This JavaScript code represents the front-end interface for an interactive, real-time multiplayer poker game. It leverages the capabilities of HTML, CSS, JavaScript, and the real-time bidirectional event-based communication capabilities provided by Socket.IO.

Firstly, the code uses JavaScript functions and methods to interact with the Document Object Model (DOM) and HTML local storage. The game's state, the user's unique ID, and username are stored locally when the user logs in. Two main functions, **isUserLoggedIn** and **redirectToLobbyIfAuthenticated**, check for user authentication and navigate to the game lobby accordingly.

The game lobby is fetched by the **fetchLobby** function, which interacts with the server via asynchronous fetch requests and updates the HTML body with lobby content. A WebSockets connection is then established, and the user joins the game lobby. Several listeners for socket events are set up, such as receiving messages and game updates.

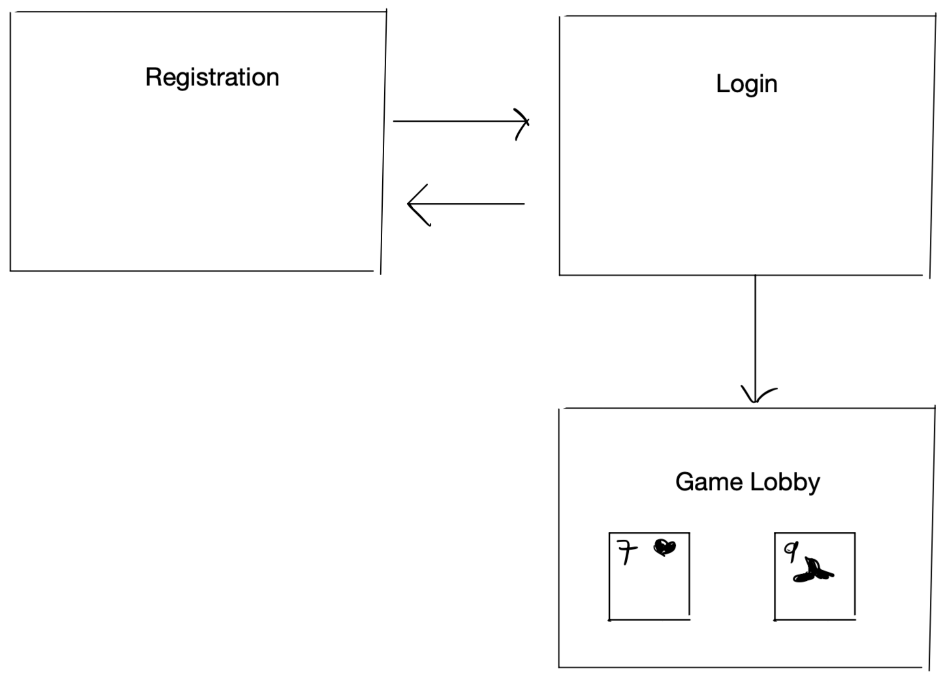
The **renderGame** function dynamically generates and updates the game interface, including the scoreboard, user's cards, and game buttons like 'Bet' and 'Fold'. The function uses JavaScript's DOM manipulation methods to dynamically create, modify, and append HTML elements. A notable feature here is the inclusion of event listeners for 'Bet' and 'Fold' buttons that trigger respective socket events to the server.

Lastly, numerous socket event listeners are present to handle real-time game updates and user actions. These include the start of a game, game updates, a user folding, bet results, resumption of a game, and the end of a game. Each of these listeners updates the DOM to reflect the new state of the game.

In conclusion, this code effectively integrates Socket.IO, a fast and reliable real-time engine, with HTML, CSS, and JavaScript to create a dynamic, interactive, multiplayer poker game. It stands as a testament to the capabilities of modern JavaScript and real-time technologies in creating engaging, real-time applications.

API Documentation

Wire Frame



This documentation details the routes and corresponding controllers for the User feature of our application.

## Endpoints

### POST /register

This endpoint is used to create a new user. The input parameters are `username`, `password`, and `email`. Upon successful creation, a token is generated, and the user's details are sent back with a status code of `201`.

### POST /login

This endpoint is used to log in an existing user. The required parameters are `username` and `password`. If login is successful, a token is generated, and the user's details are returned with a status code of `200`.

