indicates keyword-only args
 graph: graphs.GenericDigraph,
 queue: BFSQueue,
 discovered: disc_map,
 target: src.simplify.graph_objects.Vertex | None,
 previous: prev_map,
 neighbours: Callable,
 do_to_neighbour: Callable = void,
) -> prev_map:
```

Note: `prev_map` and `disc_map` are not objects but custom type aliases. They are equivalent to

```
prev_map = dict[src.simplify.graph_objects.Vertex,
src.simplify.graph_objects.Vertex | None
]

disc_map = dict[
    src.simplify.graph_objects.Vertex, src.simplify.graph_objects.Vertex | bool
]
```

To do the BFS, I first require a queue data structure. I will implement one myself instead of using a builtin queue. A BFS queue is interesting, as it should not allow for the same element to be enqueued twice. Thus, the data structure I will create will be an ordered set with only two operations: `enqueue()` and `dequeue()`.

I also need a data structure to keep track of the nodes that had been previously discovered, and where they had been discovered from. This will be important to be able to reconstruct a path through the graph. For this, I will use a dictionary of type `dict[Vertex, Vertex]`, where keys are vertices in the graph and values are where they were discovered from.

Since the BFS will end up being used in more than one way (searching through the standard graph, or searching for augmenting paths as part of the maxflow algorithm), it is important to specify to it how to look for neighbours. Since functions are first class objects in Python, this is done by passing in a function as and when it is needed.

Similarly, with `do_to_neighbour`, different algorithms that use the BFS will require different things to be done to the neighbours of a node. Hence, this is specified when the function is called, as opposed to when it is defined.

Here is a python mock-up of how I intend to implement the BFS

```
@staticmethod
def BFS(
     *,
         graph: graphs.GenericDigraph,
         queue: BFSQueue,
         discovered: disc_map,
         target: src.simplify.graph_objects.Vertex | None,
         previous: prev_map,
         neighbours: Callable,
         do_to_neighbour: Callable = void,
        ) -> prev_map:

         # will only happen if no path to node
        if queue.is_empty():
            return previous

    else:
        # discover next node in queue
                current = queue.dequeue()
        discovered[current] = True

         # check we haven't been fed a standalone node (i.e. no forward or
backwards links)
                 if not graph.connected(current):
                        if not queue:
                raise SearchError("Cannot traverse a non connected node",
current)

                # if discovered target node return prev
                 if current == target:
```

```
                    return previous

                else:
            # otherwise, continue on
                        # enqueue neighbours, keep track of whose neighbours
they are given not already discovered
                        # do passed in function to neighbouring nodes
                        for neighbour in neighbours(current):
                if not discovered[neighbour.node]:
                    previous[neighbour.node] = current
                    queue.enqueue(neighbour.node)

                do_to_neighbour(current, neighbour.node)

            # recursive call on new state
                        return Path.BFS(
                            graph=graph,
                                queue=queue,
                                discovered=discovered,
                                target=target,
                                previous=previous,
                            neighbours=neighbours,
                                do_to_neighbour=do_to_neighbour,
                        )
```

After I implement the BFS, I will move onto the max-flow algorithm.

This requires an updating the graph data structure, and the edge data structure. To do this, I will inherit from the `BaseGraph` and `Edge` that I will have already designed.

The key changes to the edges (in a new class `FlowEdge`) will be:
1) Edges will need two new fields: `flow: int` and `capacity: int`, where capacity is a non-negative integer, and flow is an integer
2) Edges will need a new method: `unused_capacity()` which will return `capacity` - `flow`
3) Flow graphs contain a residual edge, as discussed in the Analysis phase. This will be accounted for with a field `residual: bool`. Residual edges will be treated as such.
4) In order to ease transaction integration later, I will also have edges explicitly track their `src` and their destination (`dest`). This will allow me to build a list of transactions just from a list of edges, but will be discussed further later on.
5) An `__eq__` function is needed, allowing differentiation between residual edges [5]

The key changes to the graph, in the new class `FlowGraph`, will be:
1) A backwards graph is no longer needed, instead it will be possible to utilise residual edges to traverse the structure the wrong way when deleting nodes
2) Adding edges now entails adding a residual edge counterpart, as discussed in the Analysis section. Thus, when edges are removed, their residual edge also needs to be removed. Hence, the `add_edge` and `pop_edge` functions need to be overwritten to work with `FlowEdge` objects.
3) Any pair of nodes should be restricted to just one forward edge. Thus, if there exists an edge from A → B of weight 5, and an edge from B → A of weight 10 is added, the graph

should result in one edge from B → A with weight 5.
4) A `flow_neighbour()` method needs to be introduced, as valid neighbours in the max-flow algorithm are any edges with unused capacity. This is different to a valid neighbour in the BFS, which is any forward-pointing non-residual edge.
5) A function is also needed to adjust the edges in the flow graph to become an edge with no flow, and only unused capacity remaining. This is added under the identifier `adjust_edges()`.
6) A way of verifying that the simplifying has resulted in a fair graph, where people owe and are owed the same (net) amount of money as they originally were. This is done with the `net_debt` field, which is a `dict[Vertex, int]`

In the analysis section, I detailed how the Edmonds-Karp max-flow algorithm works. I will implement it exactly as described in the analysis section. All the methods which are involved in finding the max-flow through a flow graph will be contained in the `flow_algorithms.MaxFlow` object. I will decompose the task of finding the max_flow into 5 functions. Their signatures are listed below

```
class MaxFlow:

    @staticmethod
        def edmonds_karp(graph: FlowGraph, src: Vertex, sink: Vertex) -> int:
...

    @staticmethod
        def augmenting_path(graph: FlowGraph, src: Vertex, sink: Vertex) ->
list[Vertex]:  ...

