Secret Token Battleships

Isaac Ure

Abstract

This document outlines the design and implementation of a battleships game hosted on the secret network using smart contracts. The game is designed so that one user can place ships, and many users can all fire shots against them. Tokens are effectively integrated into the game to add an element of gambling thrill. The implementation makes efforts to control user flows and keep the relevant states visible to the user. Future work may be done to give the game a richer feature set and to give users more incentive to play.

Problem Statement and Motivation

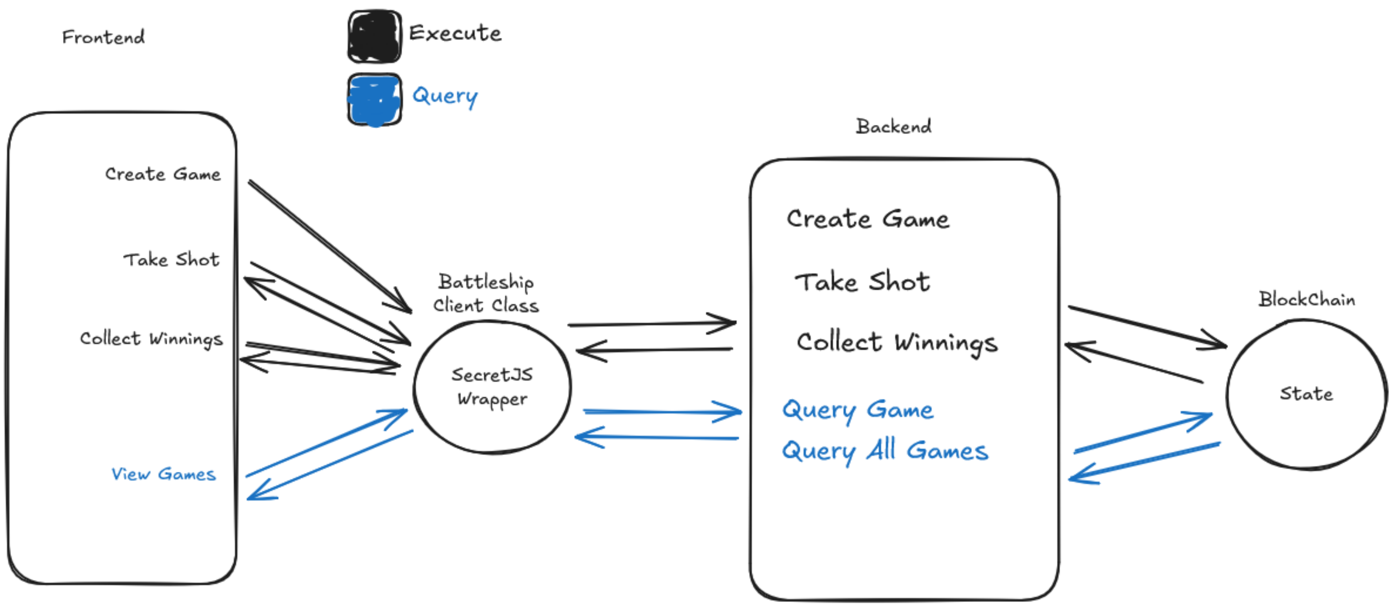
Battleships is a well-known game with many implementations and variants. The physical version is fun to play and a good way to socialize, however the board is difficult to setup and contains many small pieces that are easily lost. Digital versions of the game are also commonly used, however much of the thrill of the game is lost in this medium as you are no longer playing face to face with a person. To overcome the limitations of both the physical version and the existing digital versions, a blockchain hosted battleships game with token spending is considered. By integrating the spending of tokens into a chance-based game, the thrill is restored as money is now on the line. Additionally, due to the decentralized design of the blockchain, many players can easily play together and share the battleships experience.

Related Work and Context

Blockchain based remakes of games are widely produced with the large incentive being earning money through play. There are even several existing implementations of battleships that use smart contracts. From review of these implementations, many adhere closely to the base game with an emphasis on graphics or they simply use the tokens to bet on games. My implementation deeply integrates the token system into the game and makes changes to the base game by allowing multiple players to all guess against the same board.

Architecture and Contract Design

Diagram



API Schema:

Execute:

Request	Response
CreateGame { size, ships, name }	{}
TakeShot { game_id, x, y }	{}
CollectWinnings { game_id }	{}

Query:

Request	Response
AllGames {}	{ ids: [1, 2, 3, ...] }
Game { game_id }	{game_id, size, total_reward, shots_taken, name, ships, owner, completed, reward_collected }

State Explanation:

The primary state variable in the contract is a keymap of ids to games, respectively called GAMES. The other state variables are items called NEXT_ID and LAST_ACTIVE_ID.

NEXT_ID is initialized on instantiation and is incremented every time a game is created.

LAST_ACTIVE_ID keeps track of the oldest id for which all prior games are inactive and have had their rewards collected.

GAMES contains a map to all the games that have been played and are currently being played on the chain. The game keeps track of its creator, how old it is, and the general state of the battleships game, largely made up of ships and shots. This state is never directly returned in queries because the ship position information could be exploited with falsified queries, so once a game is created, the ships positions are only ever used on the backend.

Frontend Workflow and UX

Connection

The user is first shown the login page. This contains all the configuration parameters for the user to input their own wallet details, and the url and hash for where the contract is stored. When the user clicks the login button, a test query is made to determine whether the site is accessible from the parameters provided. The user cannot login until a successful test query is completed to prevent the user from encountering strange functionality from invalid parameters.

Error Handling

Toast notifications are tastefully used in the application to keep the user up to date with the result of the queries and operations that have been performed. When an operation is happening, interaction with the frontend is limited as much as possible to prevent conflicting actions from the user.

Implentation Choices and Trade Offs

Because a keymap was used to keep track of games, the system of using an incrementing id variable to know what games to query was used. This system has several trade-offs and some benefits. There is no way for the backend to get a summary or subset of data about a game. Any time any information is needed, the whole game is retrieved. Having a different, more fractured, storage method may allow for more efficient operations however, it would likely come with a high difficulty of maintaining a consistent state.

The LAST_ACTIVE_ID variable was introduced to prevent old and stale games from being returned in the query results. Because the actual information contained for a game is quite small, within the scope of the application there was no need to delete old game data, simply moving the id range so that they are never queried is sufficient for this case.

Future Work and Roadmap

Having a system of varying shot costs or just balancing the costs more in general would make the game more engaging with more risk/reward scenarios.

When a game has been closed and all the shots are visible, there could be a system implemented for the user to save the board as an NFT in a personal collection.