Developing a Real-Time Multiplayer Blackjack Game

presented by

mdbrnd

Contents

Preface

The inspiration for creating a blackjack game came from my experiences with mobile blackjack with friends. Losing large amounts of virtual currency, especially with high bets, was a frustration I often faced, almost too often to align with the actual odds. This gave me the idea to develop my own blackjack game so I could be sure of the fairness of the game. I also wanted to build this project as a way to learn more about web development. My aim with this project is to explore the technical side of building a real-time multiplayer web application. By doing so, I can learn a lot about the underlying technologies of the web we visit every day, help others improve their skills and enjoy the game while ensuring a fair playing experience.
I named the game "Quick21" because I wanted to create a fast-paced experience that allows players to jump in and play without unnecessary delays, such as confirming their email or having to find the correct game mode and lobby out of 1000 other ones. The "Quick" part of the name emphasizes the fast experience, while "21" points to the central goal of blackjack.

1 Introduction

Blackjack is a popular card game played between one or more players and a dealer. This project focuses on building a real-time multiplayer version of blackjack.
The problem I aim to address is the complexity of implementing the core mechanics of blackjack in an accessible, multiplayer environment while ensuring fairness and a simple user experience. Real-time communication between players, fairness of the game mechanics, and accessibility are key challenges in multiplayer game development.
This project will focus on the technical aspects of developing such a web-based multiplayer game, including handling game state synchronization, ensuring fairness in the game, and making it accessible to a wide range of users. It will not delve into more advanced aspects of game theory or gambling systems.
This accompanying paper seeks to answer the central question: **How can the mechanics of blackjack be implemented in an accessible multiplayer application?**

2 The Game of Blackjack

2.1 The Evolution of Online Blackjack

To understand the game of blackjack, we also need to know a bit about its history and how it developed into the game we recognize today[1]. Blackjack, also known as 21, has been a popular casino card game for centuries. Its origins trace back to European card games in the 16th and 17th centuries, particularly the Spanish game "Twenty-One" (Veintiuno). It gained popularity in France during the 18th century as "Vingt-et-Un" and evolved into its modern form in North America. The unique 3:2 (1.5x) payout for a two-card hand totaling 21, known as a "blackjack," added to its appeal. Over time, blackjack has adapted to different casino settings and player preferences, ensuring its lasting popularity[2]. The transition to the online platform marked a significant milestone in the history of card

[1]Blackjack Champ, 2024; Lane, 2024; OnlineCasinos UK, 2024.
[2]Blackjack Maverick, n.d.; Casino.com, 2022; OnlineCasinos UK, 2024.

games, bringing the excitement of the casino into the homes of millions of players. The evolution of online blackjack began in the late 1990s, coinciding with the rise of the internet and advancements in digital technologies. This transition provided opportunities for enthusiasts to enjoy the game without the geographical constraints that were previously associated with traditional casinos[3].
As internet speeds and digital security improved, online blackjack evolved from simple, single-player games to more complex multiplayer environments that included real-time interactions and live dealers. These innovations not only boosted the game's popularity, but also introduced new challenges, particularly in maintaining fair play, ensuring game integrity, and providing a seamless user experience. The rise of mobile platforms further expanded access to blackjack, allowing players to enjoy their favorite games anywhere and at any time, contributing to the rapid growth of the online casino industry.

2.2 Basic Rules and Gameplay

In the following chapters, the basic rules and gameplay of blackjack are explained[4]. Blackjack is played between one or more players and a dealer. The objective is to have a hand value closer to 21 than the dealer's without exceeding 21.
It is typically played with one to eight standard 52-card decks, depending on the casino or game (most casinos use a shuffler and multiple decks to avoid card counting, as discussed in Section 2.3.2). Cards numbered 2 through 10 are worth their face value, face cards (King, Queen, Jack) are worth 10, and Aces can count as either 1 or 11, depending on which value benefits the hand. Each round of blackjack begins with players placing their bets. Once the bets are placed, the dealer starts from the far left, giving each player one up card and a card for themselves. Then, they deal a card to each player and a card to themselves. Each player and the dealer end up with two cards. The players' cards are all visible, while the dealer has one card face up (known as the "up-card") and one face down.
Players then take turns, starting from the player seated *furthest to the dealer's left* (anticlockwise). Each player can choose from several possible actions on their turn:

- **Hit**: The player requests another card from the dealer. The player can continue hitting until they either reach 21 or exceed 21, in which case they "bust" and automatically lose their bet, even if it turns out that the dealer also busted.
- **Stand**: The player chooses to keep their current hand and ends their turn without taking additional cards.
- **Double Down**: The player doubles their original bet and commits to receiving only one additional card, regardless of its value.
- **Split**: If a player has two cards of the same value, they can choose to split them into two separate hands. Each hand is then played independently, and the player must place an additional bet equal to their original bet for the second hand.
- **Surrender**: Some casinos allow players to surrender their hand if they feel it is unlikely to win. In this case, the player forfeits half of their original bet and ends

[3]C., 2022.
[4]Blackjack Apprenticeship, n.d.-a; Blackjack Champ, 2024; KamaGames, n.d.; WilliamHill, n.d.

the round for themselves. This option is typically only available on the player's first action and may vary depending on the casino's rules.

- **Insurance**: If the dealer's up-card is an Ace, players have the option to take insurance. This is a side bet that the dealer has a natural blackjack (an Ace and a 10-value card). The insurance bet is half of the player's original bet, and if the dealer does have blackjack, the insurance bet pays 2:1. For example, if a player bet \$100 and insured his bet for an additional \$50 and the dealer turns out to have a natural blackjack, the player will break even since he lost \$100 from his main bet but gained \$100 from the insurance bet. If the dealer does not have blackjack, the insurance bet is lost, and the game continues as normal.

Once all players have taken their turns, it is the dealer's turn. The dealer reveals their hidden card and must follow a set of rules for their actions:

- The dealer must hit until their hand totals 17 or more.
- If the dealer's hand exceeds 21, they bust, and all remaining players win the round.
- If the dealer's hand is between 17 and 21, the dealer compares their hand to each player's hand. Any player with a higher hand value than the dealer wins, while those with a lower hand lose their bet. If the hand values are equal, the round is a tie or "push," and the player's bet is returned.

Example: Suppose a player is dealt a 10 and a 6, while the dealer's up-card is a 7. The player decides to "hit" and receives a 4, making their hand value 20. The player then chooses to "stand". The dealer reveals his hidden card as 9, meaning their hand totals 16, so they must "hit." The dealer draws a 6, thus their hand exceeds 21 (a total of 22), and they "bust," meaning the player wins the round.
Ace Example: If a player is dealt an Ace and a 6, the Ace can count as either 1 or 11, depending on which value benefits the hand. In this case, the hand at the start is counted as 17 (Ace as 11). If the player were to "stand", their turn would be over and their hand value would be 17. If the player decides to "hit" and pulls a 9, the ace would now change its value to make it better for the player: the hand value is now 16 (1 + 6 + 9). If the ace were to remain as 11, the player would bust.

2.3 Strategies and Decision Making

Strategic decision-making is essential for success in blackjack[5]. Players must choose between actions based on both their current hand and the dealer's up-card. The dealer's visible card provides critical information that helps estimate the strength of their hand, guiding the player to make more informed decisions. Strategies can range from simple rules, like "always stand on 17 or higher," to more advanced techniques rooted in statistical probabilities and even card counting. These approaches aim to maximize potential winnings and minimize losses by exploiting favorable scenarios or reducing the casino's house edge[6].

[5]Blackjack Apprenticeship, n.d.-a, n.d.-b.
[6]Blackjack Apprenticeship, n.d.-b.

2.3.1 Basic Strategy

Basic strategy is a mathematically derived set of guidelines that provides optimal decisions for every possible player hand against every possible dealer up-card[7]. By following these guidelines, players can reduce the house edge[8]. For example, if a player has a hand totaling 12, they should hit if the dealer's up-card is a 2 or 3, but stand if the dealer shows a 4, 5, or 6, as the dealer is more likely to bust with these cards. Adhering to basic strategy significantly improves a player's odds, ensuring they are making the most statistically favorable decisions at every turn.