    @staticmethod
        def bottleneck(graph: FlowGraph, node_path: list[Vertex]) -> int: ...

    @staticmethod
        def augment_flow(graph: FlowGraph, node_path: list[Vertex], flow: int)
-> None: ...

        @staticmethod
        def nodes_to_path(graph: FlowGraph, nodes: list[Vertex]) ->
list[FlowEdge]:  ...
```

`augmenting_path` makes use of the recursive breadth first search that will have been implemented to find the shortest path through the graph from the source node to the sink node(in terms of number of edges).

`nodes_to_path` will be used to convert the list of nodes returned by the BFS function to a path.

`bottleneck` will return the bottleneck of a path through the graph, and `augment_flow` will augment the flow of a path, changing the `flow` field in each `FlowEdge`.

These are all combined in the `edmonds_karp` function to return an integer - the maximum flow from the given source node to the given sink node.

A python mock up of the implementation of the `edmonds_karp` function is given here

```
def edmonds_karp(graph: FlowGraph, src: Vertex, sink: Vertex) -> int:

    max_flow = 0

        while aug_path := MaxFlow.augmenting_path(graph, src, sink):
            bottleneck = MaxFlow.bottleneck(graph, aug_path)
        max_flow += bottleneck

        MaxFlow.augment_flow(graph, aug_path, bottleneck)

        return max_flow
```

With all the components having been designed, it is possible to integrate the process entirely. The `Simplify` class has one method: `simplify_debt(graph: FlowGraph)`. This combines all of what is above into my user-defined algorithm to simplify the graph as a whole. Again, this algorithm works exactly as laid out in the analysis section.

For every edge in the graph, a max-flow is run between the nodes at either end of the edge. This changes the state of the graph, as augmenting the flow through the graph will change the flow on edges / residual edges.

After the max-flow is run, an edge is added to the clean graph with a weight of the max-flow, and the `adjust_edges()` method is run on the 'initial' graph (initial in inverted commas because, of course, its state will have been changed by the algorithm. Calling it 'initial' just differentiates it from the clean graph that I'm building along the way).

This process should continue until there are no more edges in the 'initial' graph.

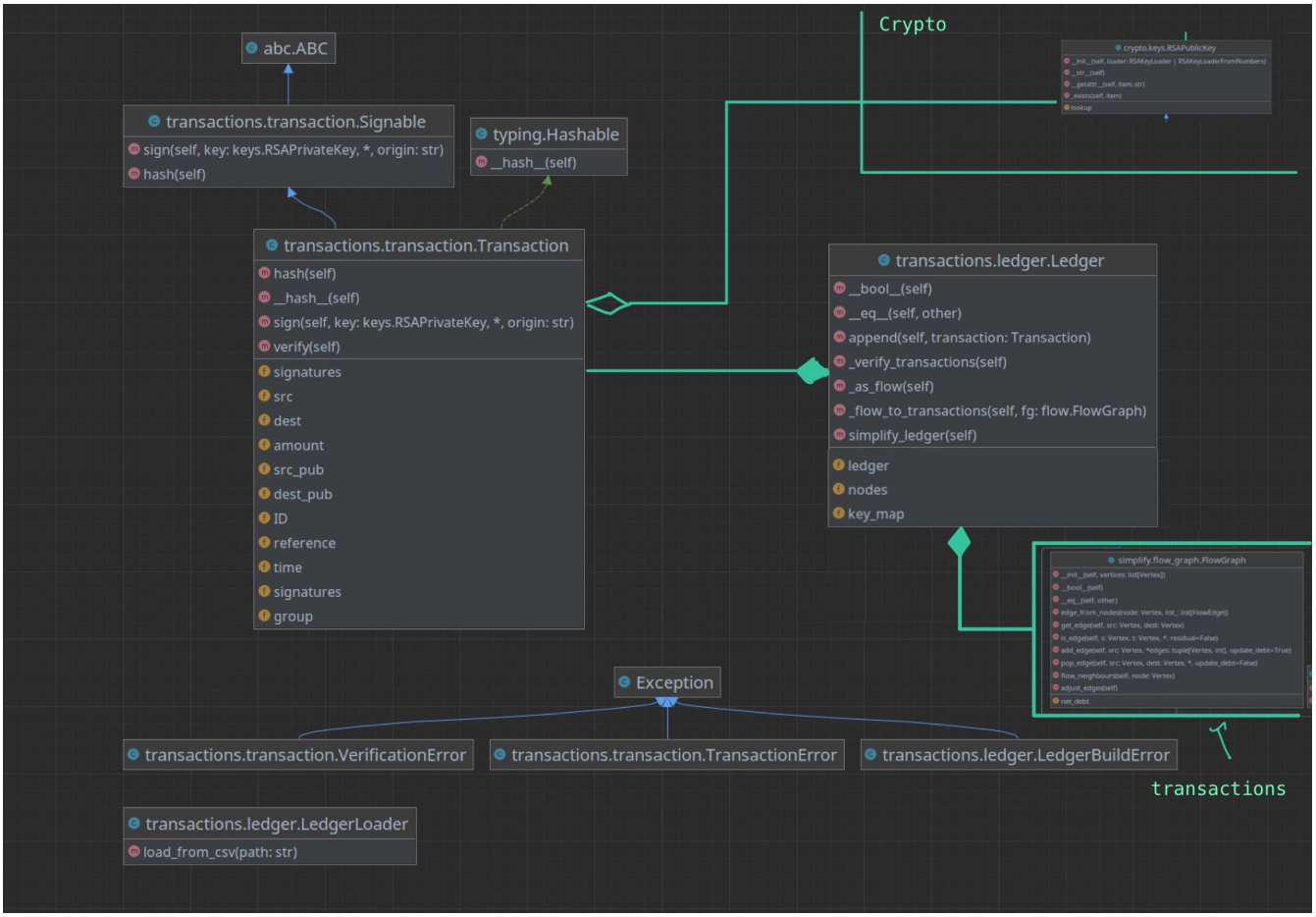
In pseudocode

