FT5004 Project: Tokenized Stock Slam Game

Group Participants: Yuan Han (A0107507L), Michael Hellman (A254412M)
Github repo: https://github.com/ag3nt3154/Tokenized-Stock-Slam-Game

Introduction

In this project, we present a tokenized version of the popular board game Stock Slam, where players compete against each other by trading fictional shares. In this project, players will place their ETH into a betting pool and obtain payouts based on the outcome of the game.

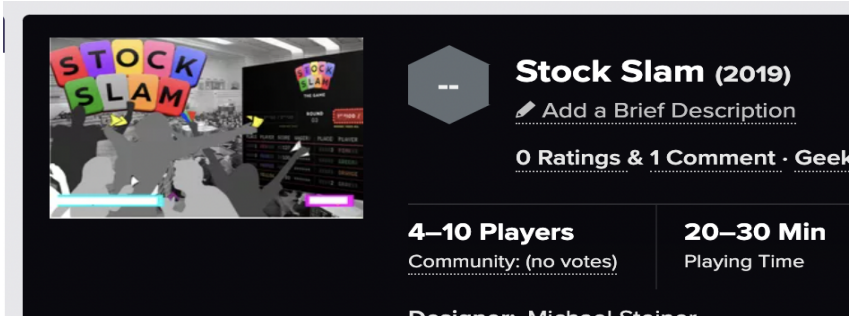


Figure 1: Stock Slam, the boardgame

Game Rules

The rules of the game are defined as follows:

1. There are 5 players.
2. At the start of the game, each player pays 1 ETH to begin the game with 1000 StockTokens (ST) and 2 shares of each stock (5 stocks in total).
3. During the game, players can trade shares with each other. The trading can be done on an order book or bilaterally over-the-counter (OTC).
4. At the end of the game, 1 stock will be considered “in-the-money” (ITM) and worth 100 ST for each share. All other stocks will be considered worthless. 100 ST will be transferred to each holder of ITM shares.
5. The total net worth (in ST) of each trader will determine the payout from the original pool of 5 ETH.

Stock Slam is a game that builds upon aspects of market-making, negotiation, and risk management.

For example:

1. If a player believes that stock 1 is worth 100 ST, he might be willing to buy at 99 ST and sell at 101 ST for each share of stock 1. The player can then earn 2 ST per round trip.
2. Players have to optimize for how they want to end the game based on their networth and risk appetite, e.g. by keeping all cash, keeping a spread of stocks, or keeping everything in 1 stock.

Contractual Relationships

This tokenized implementation of Stock Slam is made up of six contracts: StockToken, Stock, StockMarket, Player, Payout, and Play.

StockToken defines an ERC20 token, while Stock and StockMarket handle trading and asset distribution respectively. These processes are trustless, meaning there is no need for a middleman (clearing house e.g. DTCC) to facilitate trading.

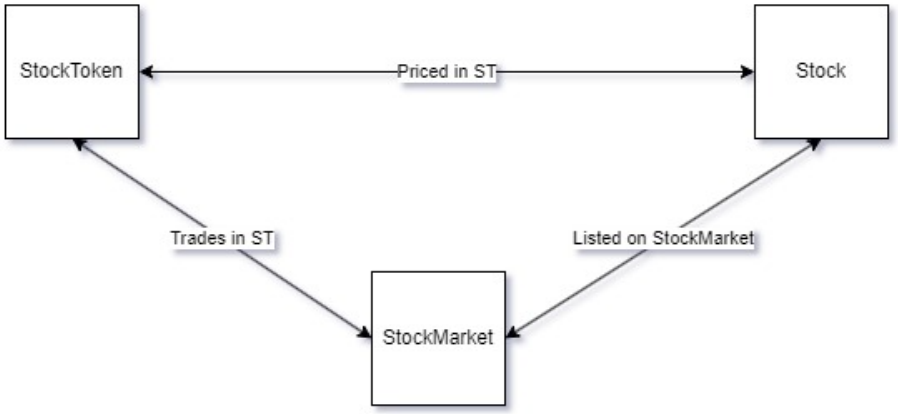


Figure 2: Relationship between Stock, StockToken, and StockMarket contracts

Player.sol describes a balance of Eth and StockTokens each player has, their trade logs, their active games. Payout.sol experiments with changes to how the game works and includes unique web3 features like airdrops and consensus voting. Finally, Play.sol serves an ad-hoc main file for implementing and invoking a game, which typically lasts five rounds.

