Single tx swap - a trustless multisignature escrow service - Specification

Daniel Oravec

9. 3. 2023

1 Overview

Single-tx-swap is a trustless escrow service built for the Ergo blockchain. A swap of assets between two parties happens in a single transaction. The service works as a GUI multisignature transaction builder for participants who would like to exchange some of their assets. Both parties involved have to sign the same transaction in order for the swap to take place.

Users that *single-tx-swap* aims at are mostly people who like trading NFTs (non-funbible tokens) for other NFTs and fungible tokens on Discord or other communication platforms. Discord channels of NFT projects usually have a dedicated channel where members can post their NFTs with information about what they would like to trade them for. Once two people interested in a trade agree on the contents of the trade, they need an escrow service in order for them to not get tricked by each other.

An escrow service utilizing a smart contract already exists. However, it has several shortcomings that we would like to solve.

- Two transactions are needed for the swap.
- Tokens can only be exchanged for one token in one swap.
- Users need to either understand the contract's code or trust the service provider.
- Users might be scared of locking their funds in a smart contract.

Therefore, we propose a solution using multisignature transactions that solves all of these problems. However, it requires users to perform the swap at the same time, which is a disadvantage against the smart contract solution. We only consider this disadvantage marginal for our use-case.

2 Requirements

There are two main user roles:

1. Trading session **host**
2. Trading session **guest**

Besides that, everyone has a **visitor** role. A browser extension wallet exists and can be downloaded in most browser web stores. We use this wallet for authentication and transaction signing. Anyone who wants to actively interact with our website has do download such wallet extension for Ergo blockchain. A user who connected their wallet using a button present on the website has an **authenticated user** role. Once two parties agree on the swap, they decide which one of them will be the session host. The host's role is to visit our website, connect their wallet and create a private trading session. After that, they will send a private invite link to the other party, who will be the session guest.

The host will wait until the guest joins the trading session and connects their wallet. Once that happens, The host is presented with wallet contents of both themselves and the guest. Wallet contents are fetched on our backend from one of the APIs (Ergo node API) provided by the Ergo ecosystem. There will be a separate display for the host's assets. and a separate display for the guest's assets. Both of these displays will look identical besides the heading. Assets in each display will be split into NFTs and fungible tokens and the user will be able to use toggle buttons to switch between those two views. No search functionality nor filtering will be available in the initial version, as the amount of tokens in an average user's wallet is sufficiently small. The host selects NFTs to swap from both wallet views. Besides that, they specify amounts of fungible tokens, including ERG (Ergo's native token) that they would like to swap too. While the host is building the swap, the guest is asked to wait.

After the host is done building the swap, they are asked to partially sign the transaction. By doing this, they allow spending their inputs in such a transaction. Inputs of the guest are still unsigned and therefore, the host can't submit the transaction to the network, as it would get rejected by the validator.

This partially signed transaction is then sent to the guest and the host is asked to wait until the guest also signs the transaction. Once the guest does that, the transaction is fully signed and is submitted to the network.

If the transaction is accepted by the Ergo blockchain, the swap is performed. If the transaction is rejected, nothing happens and participants do not lose any funds. If both participants carefully review the transaction in their wallet before signing it, there is no risk for users.