```
for edge(u, v) in graph:
        if new := maxflow(u, v):
                clean.add_edge(u, (v, new))
                messy.adjust_edges()
```

Once the simplification has concluded, the net debts of the new graph should be compared with the (cached) net debts of how the graph was initially. These should always be identical, and the simplification process will raise an error if this is not the case, indicating that simplification should be aborted and retried.

This scenario is never expected to arise, but this failsafe should be implemented in favour of writing robust code.

## Transaction integration (`transactions`) module




The purpose of the `transactions` module is to combine the `crypto` and `simplify` module - to allow transactions to be created, signed and verified.

A `Ledger` object will also be introduced to represent a group of transactions, and with the ability to simplify them, ensuring that they are all verified before doing so.

Since this module is effectively the assembly through aggregation and composition of the `crypto` and `simplify` modules, it does not have much in the way of complex data structures or algorithms.

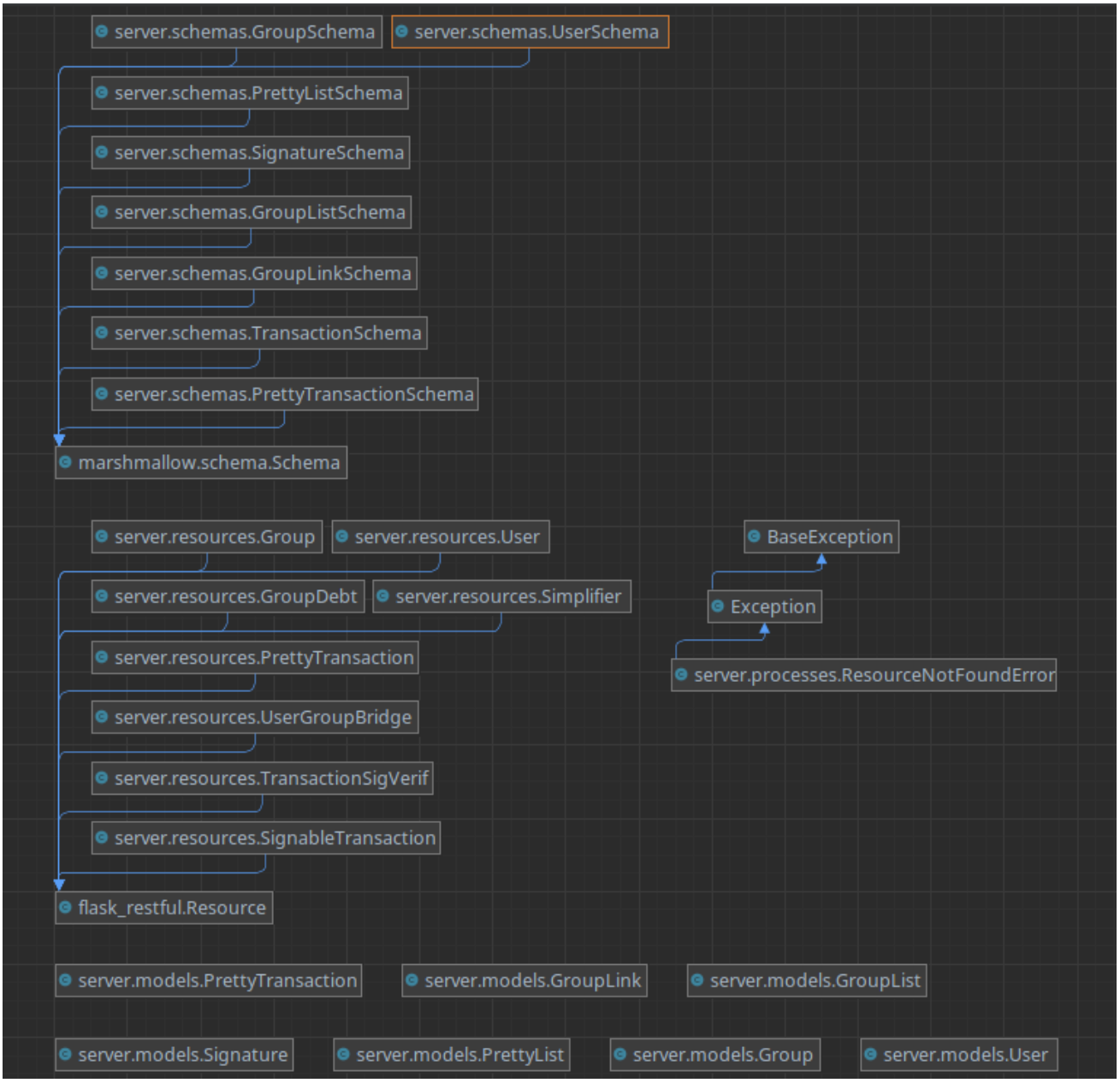
Signatures are stored as bytes inside the transaction object. Time is stored as a `datetime.datetime` object, and keys are stores as `crypto.keys.RSAPublicKey` objects (as per class diagram).

Ledger's `ledger` field is a list of `transaction` objects. Its `nodes` field is a list of all the people in the ledger, used to generate the flow graph that it creates to simplify transactions. `key_map` keeps track of which key belongs to which user ID.

The `Transaction` class inherits from the `Signable` class. This happens due to the order in which I intend to implement the project. The `crypto` module comes first, and I need to be able to test signing objects before I have transactions in place. This is also advantageous to me in case I decide to continue this project in the future and want to differentiate between different things that each need signing.

## Server-side (`server`) module

On a high level, I anticipate the server module to have a class diagram as below




This is a high level overview of what resources, schemas, and models I will need to be able to transfer all the data that I need over my API.

Resources are the objects that sit behind endpoints - they each implement various HTTP methods. The design of some less obvious resources is discussed below.

## API Endpoints and Resources

A lot of the work of the server is in serialising and deserializing objects / JSON. I will do this using the `marshmallow` library for Python. This requires that schema objects are set up with the same fields as the objects you want to serialise from / deserialize to.

This means that a lot of boilerplate code is needed, so I will not talk too much about that here as it is not particularly interesting, and does not prevent me from having a fully considered design of my problem.

I thought it to be more important to discuss the resources and endpoints that I would need to serve over my API. My resources are listed in the class diagrams above, and all serve an important purpose.