StockToken.sol

Stock Token (ST) is an ERC 20 token that players trade in during the course of a game. We chose to issue our own token for a few advantages:

1. We can control the supply of currency within each game. Players cannot bring in external advantages, e.g. by being well-capitalized.
2. We can easily define payout mechanisms at the end of the game by simply swapping out ST for ETH.

Within the game, all trading is executed in ST. The figure below demonstrates the flow of ST at the start and end of the game (excluding trading occurring due the course of the game between players).

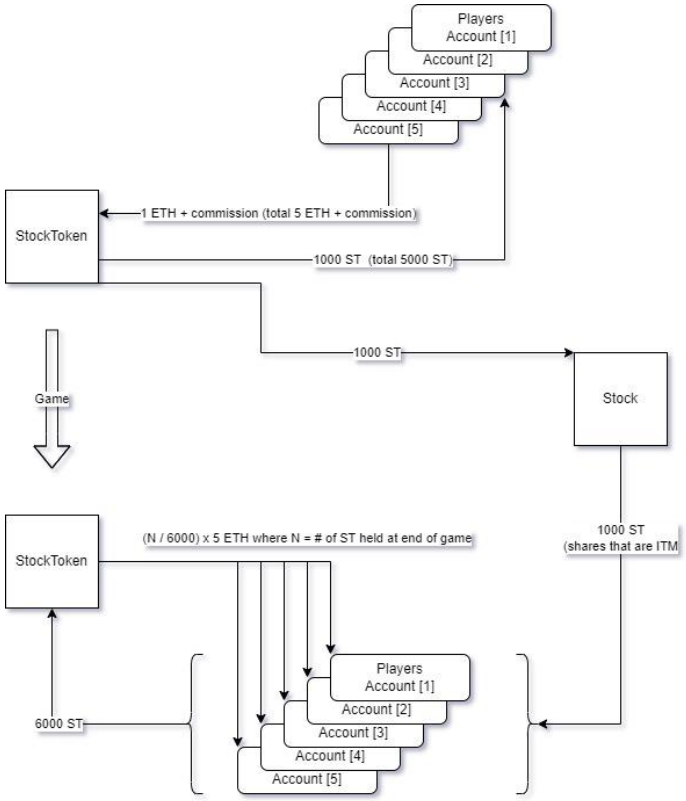


Figure 3: Movement of ST at the start and end of the game

To facilitate the game, beside the standard functions defined in ERC 20 protocol, we added another 2 functions to exchange for and redeem ST at the start and end of the game respectively.

```
function topUpST() public payable {
    require(msg.value == 1 ether + commission_fee, "Please pay 1 ether + commission fee");
    require(balanceOf(msg.sender) == 0, "Sender already owns ST");
    require(total_supply > 0, "Too many players");

    total_supply -= 1000;
    accounts[msg.sender] = 1000;

    id_to_address[player_count] = msg.sender;

    address addr = id_to_address[player_count];

    require(balanceOf(addr) == 1000, 'account balance');

    player_count++;

    // id = 0 is the game master
    payable(id_to_address[0]).transfer(msg.value);
}
```

Figure 4: Enter the game and obtain ST token.

```
function distributeEther() public payable{
    // called by game master
    require(msg.value == 5 ether, "Pay 5 ether to settle the game");
    require(total_supply == 0, "There must be a game ongoing");

    uint256 totalST = 6000;
    uint256 stBalance;
    uint256 amountToSend;

    for (uint256 i = 1; i <= 5; i ++) {
        require(id_to_address[i] != address(0), 'valid address');
        stBalance = balanceOf(id_to_address[i]);
        amountToSend = msg.value * stBalance / totalST ;
        payable(id_to_address[i]).transfer(amountToSend);
    }
}
```

Figure 5: Distribute winnings at the end of the game.

Since it was difficult to execute the game and track the amount of the ST and ETH possessed by each player, we added a temporary "game-master" role to distribute ST and ETH at the start and end of the game.