2.3.2 Card Counting

Card counting is a method players use to track the ratio of high cards (10s, face cards, and Aces) to low cards (2-6) remaining in a deck[9]. A higher count of high-value cards benefits the player, while a deck rich in low cards favors the dealer. By keeping a running count, players can adjust their bets and strategies accordingly. While card counting is not illegal, casinos strongly discourage the practice and often ban players suspected of using it. Mastering card counting requires memory, focus, and subtlety to avoid detection by casino surveillance systems.
Online blackjack makes card counting obsolete. The use of continuous shuffling machines (CSMs) (which are also nowadays found in many in-person casinos) that reshuffle the deck after each round, makes it impossible to gain an advantage through card counting. As a result, card counting strategies are ineffective in online environments, where automated monitoring software detects patterns consistent with card counting and takes measures to counteract them[10]. In the long run, the house will always have the edge. While players may win in the short term, the odds are designed to favor the casino over time. This built-in advantage, known as the "house edge," ensures that, statistically, the casino will come out ahead as more hands are played. It is the balance of risk and reward that keeps the game exciting, but ultimately, the house is designed to profit[11].

2.4 Ethical and Legal Considerations

Although the blackjack game developed in this project does not involve real money, ethical considerations are still essential to address. Fairness, security, and responsible gaming are the core aspects to consider, particularly due to the potential for gambling addiction, even in non-monetary settings.

Fairness: The game employs a reliable random number generator (discussed in Section 3.5.2) to shuffle and deal cards. This ensures that outcomes are based purely on chance, giving all players an equal opportunity to win. This aligns with the fairness expectations of digital gaming environments.

Security: Safeguarding player data is a priority. The game collects minimal user data, limited to usernames, virtual balances, and hashed passwords. By hashing passwords,

[7] Blackjack Apprenticeship, n.d.-b.
[8] Blackjack Apprenticeship, n.d.-b.
[9] Blackjack Apprenticeship, 2024b.
[10] Blackjack Apprenticeship, 2024a, 2024b.
[11] Investopedia, 2024.

sensitive information is protected, even in the case of a breach[12]. Moreover, the virtual balance is only used to track progress in the game, holding no real monetary value, which reduces risks associated with financial exploitation. Further details on the implementation can be found in Section 3.5.2.

Responsible Gaming: The game is designed solely for entertainment purposes, with no real money involved. It is intended for private use among a small group of players, rather than for public distribution. This limited scope reduces ethical concerns typically associated with gambling. The controlled environment, where players are known to each other, potentially minimizes risks of exploitation or problematic behavior.
That being said, ethical concerns can still arise from simulated gambling. A study by Fielder et al. (2023) found a strong link between simulated gambling (SG) and real money gambling (RMG). The study revealed that 17% of participants and 54% of problem gamblers reported that simulated gambling led them to real money gambling (Fiedler et al., 2023). However, in the context of this project, these risks are mitigated by the limited number of users and the absence of marketing.

Legal Considerations: In Switzerland, simulated gambling games are subject to regulatory scrutiny under the Money Gaming Act (Geldspielgesetz)[13]. However, games where participation requires no monetary stake and offers no real-value prizes are generally exempt from the strictest provisions of the law. The law states: "[...] Nicht unter das Geldspielgesetz fallen Gratisspiele, die einen Gewinn in Aussicht stellen, deren Teilnahme aber keinerlei Einsatz in Geld, [...] erfordert, oder Spiele, [...] bei denen keine Möglichkeit eines Gewinns in bar oder in Form von Sachwerten besteht." (Bundesamt für Justiz, 2020).

3 Implementation and Evaluation

To implement a real-time multiplayer blackjack game, a robust architecture is essential for handling both the game logic and player interactions. I chose to make the game web-based because it offers significant advantages in terms of accessibility and ease of use. By developing a website, I ensure that players can access the game from any device with a browser, without the need to install additional software, making it playable on essentially any platform that supports a web browser. Additionally, building the game as a web application gave me valuable hands-on experience in web development which is a field in which I had little experience, having only dabbled in small projects before. I opted for a client-server architecture using WebSockets[14]. This approach allows for seamless data flow between the server and client, effectively separating their responsibilities while ensuring real-time updates and smooth communication.

3.1 Scope of the Project

The game is intended for a small group of players with minimal traffic. In Quick21, certain actions common to traditional blackjack are not available. This is in line with the core

[12]Auth0, 2019.
[13]Bundesamt für Justiz, 2020.
[14]PubNub, n.d.-a; Terra, 2024.

philosophy of the game, which is focused on providing a fast, minimal and streamlined experience.