```
(Group, "/group/<int:id>", "/group")

(PrettyTransaction, "/transaction", "/transaction/<string:email>")

(User, "/user/<string:email>", "/user")

(UserGroupBridge, "/group/<int:id>/<string:email>", "/group/<string:email>")

(TransactionSigVerif, "/transaction/auth/<int:id>", "/transaction/auth/")

(Simplifier, "/simplify/<int:gid>")

(GroupDebt, "/group/debt/<int:id>")

(SignableTransaction, "/transaction/settle/<int:t_id>",
"/transaction/signable/<int:id>")
```

This shows the `Resource` child classes as shown in the class diagram above with their endpoints.

A lot of the above resources implement GET and POST, which are self-explanatory by design in most cases (i.e. GET "user/<string:email> will return user data for a given email"). I will discuss certain less obvious resources and endpoints.

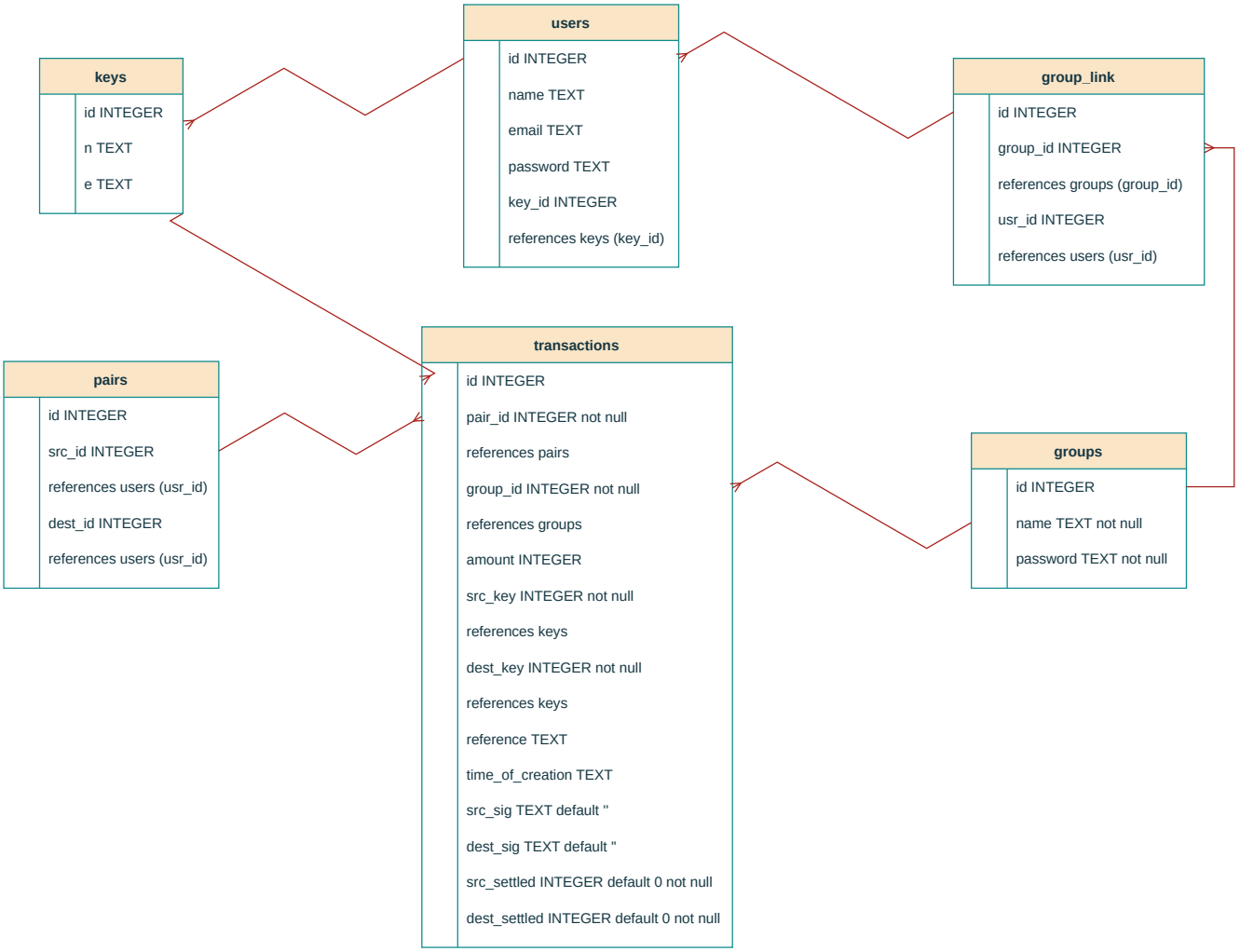
`PrettyTransaction` is a transaction object that is intended for being viewed on the front end by a user. It has people saved as emails as opposed to IDs, an association with a group, no keys involved, the time of creation, reference, and verification status. It also has the transaction ID. The POST method of pretty transaction is used to post new transactions to the database. This is because the details entered by the user about new transactions line up exactly with the pretty transaction schema. All processing such as adding public keys and user IDs is done by the server.

`UserGroupBridge` also warrants discussion. The resource implements POST and GET. POST will add a user to a group, and GET will get all groups associated with a given user.

`TransactionSigVerif` implements GET and PATCH. GET will verify a transaction, returning copy of verified transaction. PATCH will, upon receiving a signature, check that the signature is valid with data from the database (public key data, etc.), and if the signature is valid, the signature will be inserted into the database on the given transaction ID.

I intend to run the API using `flask` and `flask_restful`; two commonly used Python libraries for such purpose. During testing, I will run the server on `localhost`.

## Database Access




Here is the same database's DDL