### POST /logout

This endpoint is used to log out an existing user. The route is protected by an authentication middleware that requires a valid token. Upon successful logout, a response with the message `"User logged out successfully"` is returned with a status code of `200`.

GET /is-authenticated

This endpoint is used to check if a user is authenticated. The route is protected by an authentication middleware that requires a valid token. If the user is authenticated, a JSON response `{ authenticated: true }` is returned with a status code of `200`.

### GET /lobby

This endpoint is used to fetch the lobby view. It is protected by an authentication middleware that requires a valid token. The lobby view is rendered with the authenticated user's details.

### GET /register

This endpoint is used to fetch the registration view. It returns the view with a status code of `200`.

### GET /login

This endpoint is used to fetch the login view. It returns the view with a status code of `200`.

## Controllers

### createUser

This controller is used to create a new user. The function takes the `username`, `password`, and `email` from the request body. It then calls the `createUser` function from the User Model. A token is generated for the new user, and a cookie is set with this token. If the user creation is successful, a JSON response with the new user's details and a success message are returned with a status code of `201`.

### getUserById
This controller is used to fetch a user's details by their `id`. If the user is found, their details are returned with a status code of `200`. If the user is not found, an error message is returned with a status code of `500`.

### login

This controller is used to log in an existing user. The function takes `username` and `password` from the request body, and then calls the `login` function from the User Model. A token is generated for the logged-in user and a cookie is set with this token. If the login is successful, a JSON response with the user's details is returned with a status code of `200`.

### logout

This controller is used to log out an existing user. The function calls the `logout` function from the User Model. If the logout is successful, a JSON response with a success message is returned with a status code of `200`.

### getCurrentUser

This controller is used to fetch the details of the currently logged-in user. The function calls the `getCurrentUser` function from the User Model. If the user is found, their details are returned with a status code of `200`. If the user is not found, an error message is returned with a status code of `500`.

For all controllers, in case of an error, an error message with a status code of `500` is returned, unless otherwise specified by the error's `status` attribute. The controllers are implemented as part of the `userController` object, which is exported for use in routes or other modules as required

Game Logic Documentation

1. **createDeck**: This function creates a deck of cards by iterating over the suits and ranks and combining them.
2. **shuffle**: This function shuffles the given deck of cards using the Fisher-Yates algorithm.

3. **gameState**: This variable represents the state of the game, including information about the active status, pot, current bet, dealer, current player, and players' data.
4. **getGameState**: This function returns the current state of the game.
5. **isUserInGame**: This function checks if a given user is in the game by checking their presence in the **gameState.players** object.
6. **removeUserFromGame**: This function removes a user from the game by updating the game state accordingly. It checks if the player exists in the game, handles the cases where the player is the dealer or current player, and adjusts the active player count. It returns the result of the game based on the number of active players.
7. **playerJoinGame**: This function allows a player to join the game. It checks if the player already exists and updates their state if they are reconnecting. Otherwise, it adds the player to the game state.
8. **playerFold**: This function handles a player folding their hand. It sets the player's active status to false, updates the player's state in the database, determines the next player, and checks if the round needs to be ended. It returns a boolean indicating whether a new round should start.
9. **startGame**: This function starts the game by creating and shuffling a deck of cards. It assigns two cards to each player, sets the active status of players, and determines the dealer and current player if not already set.
10. **playerBet**: This function handles a player's bet. It checks if the player has enough money and if it's their turn to bet. It updates the player's bet amount, current bet, money, and

active status. It returns an object with information about the bet result, such as if the player is all-in or if the round should end.

11. **getNextPlayer**: This function returns the ID of the next player based on the current player. It iterates through the players and skips inactive players. It returns null if no active players are found.
12. **showCards**: This function returns the cards of a given player or a message if the player is not in the game.
13. **calculateHandValue**: This function calculates the value of a hand based on card ranks. It assigns values to pairs, three of a kind, and high cards. It returns the maximum value among them.
14. **determineWinner**: This function determines the winner(s) of the round based on the calculated hand values. It returns the ID(s) of the winner(s) or "tie" if there's a tie.
15. **endRound**: This function ends the current round by revealing each player's hand, calculating the winner, updating players' states and game state, and starting a new round if there are more active players. It returns an object with information about the round result, including cards, winner, and money.
16. **startNewRound**: This function starts a new round by resetting the pot, current bet, and players' bet amounts, shuffling the deck, updating player states, and determining the dealer

17. **endGame**: This function is called when the game is over. It determines the winner(s) based on the maximum number of rounds won and updates players' states accordingly. It sets all players to inactive and not participating. It returns an object with information about the winners and losers of the game.
18. **playerRejoinGame**: This function allows a player to rejoin the game by updating their game state with the provided state.