In the basic version of the game, the payout is determined in proportion to the amount of ST held by each player at the end of the game. For example, if player 1 holds 900 ST in cash and 1 ITM share, he will have 1000 ST net worth in total. Given that the prize pool is 5 ETH representing 6000 ST in net worth, player 1 will get:

$$Payout = \frac{1000ST}{6000ST} \times 5ETH = 0.833ETH$$

Stock.sol

The contract stock.sol defines the creation and functionalities of shares and stocks. Each share is structured as an object with a *stockId* and *shareId* that identifies the share.

	Stock 1	Stock 2
Share 1	stockId: 1 shareId: 1	
Share 2		

Figure 6: Abstraction of individual shares

At the start of the game, the stock contract runs a loop to create the shares and distributes 2 shares of each stock to every player.

```
struct Share {
    address prevOwner;
    address owner;
    uint256 askPrice;
}

mapping(uint256 => Share[]) private stocks;
uint256 public constant STOCK_COUNT = 5;
uint256 public constant SHARE_COUNT = 10;
StockToken private st;

constructor(address stAddress, address[] memory playerAddresses) {
    require(playerAddresses.length == STOCK_COUNT, "Must provide 5 player addresses");
    st = StockToken(stAddress);
    for (uint256 i = 0; i < STOCK_COUNT; i++) {
        for (uint256 j = 0; j < SHARE_COUNT / 2; j++) {
            uint256 shareIndex = j;
            stocks[i].push(Share(address(this), playerAddresses[shareIndex], 6000));
            stocks[i].push(Share(address(this), playerAddresses[shareIndex], 6000));
        }
    }
}
```

Figure 7: Constructor for stocks and shares

The ask price is the price that the owner is asking for the share. This is the price that the share will be traded at during the course of the game. By default, the ask price is set to 6000 ST, since this represents the entire supply of ST and no player will be able to accumulate this amount. This is equivalent to the owner setting the share as ‘not-for-sale’.

Other functions that the owner has access to are:

- changeAskprice
    - The owner may change the ask price of the share.
- sellShare
    - The owner may sell the share for the ask price.
    - This is the function used for OTC trading.
- transfer
    - Owner transfers ownership to another address without payment.
    - This function is called when listing the share on the market.

StockMarket.sol

The stock market contract describes the order book mechanism for this game. The order book is a place where ask prices of shares listed on the market are stored. Players looking to buy shares can then go to the order book and buy a share at the ask price.

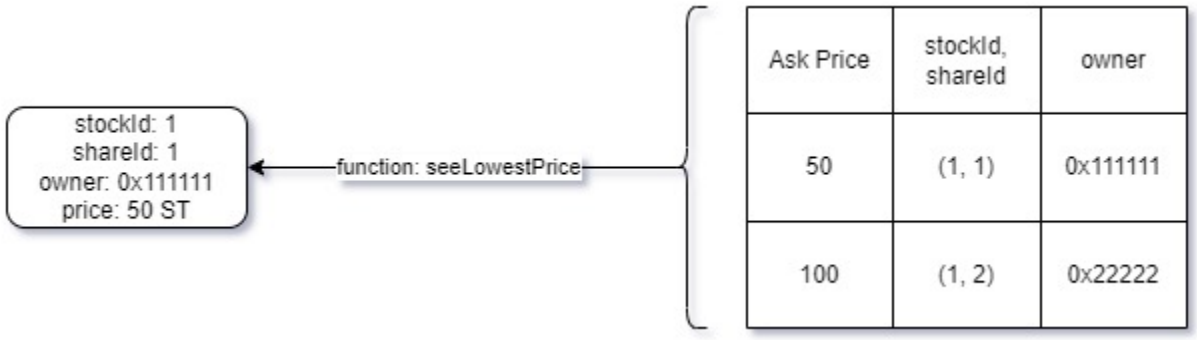


Figure 8: Orderbook for stock 1

From the figure, we see that an interested buyer can call the function seeLowestPrice to view the share listed with the lowest ask price on the orderbook.

Other functions of the stock market contract include:

- List
    - Owner of the share transfers the share and lists the share at the ask price of the share.
- Unlist
    - Stock market deletes the listing and transfers the share back to the owner.
- Buy
    - Buyer purchases the share with a specified stockId and shareId.