```
create table groups
(
    id       INTEGER
        primary key autoincrement,
    name     TEXT not null,
    password TEXT not null
);

create unique index groups_group_id_uindex
    on groups (id);

create table keys
(
    id INTEGER
        primary key autoincrement,
    n  TEXT,
    e  TEXT
);

create table sqlite_master
(
    type     text,
    name     text,
    tbl_name text,
```

```
    rootpage int,
    sql      text
);

create table sqlite_sequence
(
    name,
    seq
);

create table users
(
    id       INTEGER
        primary key autoincrement,
    name     TEXT,
    email    TEXT,
    password TEXT,
    key_id   INTEGER
        references keys (key_id)
);

create table group_link
(
    id       INTEGER
        primary key autoincrement,
    group_id INTEGER
        references groups (group_id),
    usr_id   INTEGER
        references users (usr_id)
);

create table pairs
(
    id      INTEGER
        primary key autoincrement,
    src_id  INTEGER
        references users (usr_id),
    dest_id INTEGER
        references users (usr_id)
);

create unique index uniq_pair
    on pairs (src_id, dest_id);

create table transactions
(
    id               INTEGER
        primary key autoincrement,
    pair_id          INTEGER not null
        references pairs,
    group_id         INTEGER not null
        references groups,
    amount           INTEGER,
    src_key          INTEGER not null
        references keys,
```

```
    dest_key         INTEGER not null
        references keys,
    reference        TEXT,
    time_of_creation TEXT,
    src_sig          TEXT    default '',
    dest_sig         TEXT    default '',
    src_settled      INTEGER default 0 not null,
    dest_settled     INTEGER default 0 not null
);
```

(There is more on the structure of the database in the Evaluation section.)

The server module will handle all interactions with the sqlite3 database (entity relationship diagram below)

Since this is a fairly complex relational database system, I put some thought into the queries that I would use to select data. Below is an example of such a query.

