Final Report

Mason Evans

Nicolas Towery

Luke Schnetlage

# Introduction

Sellswords and Spellcrafts is a real time player versus player turn-based card game that is played in a client’s web browser.

# Technology

## Original Tech-Stack

* Hosting service for domain: iFastNet
  + iFastNet was originally chosen as the hosting service for the domain because a group member already paid for the service.
* Hosting service for database: iFastNet
  + iFastNet was chosen as the hosting service for the database as it was already offered under the plan the group member was paying for.
* Database language: MySQL
  + MySQL is the only database language that iFastNet supports.
* Frontend framework: React
  + React was chosen in order to streamline the frontend development process with rich online documentation and the use of JSX components and hooks to write code more efficiently compared to vanilla HTML, CSS, and JavaScript.
* Backend framework: Node JS / PHP
  + PHP was chosen for the back-end developer’s familiarity, however the complexity of integrating another language into an already challenging system proved unnecessarily complex.

## Final Tech-Stack

* Hosting service for domain: Replit
  + The swap to Replit was made primarily due to the fact that iFastNet’s version of Node was from 2016 and did not allow us to upgrade to a newer version as any newer version of Node does not support the provided version of Linux. iFastNet’s support team was also unhelpful and offered no aid to upgrade our version of Linux stating that they did not even know what version it was running.On top of this, iFastNet has poor documentation which mostly consists of an outdated FAQ. iFastNet also has notoriously slow servers. iFastNet offers little customization for your server and is mostly built for users wishing to deploy WordPress websites. iFastNet also has a clunky UI especially when compared to Replit.
  + Replit was chosen to replace iFastNet as it is routinely updated, unlike iFastNet. Replit has rich user created and official documentation. Replit has a simple built-in authentication service. It also serves as a real time virtual IDE where all group members could connect to and make adjustments to the same document at the same time. The IDE also saves every individual keystroke allowing us to revert and refer to previous code on the fly. We pay for a premium Replit hosting plan which allows the server to always be on and has more system resources compared to the free version.
* Hosting service for database: Supabase
  + Supabase was chosen as the database service because it offered the most compatibility with Replit. It was infeasible to try and make a MySQL connection with the React project from Replit. Node makes it possible to have a direct connection to any Supabase database, provided you have the API key and the database-specific URL.
* Database language: Postgresql
* Frontend framework: React
  + React was still chosen as a frontend framework because Replit has good support and documentation for using it.
* Backend framework: Node JS

# Design

## Frontend

### Libraries

* React-router-dom
  + Used to create a functional navbar of specified JSX components.
* Socket.io-client
  + Used to create a web socket connection to the server.
* React
  + Necessary library for working in React framework. This library was specifically used for the useState and useEffect hooks.

### File Structure

The file structure for the frontend development environment is as follows:

* Index.html
* Src
  + App.jsx
  + Index.jsx
  + Pages folder
    - Game\_state folder
      * Board folder
        + Board.jsx
        + Circle.jsx
        + River.jsx
        + Slot.jsx
      * Active.jsx
      * Post.jsx
      * Pre.jsx
    - Game.jsx
    - Home.jsx
    - Login.jsx
    - Rules.jsx
  + Socket folder
    - Socket.js
  + Static\_content folder
    - Nav\_logo folder
    - Footer.jsx
    - Navbar.jsx
  + Styles folder
    - Cards folder
      * [cardid].png
    - Fonts.css
    - Main.css
    - Navbar.css
    - Rules.css
    - App.css

The file structure for the static deployment build is as follows:

* Assets
  + [cardid].png
* Index.html
* favicon.ico

### App.jsx / index.jsx

These two files are the core of any react project. Index.jsx renders the App.jsx component and is tied to a div element within the index.html file with the ID of “root”. App.jsx is the root component of all other components within the project and is what everything will be rendered through. App.jsx imports the “react-router-dom” library for the ability to make its children components become pages. It also imports the socket constant from socket.js These pages are /game for Game.jsx, / for /Home.jsx, /rules for Rules.jsx, and /login for Login.jsx. Above any rendered child will be the Navbar.jsx component, and below will be the Footer.jsx component. App.jsx also imports fonts.css, main.css and App.css for stylization. App.jsx also consists of important event listeners for the server. The “connected-socket-users” listener will initialize an array called availableUsers of all current connected users. The “user\_connection” listener will add a newly connected user to the preexisting availableUsers array. The “game\_request” listener will execute when another client has selected the current client as an opponent. It will then emit to the server a “join\_game” call. The “game\_joined” listener will grab the gameID and send it to its relevant child component. It will also set the game state in Game.jsx to “active” for the player that did not initiate the game.