Game Module Documentation

The module requires the db module for database connectivity and the CustomError module for custom error handling. An empty object gamesModel is created to store the functions related to game database operations.

gamesModel.createGame is a function that inserts a new game into the database. It takes a game_name and game_state_json as parameters. It returns a Promise that resolves to the created game if successful or rejects with an error.

gamesModel.getGame retrieves a game from the database based on the game_id. It returns a Promise that resolves to the retrieved game if found or rejects with an error if not found.

gamesModel.updateGame updates the game state of an existing game in the database. It takes game_state_json and game_id as parameters. It returns a Promise that resolves with the number of rows affected if the update is successful or rejects with an error if no rows are affected.

gamesModel.getRecentGameId retrieves the ID of the most recently created game from the database. It returns a Promise that resolves to the game ID if found or rejects with an error if no games are found.

gamesModel.getActiveGame retrieves the active game from the database. It searches for a game where the isActive field in the game_state_json is set to true. It returns a Promise that resolves to the active game if found or null if no active game is found.

Finally, the gamesModel object is exported to be used by other modules.

In summary, this module provides functions to create, retrieve, update, and get the ID of games from the database. It allows for interaction with the game model data in the database, enabling the persistence and retrieval of game states.

Connection

Upon establishing a connection, the server logs the connection, identifying it by the socket's id.

Join Lobby

The server listens for a "join_lobby" event, where a user can join the lobby by providing their user data (username and user id). This user data is stored locally and is used to track who is online and facilitate interactions. This user's details are then broadcasted to all other users currently online, indicating their entrance into the lobby. Past chat messages are fetched from the database and sent to the user so they can see the chat history.

Bet

The server also listens for a "bet" event, where users can make bets. The user's bet amount is processed and the game state is updated accordingly. The updated game state is then broadcasted to all users. If a user's bet is unsuccessful, only that user is informed of the unsuccessful bet.

Fold

Users can fold during a game by emitting a "fold" event. The game logic is updated to reflect the user's action, and the new game state is broadcasted to all users.

Send Message

When a user sends a chat message (through the "send_message" event), the server first logs the message and then emits it to all other users in the lobby. The message is also stored in the database for record-keeping and for newly joining users to see.

Summary

This code handles the main interactions in a chat room that also has some form of a game (betting, folding, etc). It uses Socket.IO to facilitate real-time bidirectional communication between the server and the clients. Users can join the lobby, place bets, fold, and send messages to the lobby. These interactions are then broadcasted to all connected users.

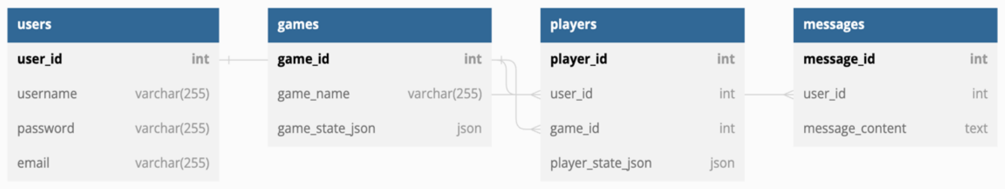
The primary models used in this code include the userModel and gameModel. The userModel is responsible for database operations relating to the users and their messages. The gameModel handles the game state, and the game logic such as betting and folding.

It is worth noting that this code uses a single room ("lobby") for all users, meaning that all messages and actions are broadcasted to all connected clients.

The code also keeps track of all online users and maps their user IDs to their socket IDs. This allows for the server to send messages or game states to specific users when required.

This file can be further optimized by splitting the different sections into their own separate modules based on functionality. For example, all chat-related event handlers could be in a "chat" module, while game-related handlers could be in a "game" module.

Data Base Frame


users
user_id int
username varchar(255)
password varchar(255)
email varchar(255)
games
game_id int
game_name varchar(255)
game_state_json json
players
player_id int
user_id int
game_id int
player_state_json json
messages
message_id int
user_id int
message_content text