```
SELECT transactions.id, group_id, amount, reference, time_of_creation,
u2.email, verified
FROM transactions
INNER JOIN pairs p on p.id = transactions.pair_id
INNER JOIN users u on u.id = p.src_id
INNER JOIN users u2 on u2.id = p.dest_id
WHERE transactions.src_settled = 0
OR transaction.dest_settled = 0
AND u.email = ?;
```

Here is an example query intended to retrieve rows of data that can be used to build a `models.PrettyTransaction` object. Data is fetched where the user's email provided is the source of the transaction - i.e. it will retrieve a user's outgoing transactions.

This statement is not group specific, but will return all transactions that have not been settled associated with a user's email.

An important thing to keep in mind when designing my SQL statements is ensuring data integrity. Thus, I will take care to protect against adding duplicate records, and ensure that data is consistent. For instance, a transaction should not be able to be signed by a user who is not part of that transaction

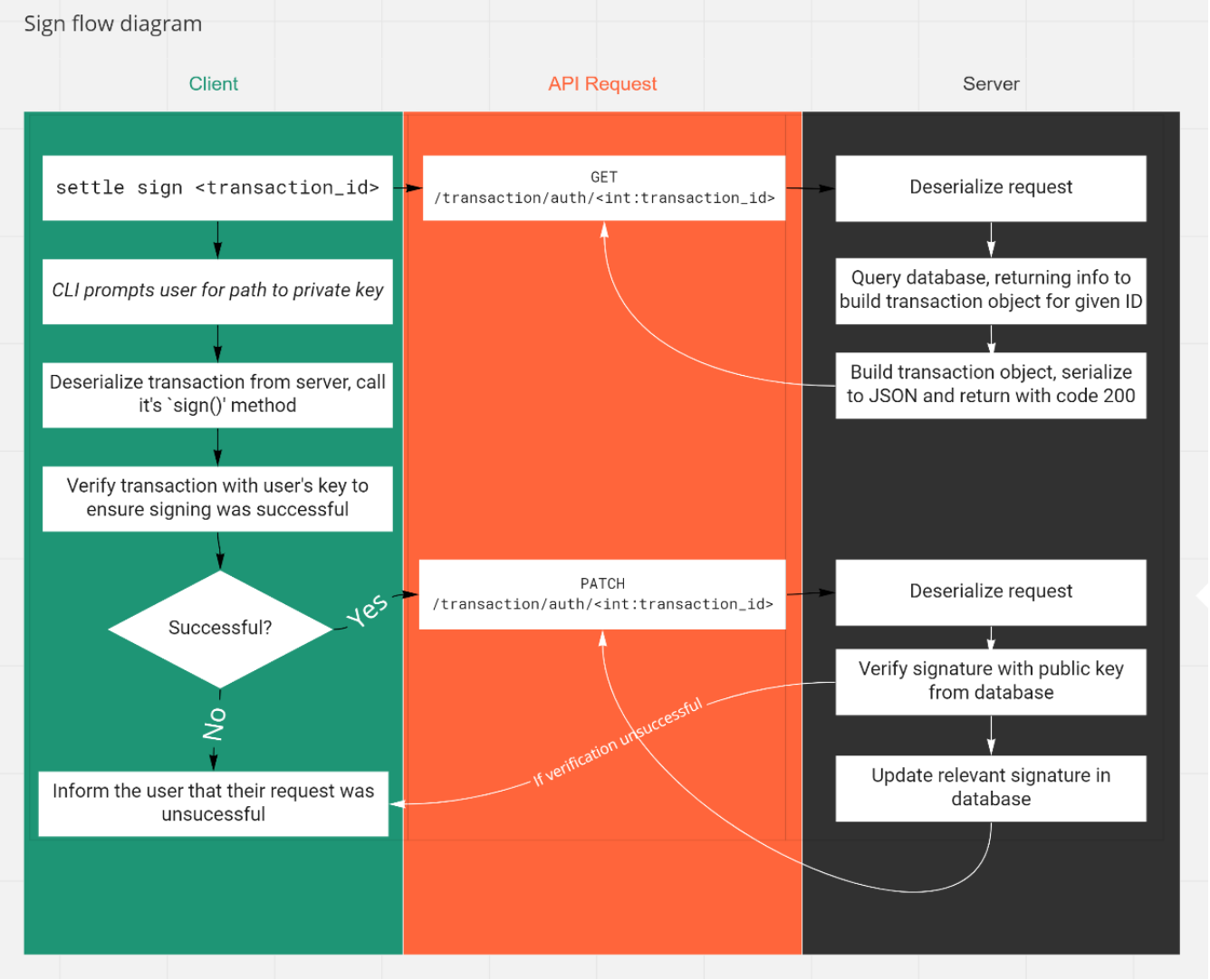
## Server Logic

The server is, of course, more than just an API and database access - it will carry out the vast majority of the data processing. In that sense, my client server models is effectively thick server, thin client.

As with the endpoints, a lot of the logic is minimal and self-explanatory by design. Thus, here I will discuss two of the more interesting processes that the server can be asked to do.

### Signing a transaction

Below is a swim lane diagram to aid my explanation of how a transaction is signed




This diagram shows the various processes that should run as a result of the user asking to sign a transaction.

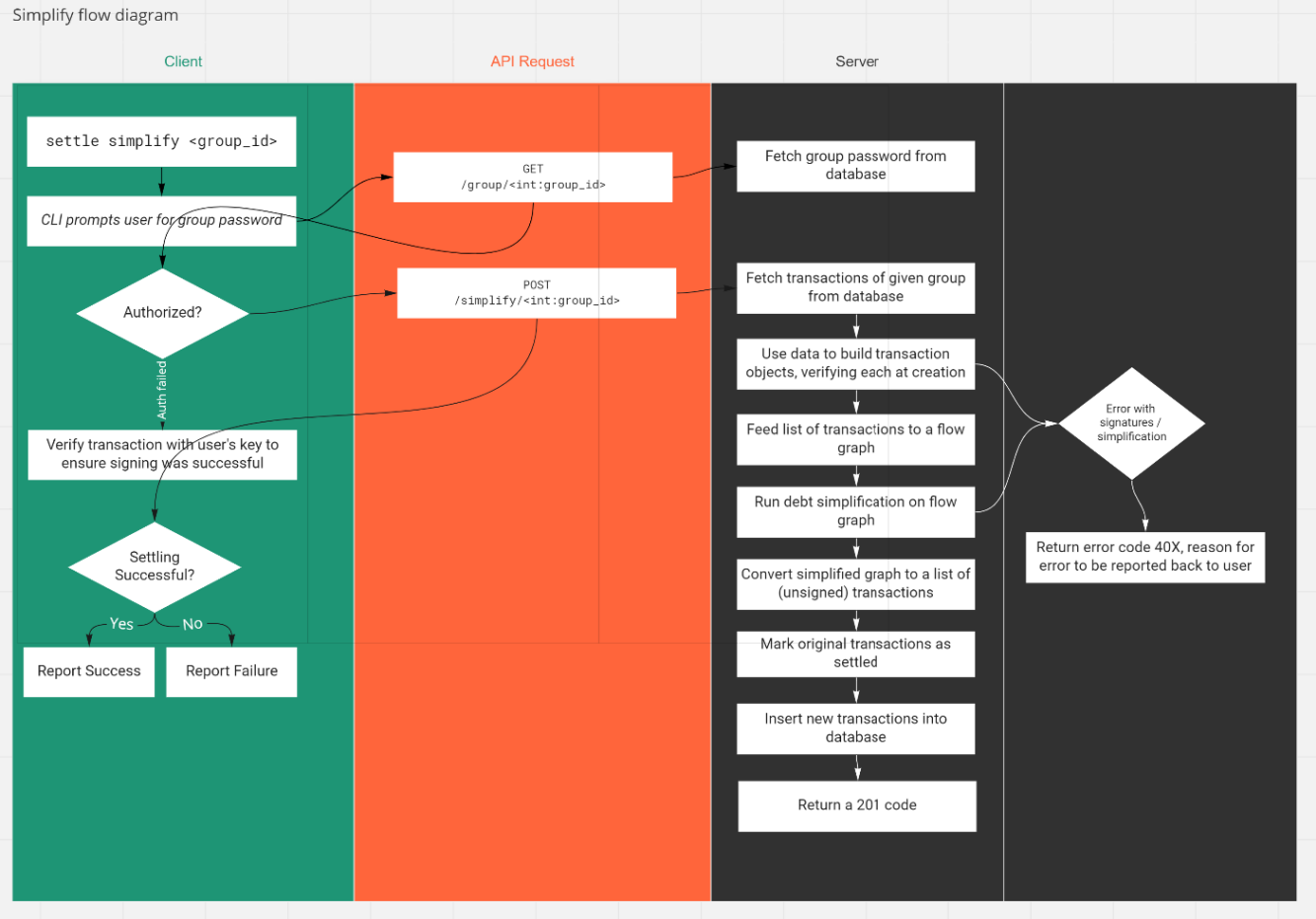
All client-server communication is done with JavaScript Object Notation (JSON) for this project.

For clarity, the diagram omits an argument to the `sign` command. `sign` also requires the user's email so that data can be kept in order in the database (more on this when the database design is discussed).

Here, 4 of the 5 modules are used. The `client` and `server` modules are clearly shown. The `transaction` module is used when constructing, signing and verifying transactions.

The `crypto` module is used in the `transaction` object, and provides the methods to be able to sign and verify the `transaction` object. It is also used to load keys on the client side (although this could be replaced with any PEM key loader).

### Simplifying a group of transactions




This is in many ways the most important process in the project. Upon being asked to settle a group, every part of the project will be used.

In the implementation, all I will need to do is query the database with a query such as