### Footer.jsx

Footer.jsx has a default export function called Home. Inside this function is a constant that is tied to a JavaScript function to get the current year. Once Footer.jsx has been referenced, it will display to the screen the group member’s names as well as the current year.

### Navbar.jsx

Navbar.jsx imports the react libraries, allowing the use of the useState and useEffect hooks. It also imports the stylings from navbar.css and the logo from the nav\_logo folder. Within the Navbar function, there are four constants created at the beginning: location, which uses the useLocation hook to get the current webpage a user is on, the pathname, which is used to highlight the current page, active, which is used to track whether the navigation menu is open or not, and the icon, which is used to track the state of the hamburger menu. The navToggle const is used to toggle the state of both active and icon. It is called when the user clicks on the hamburger menu icon. The handleClick constant is used to close the navigation menu when the user clicks outside of the menu. Following that, there is the return statement, which returns all navigation menu components. Specifically, there is the nav element which acts as the container for everything in the navigation menu, with the onScroll property to handle a click. The Link component allows for the navigation between pages. Lastly, the hamburger menu div element to toggle the menu is presented.

### Game.jsx

Game.jsx imports the following components: Pre.jsx, Active.jsx, Post.jsx. It also imports the React library. It consists of one function called “Game”. This function serves as an export function that is called on when this component is referenced. This function also has a variable called gameState that will be set to either “pre”, “post”, or “active” and will display the corresponding imported JSX component accordingly. Game.jsx will only render to the screen these three children JSX components and nothing else.

### Home.jsx

Home.jsx imports the React library and consists of only its export function that will be called on when the component is referenced. It simply returns the project title and serves as a landing page for the website.

### Rules.jsx

Rules.jsx imports the React library as well as Rules.css and consists of only its export function that will be called on when the component is referenced. It simply displays the rules for the game.

### Socket.js

Socket.js imports the “socket.io-client” library and sets a constant called “socket” to the socketID by creating a connection. This constant is set at the default export for when this file is imported into another file.

### Pre.jsx

Pre.jsx imports the React library and the socket from socket.js. It has an export function called “Pre” which will be invoked when the component is referenced. This function will return to the screen a list of all users online with a corresponding “Challenge” button. Once a user clicks the challenge button, a function nested inside of Pre will run called “startGame.” This function takes the current user’s ID, the user ID of the user which the current user challenged, and the socket ID of the user which the current user challenged. It will then use the socket to emit a call titled “create\_game” and pass the information that was passed to the function. It will then listen for the socket call “game\_created” and then set the game state within the parent function to “active” Game.jsx to display the Active.jsx component

### Active.jsx

Active.jsx imports the React library, the socket constant from socket.js, and the Board.jsx component. It has a default export function called Active which is called on when the component is referenced. By default, this function will render three buttons to the screen labeled as the following: Wind, Fire, Water. Upon clicking one of these buttons, a function nested within the Active function labeled selectDeck will execute. It will emit a socket call called “deck\_selected” to the backend and will pass the backend the deck the user selected and the gameID which was grabbed in App.jsx. This function also listens for a socket call called “opponent\_ready” from the server. If the user has selected their deck and the “opponent\_ready” call has yet to be heard, it will display to the screen “Opponent is choosing deck.” If the user has selected their deck and the server has emitted “opponent\_ready,” it will return the Board.jsx component to the screen.