- No Split: In traditional blackjack, splitting slows down gameplay by essentially creating two hands (and possibly more later) from one. Since Quick21 aims to keep the action moving quickly, the option to split is excluded to maintain a fast-paced, straightforward gameplay experience.
- No Surrender: Allowing players to surrender can reduce the intensity and suspense of the game. By removing this option, Quick21 encourages players to fully commit to their decisions, adding to the excitement and making each choice feel more consequential.
- No Insurance: Insurance can slow down the game, particularly when players have to decide whether to take a side bet. Since Quick21 is designed to offer a simple, fast game without additional layers of complexity, the insurance option has been removed.

3.2 Client-Server Communication

The client-server approach is a model in computing where tasks or services are divided between two main components: the client and the server[15]. This approach is essential because it allows the server to manage the core game logic and maintain an authoritative game state, ensuring that all players have the same view of the game. In this context, the client refers not only to the application that players interact with, but also to the user's device itself. The server handles tasks like dealing cards, calculating scores, and determining winners, while the clients are responsible for displaying the game interface and capturing player inputs. Without this approach, if the game logic were handled on the client side, a bad actor could manipulate the game's outcome by altering data before sending it to the server. This separation of tasks between client and server is essential for security, as it ensures the server controls the critical parts of the game, keeping everything fair and synchronized[16].

3.3 TypeScript

In choosing the technology stack for this project, I opted for TypeScript as the programming language for both client and server. Since I was not yet very familiar with the world of web development, choosing something popular and well-documented definitely helped. Along with choosing something popular, my prior experience with statically typed programming languages such as C# made TypeScript all the more appealing.
TypeScript is a superset of JavaScript that introduces static typing to the language[17]. This means variables, function parameters, and return values are assigned specific data types, which are checked before the code runs. This type system helps developers catch errors earlier in the development process and write more robust code[18]. It helps ensure that data is handled correctly throughout the application, which is crucial in a multiplayer

[15]Terra, 2024.
[16]Terra, 2024.
[17]TypeScript Team, n.d.-b.
[18]TypeScript Team, n.d.-b.

game like blackjack where data integrity and consistent behavior across different clients are essential. Type related errors, such as assigning a text (string) to a variable that is meant for a number, will be caught by the compiler while writing the code, whereas with JavaScript such errors might go unnoticed until the application is already running, or even worse, produce unexpected behavior without throwing any errors.

3.4 Tools and Frameworks

Apart from TypeScript, some additional tools and frameworks were used during development.

3.4.1 React, Create React App and Tailwind CSS

React is a popular JavaScript library (with TypeScript support) for building user interfaces[19]. I chose React for this project mainly because of its popularity, strong community support, and the component-based architecture that simplifies the management of reusable elements. While learning any framework is not necessarily easier than not using a framework, React's flexibility and component-based structure fit the needs of the project more than vanilla HTML, CSS and JavaScript. The component-based architecture of React allows the UI to be broken down into reusable parts which made maintaining and extending the game easier[20]. React's extensive ecosystem and available resources made it more accessible for troubleshooting and learning. Alternatives, like Angular[21] and Vue[22], were also considered, but I felt React offered more documentation and community-driven tutorials.
Create React App (CRA) is a widely used tool that provides a quick way to set up a new React project with sensible defaults out of the box[23]. I chose CRA because it was a tool I had heard of and was already somewhat familiar with, which made the setup process faster and more straightforward. CRA handles the configuration and provides some helpful features such as instant reload, allowing me to focus on coding the core functionality of the game rather than setting up the environment manually.
While other tools like Next.js[24] could offer more flexibility in specific use cases, CRA's simplicity and familiarity were the primary reasons for choosing it. Since the focus of this project was on building functionality quickly, CRA's "batteries-included" approach initially seemed like the perfect fit for Quick21—though, as I later discovered, it came with limitations (discussed in Section 4.1).
CSS (Cascading Style Sheets) is a language used to describe the look and formatting of a document written in HTML, allowing developers to control layout, colors, fonts, and other visual elements[25]. At the start of the project, I used vanilla CSS for styling. However, as the project grew, maintaining and customizing styles became more difficult. Eventually, I switched to Tailwind CSS, a "utility-first CSS framework packed with classes [...] that can be composed to build any design, directly in your markup" (Tailwind CSS, 2024). This change made the styling process much simpler, especially when working