```
SELECT transactions.id, src_id, dest_id, src_sig, dest_sig,
 amount, reference, time_of_creation, group_id, k.e, k.n,
 k2.e, k2.n
FROM transactions
JOIN keys k on k.id = transactions.src_key
JOIN keys k2 on k2.id = transactions.dest_key
JOIN pairs p on p.id = transactions.pair_id
WHERE transactions.group_id = ?;
```

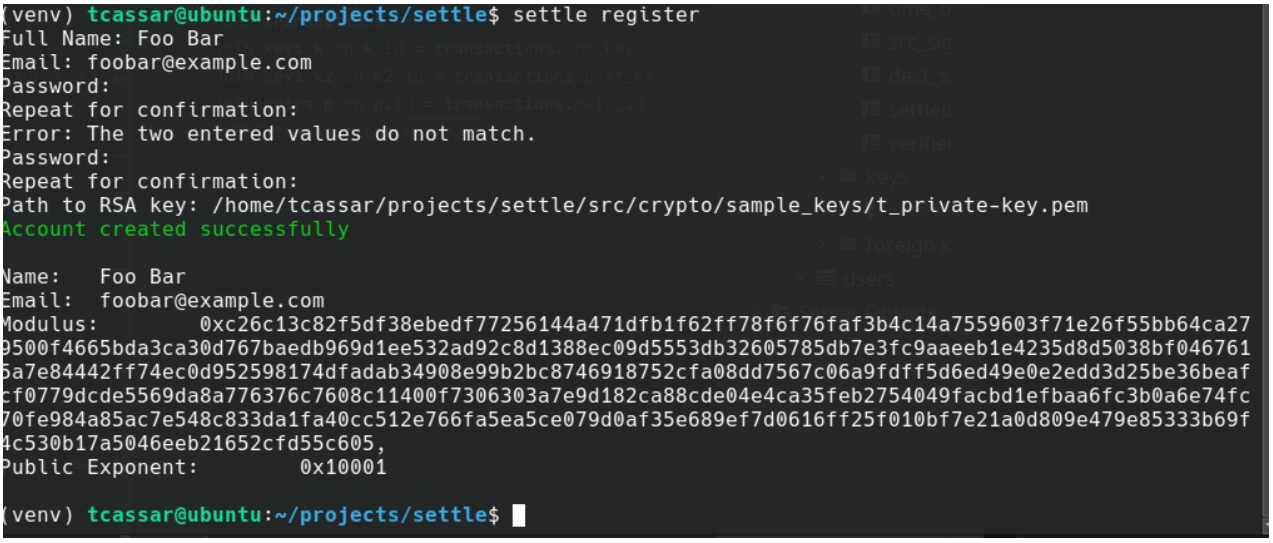
to get all the data needed to build a `transaction.ledger.Ledger` of `transaction.transaction.Transaction` objects. Once those objects are build, I call `ledger.simplify_ledger()`. This will then invoke all the logic discussed in the `simplify` module section, as well as handle the verification of signatures, as discussed in the `crypto` module section. This will be wrapped in a `try: ... except: ...` clause, and any errors will be returned with a 40X error code and reason for failure.

## Client-side (`client`) module

As aforementioned, the client is a thin client, meaning it does not have many responsibilities in the overarching structure of the program. Thus, this section will mainly be examples and mock-ups of the elements of Human-Computer Interaction.

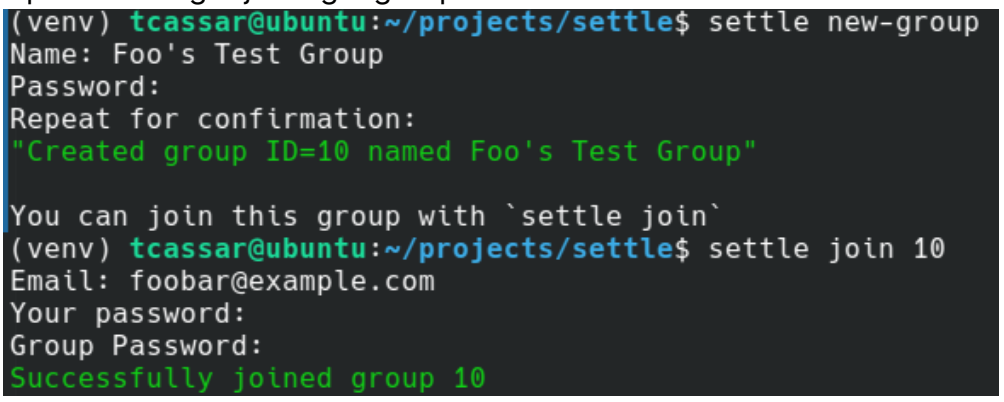
To show this, I will provide screenshots of an output to STDOUT of how I would like certain outputs to look. Here I will provide mainly ancillary outputs. and how I would like certain prompts to appear upon a command being run

- Upon a user registering a new account

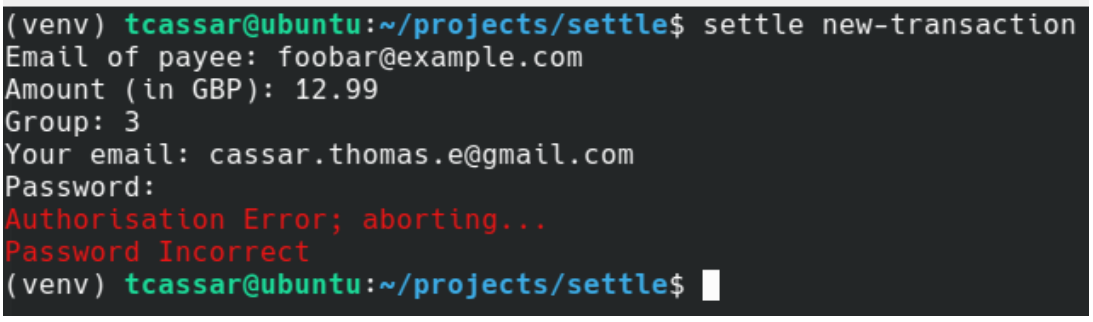



The program should prompt the user with the details that they need to enter to create their account. Password entering should be hidden, as it is commonly in CLIs.
Passwords should be confirmed through asking for confirmation, as above. If the passwords entered do not match, the program should ask the user re-enter their password. The program should report a failure if an invalid path to a key is given, as here

- Upon creating & joining a group




- An example of how a failed attempt at an action should look is



- Finally, viewing your existing transactions should look like this