### Board.jsx

Board.jsx imports the React library, the socket constant from socket.js, Circle.jsx, and River.jsx.It has a default export function called Gameboard which is called on when the component is referenced. By default, the function will return to the screen “Loading. . .” A socket emitter called “get\_game” will be sent to the server upon this component first being rendered. A socket listener called “game\_info” will receive all relevant game information and store it in a JavaScript object called gameInfo. Once the game information has been received, if gameInfo determines the current user to be “player1”, it will show player1’s cards in their hand at the bottom of the screen, information such as health, energy, and end turn button tied to player1 at the bottom of the screen, Circle.jsx tied to player1 at the bottom of the screen, River.jsx in the middle of the screen, Circle.jsx tied to player2 at the top of the screen, and the count of the cards in player2’s hand at the top of the screen. If the gameInfo object determines the current user to be player2 it will display what it displayed for player1 vice versa. As stated previously, all relevant information is within the gameInfo object. The default export function for Board.jsx also contains two nested functions called endTurn and selectCard. endTurn has a corresponding button on the screen that the user can press. Upon clicking this button, if it is the user’s turn (determined within the gameInfo object), it will emit to the server “end\_turn” which will in turn send a newly updated “game\_info” call. When A user either selects a card from their hand or within their circle, selectCard will be invoked. If it is the current user's turn, it will store the card information in a JavaScript object called cardSelected. It will then pass the cardSelected object to either Circle.jsx or River.jsx, depending on which slot the user chose to place the card.

### Circle.jsx

Circle.jsx only imports Slot.jsx. It has a default export function called Circle. If the circle that it is rendering is the current user’s opponent’s circle, it will display the back row at the top and the front row at the bottom. If it is the current user’s circle, it will display the front row at the top and the back row at the bottom. Each row consists of multiple Slot.jsx components. Each slot has an ID that matches up with that of gameInfo in Board.jsx. When a card is placed on a slot, Circle.jsx will take the cardSelected passed from the Board.jsx component and send the object to the Slot.jsx that it was placed in calling it cardAttempt. If the gameInfo reads that a card was already placed in a slot, this will also be sent to the slot in a JavaScript object called cardPlaced.

### Slot.jsx

Slot.jsx only imports the socket constant from the socket.js file. If the cardPlaced variable for the slot is null, it will return a button element which displays what type of slot it is (minion, invocation, or attack zone). Once clicked, it will execute the function slotSelected. When slotSelected executes, it will first check to make sure the current user did not select a slot from their opponent’s circle. Then it will check to make sure that the user selected a card before clicking the slot. It will then check to see if the card can be placed in the specific slot (example: it would reject if a user attempted to place an invocation in a minion slot). Once it passes all of those checks, it will do one of two things. If the card is being placed in a minion slot, it will emit to the server “place\_minion” and send it the relevant information. If the card is being moved from a minion slot to an attack zone, it will emit to the server “attack\_zone” along with the relevant information. Both of these emits will eventually result in a new “game\_info” call within Board.jsx. When the gameInfo has been updated it will extract what cards have been placed in what slots and send it to the relevant Slot.jsx.There is currently no implementation to place an invocation card. If the cardPlaced variable for the slot contains a value, it will display a button with a class named “card-[cardPlaced].” This will be read by App.css and will display the corresponding card within the slot.

### Cards folder

The cards folder contains the image files of all cards that have been created for this game. They are all in PNG format.

### Styles

Fonts.css applies all fonts to relevant selectors. main.css is where styles for the footer and headers are created. navbar.css is where the styling for each aspect of the navigation menu is created. rules.css is where the styles for the Rules page are applied. Lastly, in App.css, the game-specific styles are created or imported to allow for design to specifications.

### Index.html

This is an HTML file pre generated by React. All information will be placed in the div element with the id “root”.