[19]React Team, n.d.
[20]React Team, n.d.
[21]Angular Team, n.d.
[22]Vue.js Team, n.d.
[23]Create React App, 2024.
[24]Vercel, 2024.
[25]W3Schools, n.d.

with React, as Tailwind integrates smoothly with component-based architectures. I also considered a framework called Bootstrap[26], but Tailwind offered more flexibility and cleaner integration with React components without the overhead of additional class names or custom styles. Tailwind's utility-first approach not only simplified the redesign but also created a more efficient workflow by allowing me to apply styles directly to components, making the process much faster compared to writing custom styles in vanilla CSS[27].

3.4.2 Socket.IO

WebSockets are a communication protocol that allows for real-time, two-way communication between a client and a server over a single, persistent connection. Unlike traditional HTTP, which opens a new connection for each request, WebSockets keep the connection open, enabling instant updates[28]. This is crucial for multiplayer games where players need to see changes as they happen.
Socket.IO is a library that simplifies using WebSockets by providing an easy interface for handling real-time events[29]. Socket.IO is used to manage key game interactions like joining rooms, placing bets, and updating the game state, ensuring all players receive real-time updates without delay. Another major reason for choosing WebSockets was that it was the most common solution I encountered while learning and researching how to build a multiplayer game[30], which made the development process smoother and more intuitive as I already had the resources I needed.
I also considered using vanilla WebSockets, but managing events and handling fallbacks for different environments seemed unnecessarily complex. Another technique called HTTP long polling could have been used as an alternative[31]. In long polling, the client sends an HTTP request to the server, and the server holds the request open until new data is available before responding. Once the server responds, the client immediately sends another request to wait for the next update, simulating real-time communication. While this approach can deliver updates relatively quickly, it creates significant overhead due to the repeated opening and closing of connections for each update, leading to higher latency compared to WebSockets, which maintain a persistent, bidirectional connection and are a "as-close-to-raw-as-possible TCP communication layer" (Kilbride-Singh, 2023). Socket.IO simplifies real-time communication and also provides many useful features such as middlewares, callbacks and error handling[32]. This flexibility made Socket.IO the most reliable and efficient choice for my project.

3.4.3 Node.js and Express

Node.js is a JavaScript runtime (with TypeScript support) that enables server-side development[33]. It is used on the server side to handle user authentication, specifically the sign-up and sign-in processes. Users provide their username and password to log in. Upon

[26]Bootstrap, 2024.
[27]Tailwind CSS, 2024.
[28]PubNub, n.d.-a.
[29]Socket.io Developers, 2024.
[30]freeCodeCamp, n.d.-a; GitHub User "Polivodichka", 2023; YouTube User "Chris Courses", 2023; YouTube User "Dave Gray", n.d.
[31]Diaconu, 2024; Kilbride-Singh, 2023; PubNub, n.d.-b.
[32]Socket.io Developers, 2024.
[33]Node.js Foundation, 2024.

successful authentication, JSON Web Tokens (JWT) are generated and used to manage user sessions (further discussed in Section 3.7).
Python with Django was considered as an alternative, but Node.js was the clear choice because I knew the game and state management on the server were going to be implemented in TypeScript using Socket.IO. Keeping everything within the same language ecosystem made development more streamlined and reduced the complexity of integrating different technologies. Additionally, Django appeared to involve more boilerplate code and greater complexity compared to Node.js[34].
I used Express to simplify building the server-side API. "Express is a small framework that sits on top of Node JS's web server functionality to simplify its APIs and add helpful new features." (GeeksforGeeks, 2024). It made it easier to configure middlewares compared to vanilla Node.js, and having already heard about it and knowing it was popular, meant there were plenty of resources and documentation available[35].