    - This may not be the lowest-priced share. To purchase the lowest-priced share, a buyer should call seeLowestPrice to find the stockId and shareId of the lowest-priced share, and then call the buy function.
- Refresh
    - This function refreshes the order book. For example, if a listed share's owner changes the ask price while the share is listed, he can call refresh so that the price change is reflected in the orderbook.

Presently, the stock market represents a vastly simplified order book mechanism. Notably, there is no price matching mechanism to match the best bids and offers to each other and it is reliant on buyers to meet asking prices for transactions to take place.

Trading Mechanisms

As previously mentioned, there are two mechanisms for trading between players. A diagram of the mechanisms are displayed here for clarity:

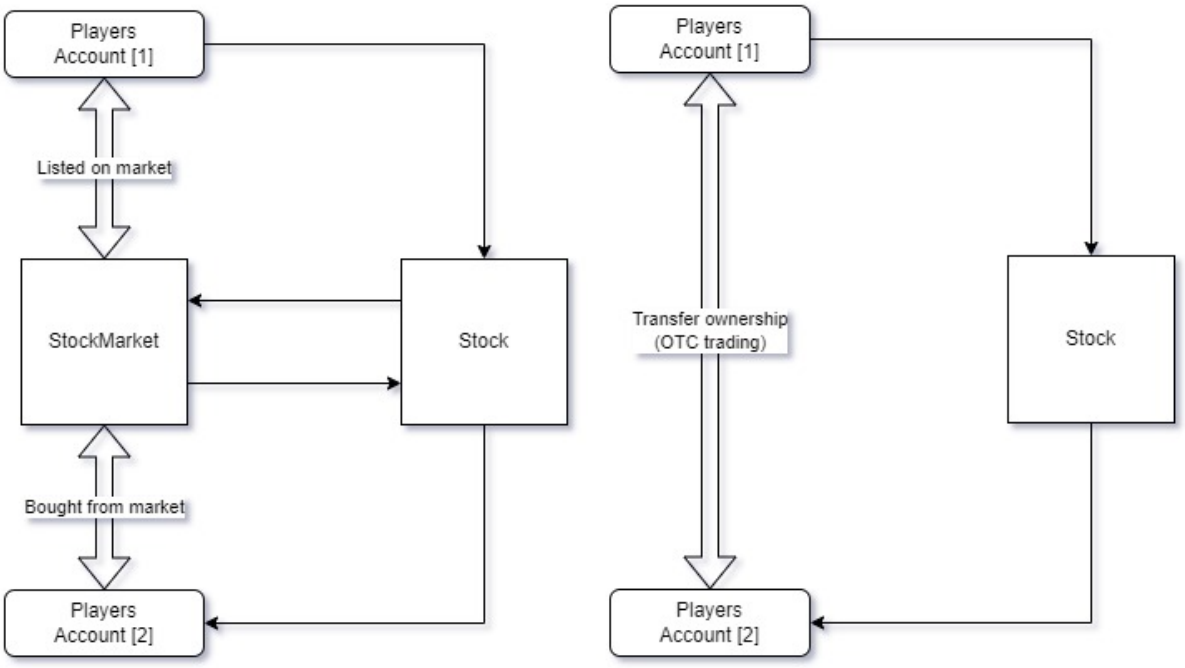


Figure 8: Trading on the stock market contract v.s. trading OTC

Player.sol

Each game participant is a Player. Player.sol describes:

- Player balance
- Individual Permissions of Assets
- Trade Log

Testing:

- Test the creation of a new player.
- Test the player's attributes (name, balance, holdings, etc.).
- Test the deposit and withdrawal of funds.
- Test the player's interaction with the StockMarket.sol contract (buying and selling stocks).
- Test access control and modifiers

Code Snippet/Modifier Properties:

```
pragma solidity ^0.8.0;

import "./StockToken.sol";

contract Player {
    struct AssetPermission {
        uint256 assetId;
        bool isPermitted;
    }

    struct TradeLogEntry {
        uint256 stockId;
        uint256 shareId;
```

Payout.sol

In the primary game description the payout rules: *The total networth of each trader (cash + shares) will determine the payout from the pool of 5 eth.* Payout.sol is a contract that extends the StockMarket.sol. Here are alternative payout rules to add more game modes:

- Performance-based bonuses. Performance-based bonuses refer to compounding the success of an individual round. These bonuses are randomly allocated (as defined in the implementation).
- Round-based progressive rewards - a determined round over round behavior
- Random “airdrop” events

- Voting-based rewards (market allocation rule changes from the players' consensus)

Testing:

- Test the implementation of performance-based bonuses.
    - Test the bonus calculation method.
- Test the implementation of round-based progressive rewards.
    - Test the reward calculation method.
- Test the implementation of random "airdrop" events.
    - Test the distribution of airdropped tokens to selected players.
- Test the implementation of voting-based rewards.
- Test the voting process.
    - Test the application of market allocation rule changes based on players' consensus

***Full Implementation Outside of Scope**

Play.sol

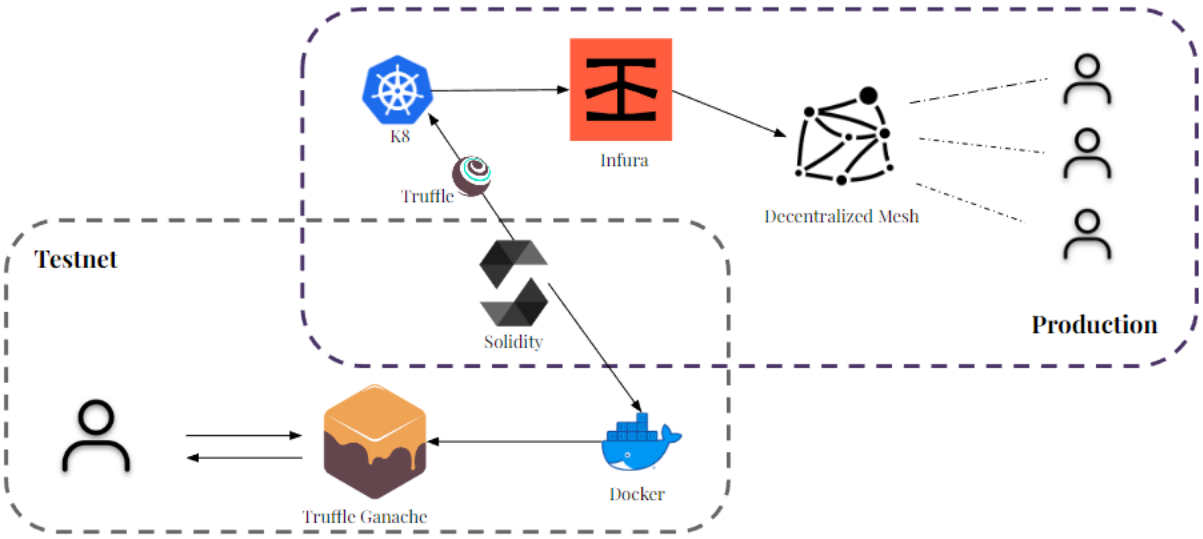
Play.sol serves as an analog to main programs of most other languages. This contract implements and invokes the other contracts. Locally this file is used for the Testnet (development)

- Test the overall functionality of the stock market game.
- Test the interaction between different players and the stock market.
- Test the end-to-end process of buying, holding, and selling stocks.
- Test edge cases, such as insufficient funds or exceeding token supply.
- Test the game's rules and conditions.

Code Snippet/Modifier Properties:

```
pragma solidity ^0.8.0;

import "./Payout.sol";
import "./Player.sol";

contract Play | {
    mapping(address => Player) public players;
    uint256 public gameId;
    uint256 public currentRound;

    // Events
    event GameCreated(uint256 gameId);
    event PlayerJoined(address indexed player);
    event NewRound(uint256 round);

    constructor(StockToken _stockToken, Stock _stock) Payout(_stockToken, _stock) {}

    // Create a new game
    function createGame() public {
        gameId++;
        currentRound = 0;
```