## Backend

### Libraries

* Repl-auth
* Express
* Node JS
* Socket.io

### File Structure

* Server
  + Server.cjs
  + Pregame\_actions
    - game\_creation.cjs
    - pregame.cjs
  + game\_actions
    - board\_functions.cjs
    - hand\_functions.cjs
    - info\_functions.cjs
    - Turn\_functions.cjs
  + dist
    - assets
      * 1.png
      * 2.png
      * …
      * 50.png

### server.cjs

The backend can primarily be viewed as an extension of ‘server.cjs’. This is, as the name implies, the file in which an express server using CORS is deployed alongside the relevant server-side socket.io object. The server then listens for a client to connect to the server using the “io.on('connection’)” call. Once a client connects to the server, the client is added to the ‘player’ table in the database using their Replit username and userID. Then the connected user is added to the ‘general’ room where all users not currently in a game are placed and an updated list of all connected users is sent to all users in ‘general’. After this, the server listens for the following socket emits correlating to the relevant client side actions alongside relevant information:

* list\_available\_users

On a ‘list\_available\_users’ emit, the server will run the ‘getusers()’ function defined in pregame.cjs and emit the result back to all connected clients to ensure all parties have a correct and updated list of connected users.

* create\_game

A ‘create\_game’ emit must be accompanied by the playerids of the challenging and challenged player as well as the socket ID of the challenged player. Both playerids are used with the ‘createGame()’ function defined in ‘game\_creation.cjs’ to create a new row in the ‘game’ table of the database. Upon success, a ‘game\_request’ emit will be set to the challenged player’s socket to trigger a ‘join\_game’ emit from the challenged client. Finally the challenging player’s socket will be removed from the ‘general’ room and placed in a room named after the newly created game’s ID.

* join\_game

Once a client has processed a ‘game\_request’ and emitted back ‘join\_game’ , the challenged player will be removed from the ‘general’ room and added to the same game room defined in ‘create\_game’.

* deck\_selected

Upon receiving a ‘deck\_selected’ emit alongside the id of the deck selected and the game ID, the server uses the ‘populate\_cards\_in\_deck()’ method defined in game\_creation.cjs to create rows in the ‘cards\_in\_deck’ table in the database equal to a random ordering of the cards in the designated decklist. Then the server emits a ‘opponent\_ready’ message to the room both players are in.

* get\_game

Upon receiving a ‘get\_game’ emit alongside the game ID, the server will use the ‘getfullgame()’ function defined in ‘info\_functions.cjs’ to collect all relevant information about a game, including board state, player hards, player decks, and player health, and emit it back to the client with the ‘game\_info’ emit.

* place\_minion

Upon receiving a ‘placeminion’ emit, the server will use the ‘placeminion()’ function defined in ‘board\_functions.cjs’ to remove a minion card from the player’s hand and add that minion to the indicated. Then the same ‘game\_info’ that is used in response to a ‘get\_game’ emit to update the client’s boardstate.

* attack\_zone

Upon receiving a ‘attack\_zone’ emit, the server will use the ‘attackzone()’ function defined in ‘board\_functions.cjs’, then perform the same ‘game\_info’ emit as ‘place\_minon’ or ‘get\_game’ using ‘getfullgame()’.

* end\_turn

Upon receiving an ‘end\_turn’, the server will use the ’start\_turn()’ function defined in ‘turn\_functions.cjs’ which will change the active player, draw a new turn player a card, and reset their active terrain to its previous max value + 1. It will then perform the same ‘game\_info’ seen previously.

### pregame.cjs

‘pregame.cjs’ is a relatively simple file, consisting of the definitions for two functions, ‘createPlayer()’ and ‘getusers()’. ‘createPlayer’ takes two arguments, the users Replit username and userid; this information is then stored in the ‘player’ table of the database under the ‘username’ and ‘userid’ columns respectively. All modifications to the database utilize asynchronous javascript and the Supabase javascript API unless explicitly mentioned otherwise. ‘getusers()’ takes two arguments, the ‘io’ object representing the server’s socket port and the ‘socket’ object representing the client’s socket port; it uses this information to create an array named ‘users’, populate this array with with the Replit ID, Replit username, and socket ID of each connected user, then return said array.

### game\_creation.cjs

‘game\_creation.cjs’ primarily consists of functions required to create games, populate game objects, and allow users to select their deck using the following functions:

* createGame (startPlayer, joinPlayer)

‘createGame()’ randomly selects one of the players passed to it and designates them as the active player for that game, meaning they will start the game. Then a new row in the ‘game’ table is inserted using the values passed in. Once a game is successfully created, createBoard() is used twice, once for each player and createcontestedzone() is used once to create all the zones required for an initial gamestate.