3.4.4 SQLite

SQLite is a lightweight, self-contained, and serverless database engine[36], which made it an ideal choice for this project due to its simplicity and ease of use. I considered more robust solutions including PostgreSQL[37] or MySQL[38], but given the small-scale nature of this project, these options felt like overkill. Since this project is intended for smaller-scale usage, SQLite's in-memory database capabilities provided an efficient and fast solution without the need for complex database configurations.
SQL (Structured Query Language) is a standardized language used to communicate with databases, allowing users to perform operations like querying data, updating records, and managing database structures[39]. SQL commands such as `SELECT`, `INSERT`, `UPDATE`, and `DELETE` enable interactions with the database to retrieve or modify information[40]. I decided to use a SQL database because I have some experience with it, making it easier to manage the structured data required for this project.
SQLite stores all data in a single file, making it easy to manage and transport. Additionally, it supports ACID (Atomicity, Consistency, Isolation, Durability) properties, ensuring that transactions are handled reliably[41]. This is crucial for maintaining data integrity, particularly when managing players' balances, game state, and user authentication. It is also important to be aware of potential race conditions in a concurrent environment. SQLite handles multiple database connections sequentially, meaning if two processes try to write to the database at the same time, one will be blocked until the other is finished. Although SQLite is generally safe for low-traffic applications, in a scenario with high concurrency, these race conditions could result in inconsistent data, especially if multiple operations (such as reading and updating the same data) happen in quick succession.
Security is always a concern when working with databases, especially when handling user inputs. One of the major risks is SQL injection—an attack where malicious users can input SQL commands through fields intended for normal user data (like usernames or

[34]Django Software Foundation, 2024; Node.js Foundation, 2024.
[35]Express.js Team, 2024; GeeksforGeeks, 2024.
[36]SQLite, 2024.
[37]PostgreSQL, 2024.
[38]Oracle, 2024.
[39]Wikipedia, n.d.-c.
[40]Wikipedia, n.d.-c.
[41]SQLite, 2024.

search boxes)[42]. Without proper safeguards, these malicious inputs could be executed by the database, leading to data breaches or unauthorized actions.
To mitigate the risk of SQL injections, all SQL queries were prepared using parameterized statements. Parameterizing a query means separating SQL code from the data inputs provided by users[43]. Instead of embedding user inputs directly in the SQL string, placeholders (such as `?`) are used, and the actual user data is bound separately when the query is executed. This ensures that user inputs are treated as plain data and not executable code, preventing attackers from injecting harmful SQL commands.
The single database.sqlite file stores the list of users containing only their username, hashed password, and balance.

3.5 Conceptualizing the Game of Blackjack

This chapter assumes a basic understanding of programming concepts including data types and Object-Oriented Programming. To effectively abstract the game of blackjack into code, Object-Oriented Programming (OOP) principles were employed. Utilizing OOP allowed for the creation of classes to encapsulate various components of the game, enhancing modularity and reusability. The use of TypeScript's type system and structural features further simplified the development process, enabling a clear representation of game entities and their interactions.

3.5.1 Program Architecture

The backend architecture of the game is organized around three key classes: `RoomManager`, `Room`, and `Game`. These classes interact with each other to manage the game state, players, and game logic.
The `RoomManager` class is responsible for handling multiple rooms, each identified by a unique code. Each `Room` contains a list of players, with one designated as the owner, and manages a `Game` instance that controls the game state. Players are connected through Socket.IO rooms, which allow real-time updates to be sent to all players within the same game room. This ensures that each player receives timely updates about the game state, such as card draws and score changes, keeping the gameplay synchronized for everyone involved. A 6-digit room code is used as a unique identifier.
The `Game` class encapsulates the state of the game, including the deck of cards, players' hands, and the current phase of the game. This modular design ensures that each component has a well-defined responsibility, making the code easier to maintain and extend.
The game flow begins with the user logging in and establishing a connection to the server. Once connected, the user can either create a new room or join an existing one.
If a user creates a room, they become the owner and can invite other players. Once enough players have joined, the game transitions to the waiting screen before the game begins. Conversely, if a user chooses to join an existing room, the application checks if the room exists and if there's space for more players.

[42] OWASP, n.d.-b.
[43] OWASP, n.d.-a.