Application Architecture



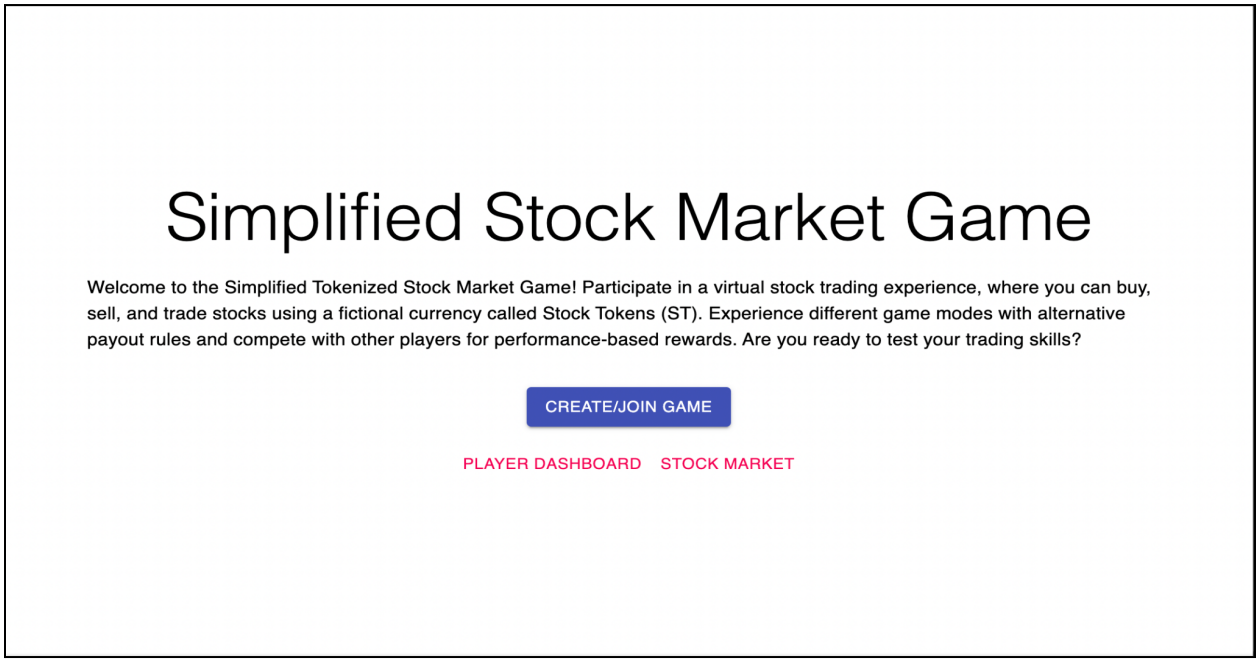
Testnet: Monolithic Development Instance, **Production:** Production Grade Implementation