* createcontestedzone (gameid)

‘createcontestedzone()’ inserts a new row in the ‘contested\_zone’ table using the game ID passed to it.

* getdecklist (decklistid)

‘getdecklist’ takes a decklist id and uses that to query the ‘decklist’ table and returns that decklist.

* shuffle (sourceArray)

‘shuffle()’ takes an array (of cards) and randomizes the order by looping through the array and swapping the contents of sourceArray[i] and a randomly selected value in the array for each value in the array.

* translateDecklisttoCards\_in\_deck (decklistid, playerid, gameid)

‘translateDecklisttoCards\_in\_deck()’ takes the decklist ID passed to it and uses it with the ‘getdecklist()’ function to get the deck in the decklist format. A nested for-loop is then used to convert the decklist into ‘cards\_in\_deck’ format by adding a card to an array for each value in ‘quantity’ for each card in the decklist. That array is then shuffled using ‘shuffle()’ then returned.

* populate\_cards\_in\_deck (decklistid, playerid, gameid)

‘populate\_cards\_in\_deck()’ takes a decklist ID and passes it to ‘translateDecklisttoCards\_in\_deck()’ and stores the returned deck in the ‘cards\_in\_deck’ format. That deck is then inserted into the ‘cards\_in\_deck’ table. Once successful, the player draws 5 cards as their “opening hand”.

* createBoard (playerid, gameid)

‘createBoard()’ inserts a new row in the ‘board’ table with the default values: spent\_terrain =0,health = 20,AFKwarnings= 0.

### turn\_functions.cjs

‘turn\_functions.cjs’ governs logic about players ending/starting their turns using the following functions:

* start\_turn(gameid, playerid)

‘start\_turn()’ takes the game ID and the player whose turn is about to begin. The player given will draw their card for turn using ‘draw()’, reset their terrain using ‘resetterrain()’, and the ‘active\_player’ column is updated to reflect the new active player.

* resetterrain(gameid, playerid)

‘resetterrain()’ queries the ‘board’ table using the game and player ID, then updates that table to have 0 ‘spent\_terrain’ and an amount of ‘active\_terrain’ equal to the number of previously ‘active\_terrain’ plus ‘spent\_terrain’ plus one. This is to give the player who started their turn all their previous terrain plus the one new terrain given to every player at the start of their turn.

### board\_functions.cjs

‘board\_functions.cjs’ is home to the functions required to make changes to the board state, such as placing minions, attacking, or paying costs for effects using the following functions:

* attackplayer(attackingplayerid,targetplayerid,gameid,minionid,zoneid)

‘attackplayer()’ assumes the minion is occupying a contested zone. With this assumption, ‘payenergy()’ is used to either pay the cost of an attack or return false if the player does not have sufficient ‘active\_terrain’ to attack. If the energy can be paid, the player’s health is reduced using ‘damageplayer()’.

* attackzone(attackingplayerid,gameid,attackingminionid,homezone,targetzone)

‘attackzone()’ first checks to see if the attack can be paid for using the ‘payenergy()’ function. If the attack is legal, the function next determines if the zone to be attacked is occupied or empty. If the zone is occupied, the function next uses ‘damageminion()’ to determine if the minion in the zone being attacked would be destroyed. If the minion would not be destroyed, the ‘moveminion’ variable is set to false, otherwise it is defaulted to true. If ‘moveminion’ is set to true, the minion is removed from its current location and added to the zone indicated by ‘targetzone’. If the minion cannot be moved, false is returned.

* damageminion(gameid, zoneid, dmg)

‘damageminion()’ assumes minions can only be damaged in the contested zones. Using this assumption, the function compares the current power of the minion in the zone indicated by ‘zoneid’ to ‘dmg’. If the result would reduce a minion’s power to zero or less, the minion ID is removed from the ‘contested\_zone’ and the zone is set to null. Otherwise, the ‘zone\_pow’ is reduced by ‘dmg’, but the minion is not removed.