3 User stories

1. As a **visitor**, I can connect my wallet.
2. As an **authenticated user**, I can disconnect my wallet.
3. As an **authenticated user**, I can push a "start trading session" button, so that I am taken into a new private trading session page.
4. As a **visitor**, I can visit a link to a private trading session, so that I'm asked to connect my wallet before I'm automatically redirected to the trading session page.
5. As an **authenticated user**, I can join a private trading session as a **guest** by visiting a link received by the session **host**.
6. As a session **host**, I can send an invite link to someone and wait for them to join as a **guest**, so that I'm presented with wallet contents of myself and the **guest**.
7. As a session **host**, once I am presented with wallet contents of myself and the **guest**, I can select assets to swap and push a "swap" button, so that a swap transaction is built.
8. As a session **host**, once I pushed a "swap" button, I'm presented with a wallet popup asking me to partially sign a transaction, so that I can carefully review it and sign it if I agree.
9. As a session **host**, once I partially signed a swap transaction, I am asked to wait until the **guest** also signs the transaction.
10. As a session **host**, once the **guest** also signed the transaction, I'm presented with a success message with a link to the transaction on Ergo explorer.
11. As a **guest**, I can wait until I'm presented with a transaction partially signed by the **host** and then review it and finalize the signing process if I agree with it, so the transaction is submitted to the network.
12. As an **authenticated user**, I can visit my profile and fill my contact info there, so that this info will be visible on my profile for other **visitors**.
13. As a **visitor**, I can search for the contact info of anyone who once was an **authenticated user** based on a NFT they hold, so that I can contact them outside of the platform.
14. As an **authenticated user**, I can send a message to any other **authenticated user**.
15. As an **authenticated user**, I can create a message associated with any blockchain address, such that once an **authenticated user** with this unique blockchain address is seen, they can read the message.
16. As a **visitor**, I can visit any profile, so that I can view wallet contents, history and statistics of that profile.
17. As an **authenticated user**, I can visit any profile, so that I can view wallet contents, history and statistics of that profile.

4 Data model

A description of all needed database tables follows.

trading_sessions
This table holds information about trading sessions. While the host and the guest are taking turns in their respective steps, this table stores information about the current state of their trading session.
`id: INTEGER;`

- Surrogate key

`secret: STRING;`

- A unique 16-byte unpredictable ID of the session, hex encoded.

`host_addr: STRING;`

- A blockchain address of the session host.

`host_assets_json: JSON;`

- JSON of all assets held by the host's address at the time of creating the session. This data will be fetched from the Ergo node API by the BE based on `host_addr`, so that the host cannot lie about their assets. The host is interested in not performing any transactions until the trading session is finished. If assets held by the host change while the trading session is not finished, we will not detect that, but the transaction might fail once it is submitted to the blockchain.

`host_nano_erg: BIGINT;`

- Amount of ERG held by the host times 10^9, similar to `host_assets_json`.

`guest_addr: STRING;`

- A blockchain address of the session guest.

`guest_assets_json: JSON;`

- Like `host_assets_json`, but for the guest.

`guest_nano_erg: BIGINT;`

- Like `host_nano_erg`, but for the guest.

`unsigned_tx: JSON;`

- A swap transaction with no witnesses yet. As the transaction confirmation time on the Ergo blockchain is usually several minutes, we are not submitting the transaction again in case it fails. If we did that, it could possibly be confusing to users. Therefore, once the transaction is submitted, we will only display a message that users need to start a new trading session in case the transaction fails. As we are only interested in text representation of the transaction and not working with it in any way other then sending it to clients, it makes more sense for us to keep it in this table.

`unsigned_tx_added_on: DATE;`
`signed_inputs_host: JSON;`

- A list of signed host's transaction inputs. This will be used to assemble the final signed transaction. As the transaction is being assembled on the client-side, we are not working with the structure of these inputs at all. Therefore, it is simpler (less code needed) to store signed inputs this way instead of having a separate table for them. We could also store them as a string (hex-encoded serialized input), but we use JSON instead to conform to the format used by the Ergo node API.

`tx_input_indices_host: ARRAY(INTEGER);`

- Which transaction inputs are host's. This is so that we can place `signed_inputs_host` to correct places in the final transaction's inputs.

`tx_input_indices_guest: ARRAY(INTEGER);`

- Like `tx_input_indices_host`, but for the guest.

`tx_id: STRING;`

- Transaction id of the swap transaction (a hash of the serialized transaction).

`submitted_at: DATE;`
`created_at: DATE;`

users
Once someone connects their wallet, they automatically become a user.
`address: STRING`

- PK, blockchain address, not home address.