To Extend Monolithic/Local Implementation of Token

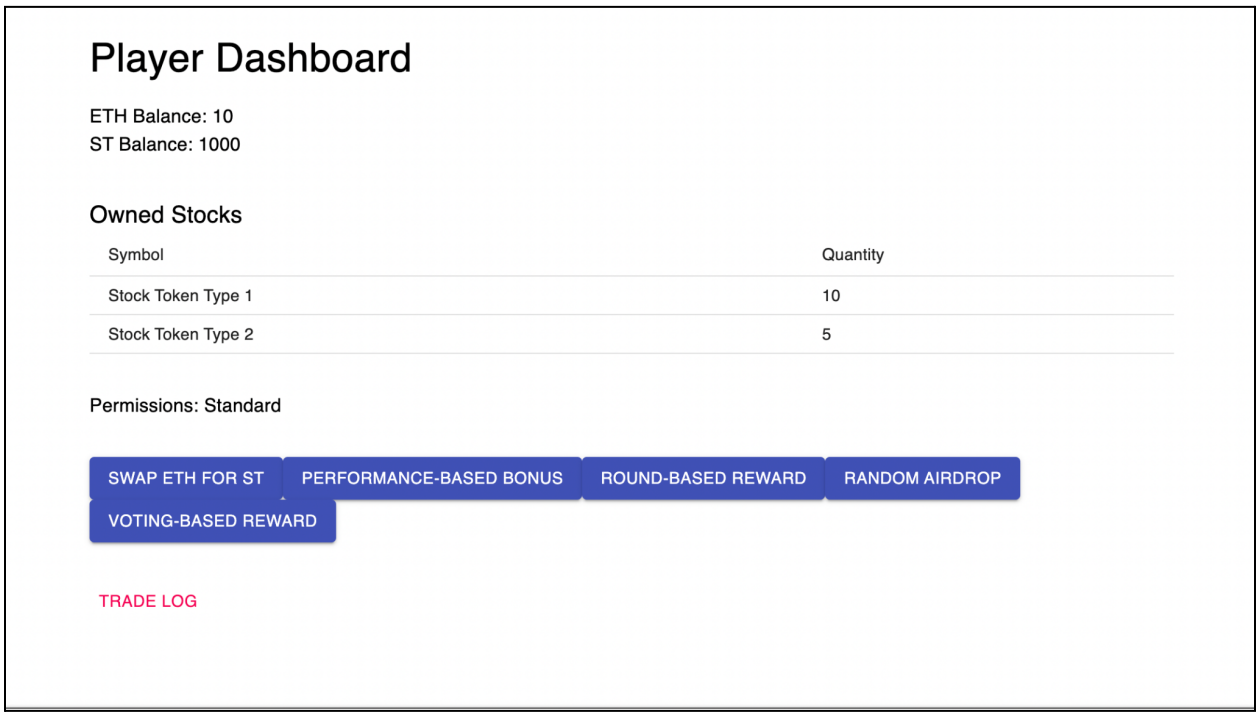
- Contract Deployment and Integration

    - Test the interaction between the front-end application and the deployed contracts using Web3.js or an equivalent library.
- Production Environment Setup
    - Set up a Kubernetes cluster and configure the necessary nodes and services.
    - Test the application's performance under varying loads to ensure it can handle increased demand.
- Monitoring and Logging
    - Test the monitoring and logging infrastructure to ensure it is functioning correctly.
- End-to-End Testing
    - Test the full user experience, including account creation, depositing and withdrawing funds, buying and selling stocks, and receiving payouts.
    - Test edge cases and unexpected user behavior to ensure the application can handle a wide range of inputs and scenarios.
- Final Deployment
    - Deploy the smart contracts to the main Ethereum network using Infura.