* placeminion(playerid, gameid, minionid, zoneid)

‘placeminion()’ initially checks to see if there is already a minion occupying the zone passed in ‘zoneid’. If the zone is occupied, the function returns false, otherwise the player is damaged equal to the minion’s summoning cost using ‘damageplayer()’. Then the indicated zone in ‘board’ is updated with ‘minionid’. Finally, the card is removed from the player’s hand using the ‘discard()’ function.

* placeterrain(playerid, gameid)

‘placeterrain()’ simply increments the ‘active\_terrain’ value in a player’s ‘board’ by one.

* payenergy(playerid, gameid, energyCost)

‘payenergy()’ checks if a player has enough ‘active\_terrain’ to pay for the ‘energyCost’, if they do not, it returns false. If they do have enough energy, the the ‘active\_terrain’ and ‘spent\_terrain’ values in ’‘board’ are reduced and increased respectively by energyCost.

* damageplayer (playerid,gameid,dmg)

‘damageplayer()’ updates the ‘health’ value in a player’s ‘board’ to be its current value minus damage.

### hand\_functions.cjs

‘Hand\_functions.cjs’ contains the logic for drawing and discarding cards from and to a players hand using the following functions:

* draw (playerID, gameID, drawCount)

‘draw()’ doesn’t actually contain much logic, it simply checks for drawCount, if drawCount is undefined, it assigns drawCount as one, the default draw. Draw then removes cards from the deck in the ‘cards\_in\_deck’ using ‘remove\_card\_from\_deck()’ and places them in the ‘hand’ table using the ‘’put\_card\_in\_hand()’ function.

* remove\_card\_from\_deck (playerID, gameID,drawCount)

‘remove\_card\_from\_deck()’ does as the name implies and removes a number of cards from a deck equal to the number passed in ‘drawCount’ and returns these values.

* put\_card\_in\_hand (playerID, gameID, card, drawCount)

‘put\_card\_in\_hand ()’ takes an array ‘card’ returned by ‘remove\_card\_from\_deck ’ and formats and inserts it into the ‘hand’ table in the database.

* discard (playerID, gameID,cardID)

‘discard’ deletes the first card in the ‘hand’ table that matches all three values passed in.

### info\_functions.cjs

‘info\_functions.cjs’ makes no direct modifications to the game state, but contains queries for collecting relevant game information to help other functions or relay information to the front-end using the following functions:

* getfullgame (gameid)
* getcardsindeck (playerid, gameid)
* getplayerhand (playerid, gameid)
* getplayerboard (playerid, gameid)
* getcontestedzone (gameid)
* getgame(gameid)
* getactiveterrain(playerid, gameid)
* getminion(minionid)

All functions except ‘getfullgame()’ do very similar things, query the database using the ID passed to get a row/rows from the table indicated by the name i.e. ‘getgame()’ queries the ‘game’ table, ‘getcontestedzone()’ queries ‘contested\_zone’, ect… ‘getfullgame()’ is notable for not directly querying the database, but instead runs ‘getcardsindeck, getplayerhand, getplayerboard, getcontestedzone, getgame, getactiveterrain’ and returns an array containing the results of all queries.

## Database

The Supabase database is used to store all game and user-related information. It has fifteen tables, each of which have varying numbers of fields. The following is a list of the tables and their respective data being stored in each table:

### Tables

* player
  + userid
  + username
* result
  + resultid
  + win\_res
* game
  + gameid
  + round\_start
  + start\_player
  + join\_player
  + result
  + active\_player
* card
  + cardid
  + card\_name
  + card\_type
  + energy\_type
* energy
  + energyid
  + energy\_type
* minion
  + minionid
  + minion\_name
  + energy\_type
  + summon\_cost
  + start\_pow
  + atk\_cost
* spell
  + spellid
  + invocation\_name
  + energy\_type
  + fast\_cost
  + slow\_time
* terrain
  + terrainid
  + energy\_type
* zone\_traits
  + zoneid
  + zone\_type
* contested\_zone
  + contested\_zoneid
  + zone1\_minid
  + zone2\_minid
  + zone3\_minid
  + zone1\_effect
  + zone2\_effect
  + zone3\_effect
  + revealed1
  + revealed2
  + revealed3
  + zone1\_pow
  + zone2\_pow
  + zone3\_pow
  + gameid
  + zone1\_owner
  + zone2\_owner
  + zone3\_owner
* hand
  + handid
  + playerid
  + gameid
  + cardid
  + card\_type
* deck\_list
  + deckid
  + cardid
  + quantity
  + card\_type
* cards\_in\_deck
  + card\_space
  + cardid
  + playerid
  + gameid
  + card\_type
* discard
  + discardid
  + playerid
  + gameid
  + card\_type
* board
  + boardid
  + playerid
  + gameid
  + minion1
  + minion2
  + minion3
  + invocation1
  + invocation2
  + invocation3
  + active\_terrain
  + spent\_terrain
  + health
  + AFKwarnings

### 

Figure 1. Database ERD

### Player Table

The player table is where user information is stored. The userid, which is the primary key, is a randomly generated integer by socket.io, and following that is the username, which is a varchar and is taken in from a user’s Replit account, which allows them to access critical features of the website.

### Result Table

The result table is where the two basic win types are stored. The resultid, which is the primary key, is an integer and is used to pass the information about what type of result will be assigned to each player at the end of a game. The win\_res is a boolean value with the purpose of being set to true and assigned to whichever player has won, and false for whomever has lost.

### Game Table

The game table is where all game-relevant information is passed to allow for actual creation of a game. The game id is an integer and is the primary key. The round\_start is a timestamp value, storing the time that each round begins. The start\_playerid is an integer that references the userid of whichever player challenges another player. The join\_playerid is an integer that references the userid of whichever player was challenged. The result is an integer that references the resultid from the result table. The active\_playerid references the userid of whichever player’s turn it is.

### Energy Table

The energy table is where the four main energy types are stored for reference by other tables. The energyid is an integer and is the primary key. The energy\_type is a text type and is used to store the four types of energy: void, water, wind, and fire.

### Card Table

The card table is used to store all cards and their general information. The cardid is an integer and the primary key. The card\_name is a text type and stores all names of all cards. The card\_type is used to store the three types of cards: minion, invocation, and terrain. The energy\_type references the type of energy associated with each card.

### Minion Table

The minion table is used to store all information about the minion card type. The minionid is an integer and is the primary key. The minion\_name references the card\_name where the card\_type is minion. The energy\_type references the energy\_type from the energy table. The summon\_cost is an integer and stores the cost to summon a particular minion into a game. The start\_pow is an integer and is the power level that a particular minion card starts with in a game. The atk\_cost is an integer and stores the amount that it would cost to attack with a specific minion card.

### Spell Table

The spell table is used to store all information about the invocation card type. The spellid is an integer and is the primary key. The invocation\_name references the card\_name where the card\_type is invocation. The energy\_type references the energy\_type from the energy table. The fast\_cost is an integer and stores the amount of life it would cost to speed up the time to use a particular invocation. The slow\_time is an integer and stores the standard amount of time it takes to use a particular invocation.

### Terrain Table

The terrain table is used to store the types of terrain. The terrainid is an integer and is the primary key. The energy\_type references the energy\_type from the energy table.

### Zone Traits Table

The zone\_traits table is used to store the type (or trait) of a zone. The zoneid is an integer and is the primary key. The zone\_type is a varchar and stores the effect that a zone can take.

### Contested Zone Table

The contested\_zone table is used to store all information regarding player involvement with a zone. The contested\_zoneid is an integer and is the primary key. The zone1\_minid, zone2\_minid, and zone3\_minid are all integers and store the minionid of the card occupying its space, if there is one present. The zone1\_effect, zone2\_effect, and zone3\_effect all reference the zone\_type from zone\_traits. The revealed1, revealed2, and revealed3 are all boolean values, storing false if the zone is yet to be revealed, and storing true if it is revealed. The zone1\_pow, zone2\_pow, and zone3\_pow are all integers and store the power of each zone. The gameid references the gameid. The zone1\_owner, zone2\_owner, and zone3\_owner all reference the userid of whichever player is occupying that zone.