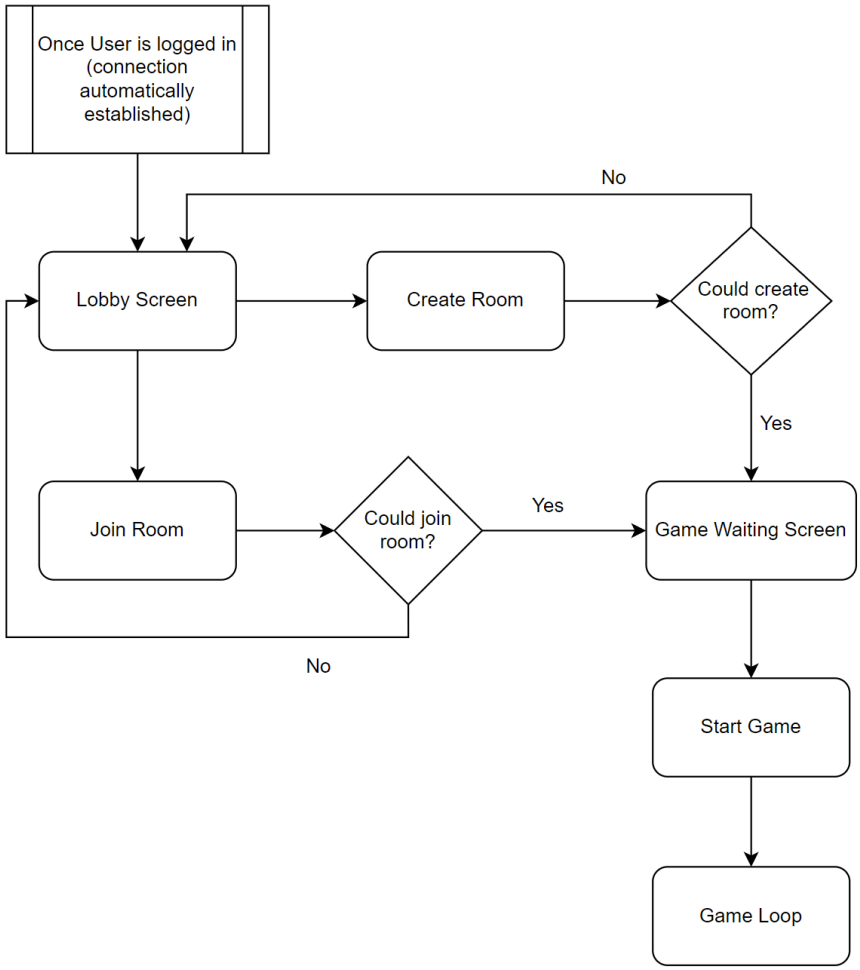

Figure 1: Game Flow Diagram: Overview of the Room Joining and Game Starting Process

Figure 1 highlights the key decision points and the transitions between different screens. Once the game has started, it operates in a loop that progresses through three main phases: `Betting`, `Playing`, and `RoundOver`.
In the `Betting` phase, players place their bets, which are registered by the server. Once all bets are placed, the game transitions to the `Playing` phase, where players take turns. The server updates the game state after each action, ensuring that all clients have a consistent view of the game.
Finally, once every player has taken their turn, the round is over. During the `RoundOver` phase, the game calculates the results based on the hands of the players and the dealer, updates the balances, and prepares the game state for the next round.