```
(venv) tcassar@ubuntu:~/projects/settle$ settle show -t
Email: cassar.thomas.e@gmail.com

Your open transactions:


You owe keith@npl.com £12.99

Reference: scran
Agreed upon at 2022-03-19T16:53:37.720130
Verified

keith@npl.com owes you £12.99

Reference: scran2
Agreed upon at 2022-03-19T17:05:18.172690
Unverified
----------

You owe a total of £12.99
Your unverified totals => you are owed £12.99
(venv) tcassar@ubuntu:~/projects/settle$
```

This is not an exhaustive list, but is the general blueprint of how interaction should look. Every possible actions will end up looking like one of these above templates, be it something like simplifying or signing a transaction, which will just result in a confirmation, or seeing all the open transactions in the group, which will look the same as seeing all of your open transactions.

The full list of commands that I would like the user to be able to enter is below

```
Usage: settle [OPTIONS] COMMAND [ARGS]...

Options:
 --help  Show this message and exit.

Commands:
 join             Joins a group given an ID
 new-group        Creates a new group given a name and email
 new-transaction  Generates a new transaction
 register         Registers a new user
 show             Shows all of your open transactions / groups along...
 show-group       Shows the transactions in a group
 sign             Signs a transaction
 simplify         Simplifies debt of a group
 tick             Ticks off a transaction as settled up in the real world
 verify           Will verify a transaction if given a transaction ID or...
 whois            Shows the name, email and public key info given an email
```

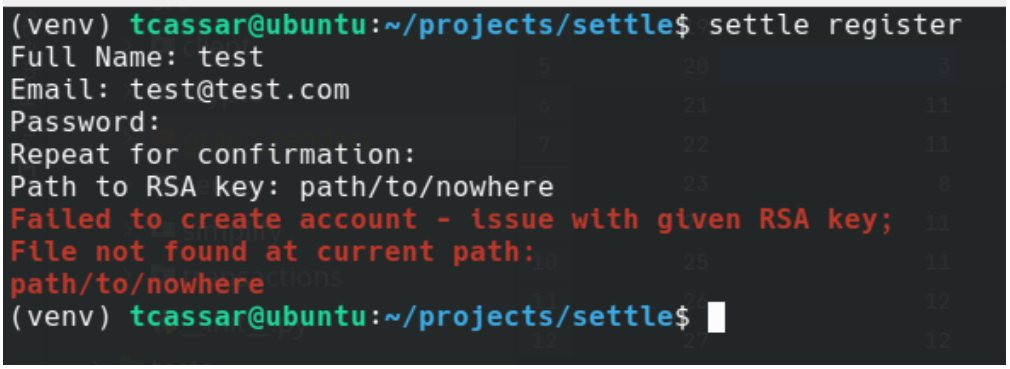
This text should also be displayed when the `--help` flag is called after any of the commands.

Note: the `whois` command may seem odd - it allows you to obtain information about other people. What makes a system like this work is the fact that it is trust free. The system is to be designed so that everyone can see everyone's public keys and everyone can see everyone's transactions.

Like this, there is nowhere to hide - you will always be held accountable for your transactions.

## Error Handling in the CLI

It is important that the user never experiences an unsightly looking crash message when an expected error in the program happens. For instance, if when registering for an account the user provides a path to a file that doesn't exist instead of their private key, they should not see the program crash. Instead, the program should prompt them that it could not complete the registration, because no file exists in that location. A mockup would look something like this.




---

1. based on a heuristic model ↩
2. shortest here referring to fewest edges traversed (not accounting for edge weight) ↩
3. ↩
4. Due to the heuristic nature of the model there will likely be multiple valid settled graphs ↩
5. the `__eq__` function in the base `Edge` class was generated by the `@dataclass` decorator. However, this would fail to differentiate residual edges due to the way that `dataclasses` generates dunder methods ↩