### Hand Table

The hand table is used to store the creation of a hand upon creation of a game. The handid is an integer and is the primary key. The playerid references the userid from the player table. The gameid references the gameid from the game table. The cardid and card\_type reference the cardid and card\_type from the card table, respectively.

### Deck List Table

The deck\_list table is used to store the list of decks in the system. The deckid is an integer and is a composite key with cardid, which references the cardid from the card table. The quantity is an integer and stores the number of that particular card that is present in that deck. The card\_type references the card\_type from the card table.

### Cards in Deck Table

The cards\_in\_deck table is used to store the index of each card in a deck. The card\_space is an integer and is meant to act as the index, but is also the primary key. The playerid references the userid from the player table. The gameid references the gameid from the game table. The card\_type references the type of card from the card table.

### Discard Table

The discard table is used to store the cards that have been moved to the discard pile after being played. The discardid is an integer and is the primary key. The playerid references the userid from the player table. The gameid references the gameid from the game table. The card\_type references the card\_type from the card table.

### Board Table

The board table is used to store most information that has been created in other tables to transfer stored information to a format so as to display it to users. The boardid is an integer and is the primary key. The playerid references the userid from the player table. The gameid references the gameid from the game table. The minion1, minion2, and minion3 are all integers and store the minionid of whatever minion is occupying each minion slot on the board. The invocation1, invocation2, and invocation3 are all integers and store the spellid of whatever invocation is currently occupying each invocation slot on the board. The active\_terrain is an integer and stores the amount of active terrain currently available to the player. The spent\_terrain is an integer and stores the amount of terrain that has already been used to further gameplay. The health is an integer and stores the amount of health each player has at the moment. The AFKwarnings is an integer and stores a value ranging from 0 to 3. Using the round\_start to frame a time, it resets once it reaches a certain time. Once it does this, it resets and the AFKwarnings increments until it reaches 3.

# How to Deploy / Build the Application

If this is the developer’s first time deploying the application, they would need to install several dependencies through the command terminal first. The commands to run in the terminal or shell would look something like this:

* npm install:
  + react
  + react-dom
  + socket.io-client
  + react-router-dom
  + @supabase/supabase-js
  + express
  + cors
  + body-parser
  + http
  + socket.io
* node:
  + server/server.cjs
* npm:
  + run build

The npm install commands are essential for properly introducing each dependency into the project, and the node command is what allows the for running of the server. Using the npm run build common is what allows the project to be compiled and built, so it can be deployed to the server. Following these commands, if working within Replit specifically, a ‘dist’ folder will be created in the root. Moving this folder into the server folder, and either overwriting or creating a new position for this folder, will allow any changes to the codebase to be seen in the web-application.

# Known Bugs

### Ghost Login

If users have previously logged in to the website and close their browser and reopen, they will have access to the website, but do not have any Replit authentication headers that the server can request, allowing them to still log in to the server, but not display a username.

### Users challenging themselves

Users are currently able to view themselves in the list of players on the “Play” screen. And can attempt to challenge themselves.

### Attacking an occupied minion slot

Attacking an occupied minion slot will crash the server.

### Refresh Bugs

If users are in a game and refresh the page, they will no longer be in the game and the other user will not be notified.

# Future Work

There are numerous targets for future work, but immediate priorities are currently based on implementing the attack player function and generally improving minion combat, including being able to attack other minions horizontally in contested zones. Beyond that, developing the ability to place invocations, active invocations, and developing card art for all invocations would be the next goal. Beyond that, implementing the effects of specific minions and invocations to do as the card is printed would be the next goal. Finally, in a short term scope, fixing inconsistent function and variable naming syntax would be a large quality of life increase.

In a longer perspective, adding complexity to minion combat such as limiting attacks based on proximity to a zone or adding hidden effects to zones once occupied by a minion would allow for more interplay and game complexity. In a much larger scope eventually developing a mobile app would increase the availability and ease of onboarding considerably.