RamBluff: Online Poker

Group 9
Sean Berlin, Kirtan Chavda, Jared Colletti, Zachary Leopold

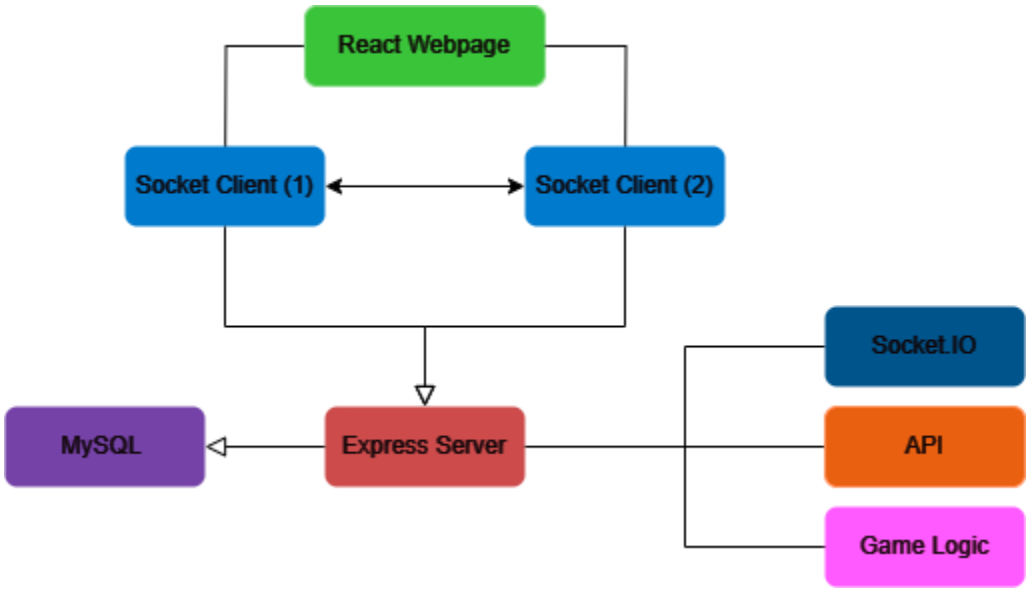
Chapter 1: Vision

Introduction

RamBluff is an online poker web application that delivers a seamless and immersive gaming experience to users. Our platform will utilize cutting-edge technologies to create a dynamic environment where players can engage in poker games with ease and convenience. The games played within our application will contain a user-friendly interface while securely managing player and gamestate data, and comply with the Texas Hold'em ruleset.

RamBluff: Online Poker

Figure 1



Chapter 2: Technical Proposal

2.0 Architecture Overview

Our proposed architecture consists of three main components: the frontend UI developed with React, the backend server powered by Node.js, and the database management system using MySQL.

2.1 Frontend Development (React)

We will adopt a component-based architecture in React to facilitate code reuse, improve maintainability, and enhance development efficiency. With the utilization of CSS, we will ensure that the UI is responsive and optimized for all screen sizes to provide a consistent user experience. Along with that, we will be employing React state / useEffect, socketClient, and axios to orchestrate the synchronization with our backend services. Our users will be presented with a homepage where they will be prompted to create a game instance. To invite others to play, the user will be provided with an invite link to share.

2.2 Backend Development (Node.js)

Our team is designing a RESTful API to handle client-server communication, allowing for seamless interaction between the frontend and backend components. In order to enable real-time communication between players and the server, we will be using Socket.IO; this will allow live game updates and possible chat functionalities. We plan to use middleware libraries, such as Express, to define our routes for the necessary HTTP methods (GET,POST,PUT,DELETE,etc.)

The functionality of the poker game will be powered by the ‘Table’, ‘Dealer’, and ‘Player’ classes. Where ‘Table’ will be written in Node in order to make use of an open-source API, made by “goldfire”, that will handle player hand rankings. Along with that, ‘Table’ will manage the pot, the rotation of the blinds, and current players. Adversely, the ‘Dealer’ and ‘Player’ classes will be written in Python. ‘Dealer’ will handle the shuffling and dealing of cards (Both player and community cards), while ‘Player’ is utilized for player actions and states.

2.3 Database Management (MySQL)

We will design a relational database schema using MySQL to store game states and player profiles(along with any other relevant data). After a player completes a successfully sits down at a table, they will be given a UserID that corresponds to the seat of the particular table at which they are sitting. This will allow us to maintain which users are in which games. Lastly, to ensure efficient data retrieval and storage, we will be using proper normalization and indexing techniques.

2.4 Conclusion

By implementing these technical strategies, we aim to develop a robust and scalable online poker platform that delivers a superior gaming experience to our users while maintaining the highest standards of security, reliability, and entertainment.