`username: STRING; email: STRING; discord: STRING;`

- Discord username

`twitter: STRING;`

- Twitter username

`allow_messages: BOOL;`

- Whether receiving messages is allowed.

assets
We want to be able to mark some assets as verified (authentic), as anyone can create an asset with any name. The only difference in that case is the `token_id`, which is unique, but users might not notice the difference in `token_id`. Displaying an alert on unverified assets could therefore avoid scams.
`token_id: STRING;`

- PK, unique token ID on the blockchain.

`decimals: INTEGER;`

- To keep all computations in whole numbers, each token defines the number of decimal places it uses, while only using integers for the representation of the amount. For example, if a token wants to be divisible to thousandths (and not to smaller units), it would set it's decimals to 3. Then, if the amount of this token a specific blockchain address holds is 123456 as reported by the Ergo node, the actual human-readable amount is 123.456.

`is_verified: BOOL;`

- Whether the asset is authentic.

user_session_stats
Statistics about trading sessions the user took part in. These data will be displayed on the profile page. We precompute this data mainly because the `trading_sessions` table will often be cleaned from old sessions.
`user_address: STRING;`

- PK, blockchain address of the respective user.

`sessions_hosted: INTEGER;`

- Number of sessions hosted by this user.

`sessions_visited: INTEGER;`

- Number of sessions where this user was a guest.

user_asset_stats
Statistics about how much of a specific token a user traded on the platform in the past. `user_address: STRING;`

- Blockchain address of the respective user.

`token_id: STRING;`

- Asset's unique ID on the blockchain.

```
amount_bought: BIGINT;
amount_sold: BIGINT;
```

follows
A relationship between two users. A user can follow many other users. `user_address: STRING; followed_address: STRING;`

- Address that is followed by the user specified in `user_address`.

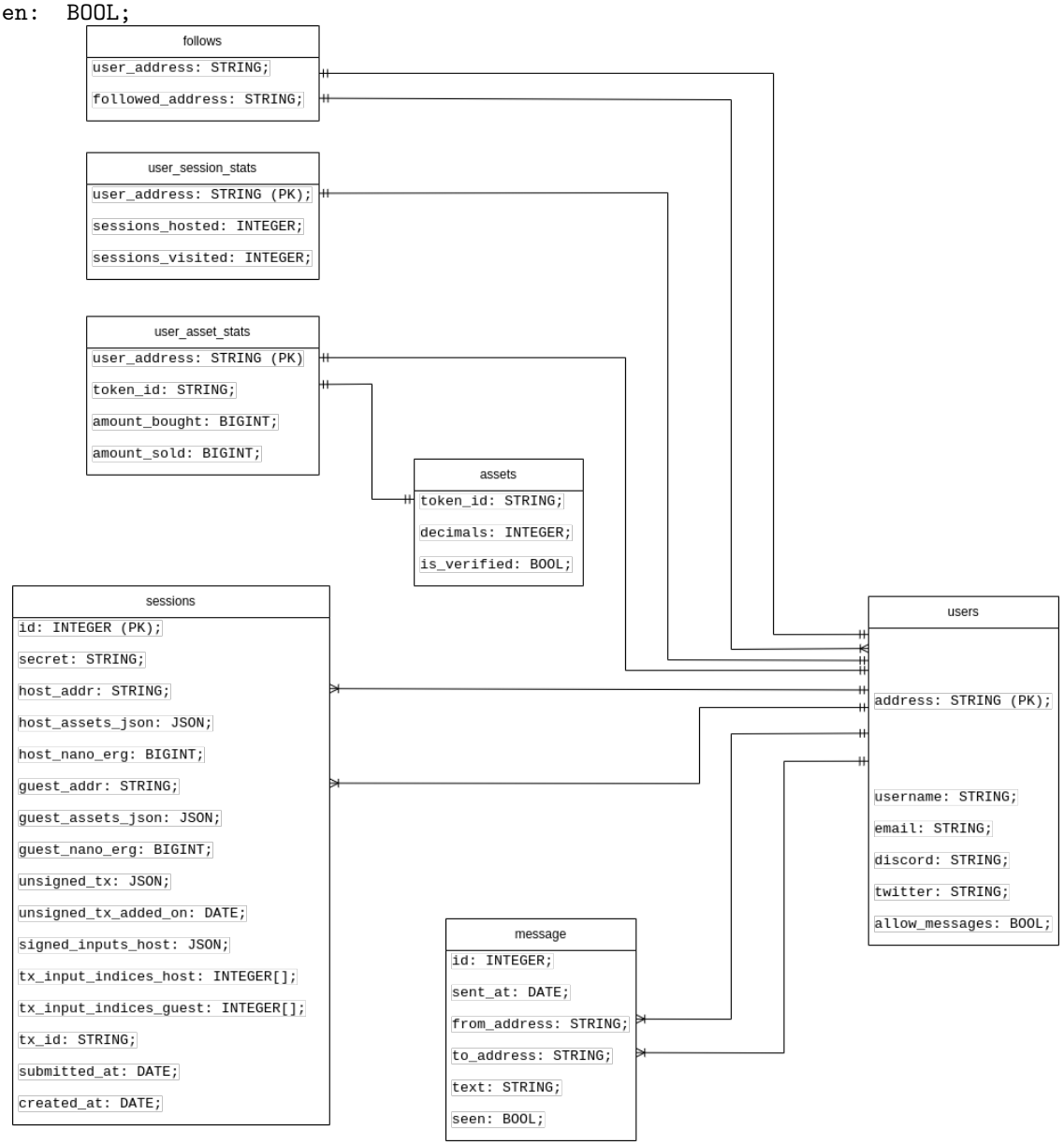
messages
`id: INTEGER;`
`sent_at: DATE;`
`from_address: STRING;`