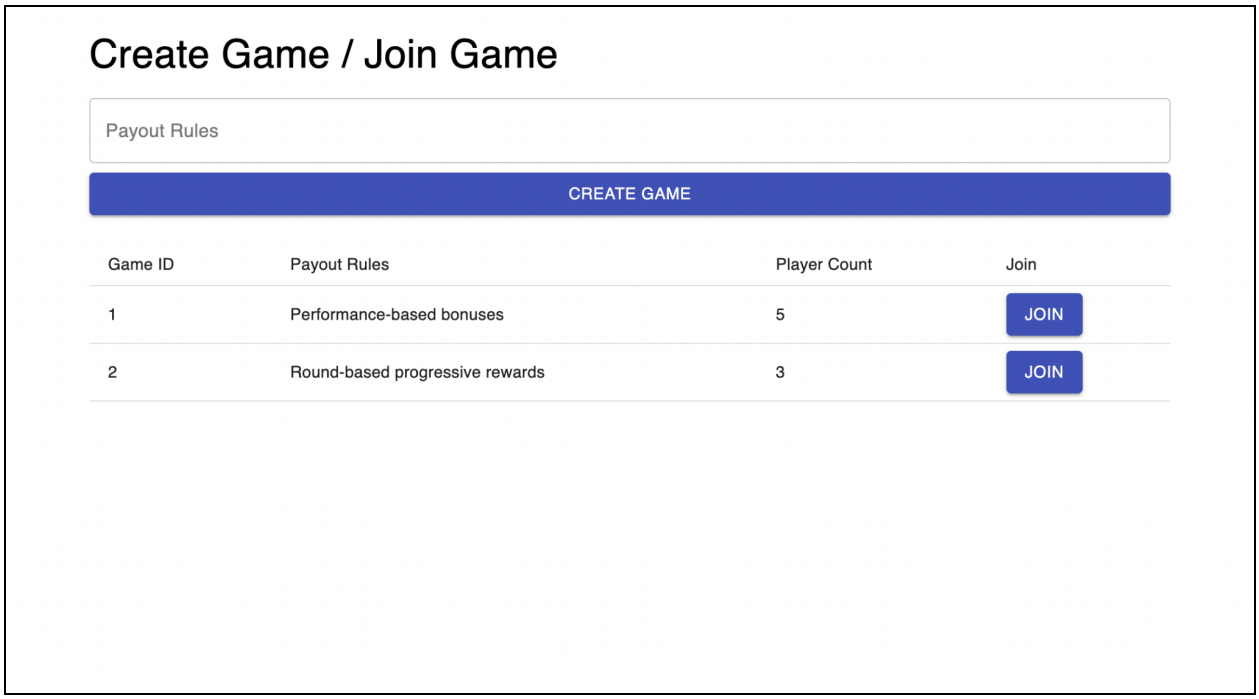
UI



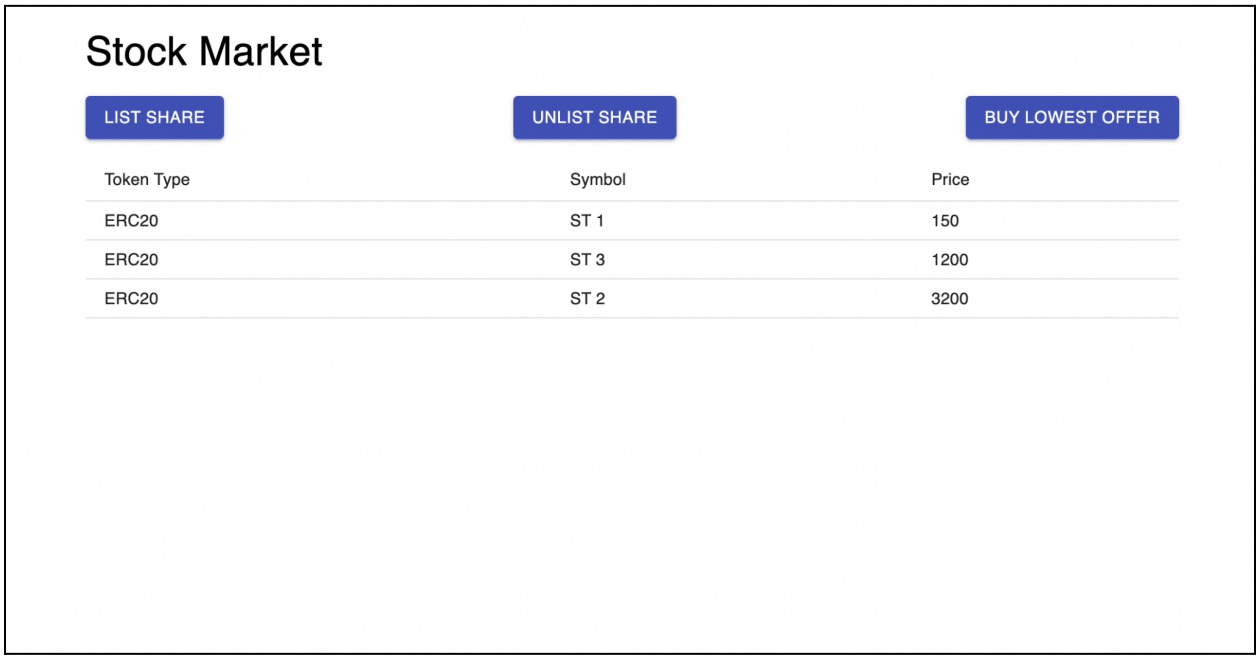
Main/Landing page



Player dashboard, showing shares owned and various metrics



Create/Join game page



Stock market interface

Trade Log

Stock Symbol

Date mm/dd/yyyy

All
Buy
Sell

Date	Symbol	Type	Quantity	ST Price
2023-01-01	Stock Token Type 1	Buy	5	150
2023-01-10	Stock Token Type 3	Sell	3	160
2023-01-15	Stock Token Type 2	Buy	2	1200

References

- *Stock slam*. BoardGameGeek. (n.d.). Retrieved April 22, 2023, from https://boardgamegeek.com/boardgame/270729/stock-slam
- https://docs.soliditylang.org/en/v0.8.19/
- https://trufflesuite.com/ganache/
- https://www.docker.com/
- https://kubernetes.io/
- https://remix.ethereum.org/
- https://www.infura.io/