Chapter 3: Intermediate Milestones

3.0 Overview

The development of RamBluff has dealt it's share of progress and challenges (specifically, the implementation of Socket.io). When attempting to leverage the library for real-time communication via gamestate updates, it proved difficult to correctly set up the Socket.io rooms. These rooms are what enable us to host multiple running poker games. The logic of creating our player objects and associating them with particular tables had to be tweaked multiple times. However, we have solved this issue. By utilizing a hashmap in our back end, where the Table ID is the key and the list of player objects is the data. When a user performs an action, such as sitting down, our frontend sends an emission with the Table ID to the backend that, in this example, creates a new player object such that it is only associated with that table (socket room). This allows for the multiple games we previously mentioned.

Currently our users are able to sit down and leave unique tables, which is visualized on all clients associated with that table. Our team is working on the implementation of the game logic and the cloud deployment. Where, the game logic is being implemented and tested locally, but the cloud deployment is our main focus. With the structure for our socket rooms in place, our game logic (betting, calling, checking, folding) will be easily tied to each individual user at each unique table. As it stands, our team is fully confident in the completion of RamBluff by the end of the semester.

In regards to the cloud deployment, we will be utilizing Docker and it's image functionality. RamBluff requires the construction of a Docker image for each of the three main components: the React frontend application, the Node.js server, and the MySQL database. Where all require a systematic approach to ensure efficiency, security, and scalability. The following sections detail the procedures for building these Docker images, along with insights on preliminary testing and data management strategies.

3.1 React App (Frontend)

To create our Dockerfile for the React component, we will be hosting our Dockerfile in the frontend's root directory. This file will use the official Node image, specifically node:alpine, for the build stage. This stage will handle installing our dependencies and building the static files for the React application. For the serving stage, we will be switching to nginx:alpine, a lightweight server, to host the built application. We are ensuring that the nginx configuration is optimized for serving a single-page application (SPA) and can handle our routing correctly. Once the Dockerfile is configured the docker build command is used to create the image. This image will be tagged appropriately for future use with Kubernetes and will also be stored on DockerHub. For preliminary testing purposes, the container is ran locally to confirm that the React app is served correctly and connects to our backend API and Socket.io without any issues.

3.2 Node.js Server (Backend)

Our backend Dockerfile will also begin with a base image of node:alpine. It will include copying RamBluff's source code into the container and installing the necessary dependencies. The configuration details, such as database connection strings, are managed using environment variables, which can be defined in our Kubernetes deployment configurations. The docker build command will be used to build our image with the appropriate tags and to ensure the Dockerfile specifies the command to run the Node.js server. For preliminary testing purposes, the container is ran locally to confirm that our API and Socket.io endpoints are accessible and operating as intended.

3.3 MySQL (Database)

The database setup involves using the official MySQL Docker image, eliminating the need for a custom Dockerfile. Configuration tasks, such as setting up our database schemas are achieved by mounting SQL scripts into the Docker container at /docker-entrypoint-initdb.d.Key environment variables like MYSQL_DATABASE, MYSQL_USER, and MYSQL_PASSWORD are utilized to configure the database instance. When building our image we are preparing the Docker Compose file instead of creating a new image, given the direct use of the official MySQL image. During testing we are ensuring the accessibility of the database from the Node.js backend container and are conducting tests to confirm smooth execution of initial data loading and schema creation scripts.

3.4 Data Collection/Creation and Tables Management

For RamBluff, data management involves handling user profiles, game states, and other relevant information within the MySQL database. Proper schema design is critical to support efficient data access and updates during gameplay. Our system does not require a user to sign-up. Therefore, we will never be storing sensitive information such as emails and passwords. Our schemas consist of two tables; Players and Games. Players will hold the name, stack size, cards, and the table ID at which their seated. Games will hold the status (active / inactive), the gamestate, the time created, and the time of the last update. The gamestate will utilize the players table to access the players seat, stack size, etc along with the current round of betting such that when a new player joins they will be able to see the current game state. The schema designs are made in a way to minimize redundancy and optimize query performance. We are using the best practices of normalization while also considering practical denormalization for frequently accessed data. In our testing, we ensure that the database properly updates the gamestate information with the use of SQL scripts to populate our tables with initial data.

3.5 Conclusion

We are making headway on our project and are tackling each issue as it comes. By carefully implementing our socket emissions and constructing Docker images for each component of

RamBluff, we ensure a scalable, secure, and manageable deployment process. While we are in our preliminary testing phase it is crucial to identify and resolve any integration issues early in our deployment cycle. Through thoughtful code and data schema design and management, RamBluff is positioned to offer a robust and engaging gaming experience.