- Blockchain address of the sender.

`to_address: STRING;`

- Blockchain address of the receiver.

`text: STRING;`
`seen: BOOL;`



5 Technological requirements

This will be a Typescript Next.js(13.1.6) React(18.2.0) application using styled-components(5.3.6) for styling. For state management, such as handling wallets, Zustand(4.3.2) will be used.

Transactions will be built using FleetSDK(>=0.1.0-alpha19). In case more complicated computations will be needed, ergo-lib will be used.

As this is a Next.js app, we chose Vercel for hosting it.

The backend will be written in Typescript (4.9.5) using Express.js(4.18.2). FleetSDK(>=0.1.0-alpha19) will be used for validating transactions received from users. Our backend will provide a REST API.

Postgres(13.7) is a database system of our choice.

The backend will be hosted on AWS Elastic Beanstalk with AWS Codebuild for CI/CD.

Major browsers such as Chrome (+Brave) and Firefox should be supported.

6 Time plan

25. 3. 2023: Working POC

- Homepage and swap UI design in Figma
- Wallet connection
- Implement footer, navbar
- Theming (dark, light)
- `[POST] /tx/register`
- `[GET] /tx`
- `[GET] /tx/partial`
- `[POST] /tx/partial/register`
- `[POST] /session/create`
- `[POST] /session/enter`
- Landing page implementations (with FAQs)
- Swap UI and logic implementation
- Parse assets according to EIP-0004 to get image links and display images
- Add SVG icons for verified fungible tokens
- Setup domains, Vercel, Elastic Beanstalk and Codebuild

1. 4. 2023: Profile section and database improvements

- Design and partially implement profile page
- List of assets (no history yet)
- Edit profile
- Add contact info (no profile picture yet)
- Use migrations

11. 4. 2023: Finalize profile section, add messaging

- Implement messaging system

- Design and implement a page for viewing messages
- Also delete, mark as unread
- Add notification bell to navbar for messages

30. 4. 2023: Statistics

- Implement statistics (sessions created, visited, amount traded too if possible) in profile section
- Implement swap history in profile section
- [TBD] Global rankings page

2. 5. 2023: Add search users functionality

- Design and implement a page for searching a user by tokenID that they hold
- Contact info will be displayed if such user is found