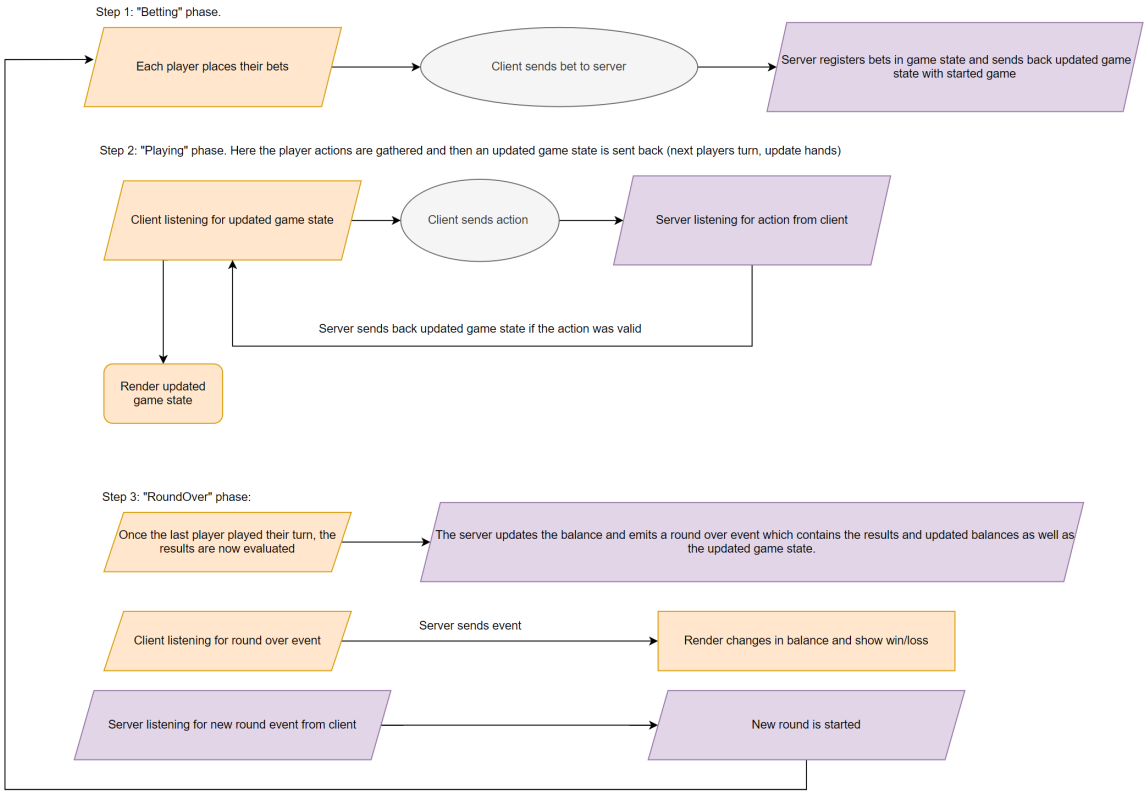

Figure 2: Game Loop: Steps from Betting Phase to RoundOver

Figure 2 illustrates this loop in detail, showing how each phase is interconnected and how the server and clients interact during the game. This design ensures a smooth and fair gaming experience, where all players are synchronized throughout the game.

3.5.2 Key Implementation Details

As mentioned earlier, a critical aspect of implementing blackjack is managing the game state, which is stored server-side in an instance of the `ServerGameState` class. This class is responsible for tracking all essential elements of the game, including the deck, dealer's hand, current turn, players' hands, game phase, and bets. The class definition is as follows:

```typescript
// server/src/models/game_state.ts

class ServerGameState {
  gameStarted: boolean;
  deck: Card[];
  dealersHand: Card[];
  currentTurn: Player;
  playersHands: Map<Player, Card[]>;
  currentPhase: "Betting" | "Playing" | "RoundOver";
  bets: Map<Player, number>;

  constructor(
    gameStarted: boolean,
    deck: Card[],
    dealersHand: Card[],
    currentTurn: Player,
    playersHands: Map<Player, Card[]>,
    currentPhase: "Betting" | "Playing" | "RoundOver",
    bets: Map<Player, number>
  ) {
    this.gameStarted = gameStarted;
    this.deck = deck;
    this.dealersHand = dealersHand;
    this.currentTurn = currentTurn;
    this.playersHands = playersHands;
    this.currentPhase = currentPhase;
    this.bets = bets;
  }

  toDTO() {
    return this.toClientGameState().toDTO();
  }

  toClientGameState(): ClientGameState {
    const dealersVisibleCard = this.dealersHand[0]
    return new ClientGameState(
      this.gameStarted,
      dealersVisibleCard,
      this.currentTurn,
      this.playersHands,
      this.currentPhase,
      this.bets
    );
  }
}
```

Figure 3: The server game state

This class encapsulates the logic for managing the state of the game, ensuring that the server efficiently handles the progression of the game. Notably, the `toDTO` and `toClientGameState` methods facilitate the secure transfer of game state data from the server to the client. The `toDTO` method is responsible for serializing the game state data,

converting it into a format that can be easily transferred over the network. This involves transforming custom, more complex data types, such as `Map<Player, number>`, into JSON-compatible formats, ensuring that key-value pairs are correctly handled ("Map and Set are not serializable and must be manually serialized" (Socket.io Developers, 2024)). Crucially, the `toClientGameState` method only sends the client a limited view of the game state, excluding sensitive information like the dealer's full hand or the remaining deck. This ensures that players cannot access information that could give them an unfair advantage before the round concludes. These methods are critical in maintaining the integrity of the game and preventing the premature revelation of crucial data.
On the client side, the game state is updated dynamically through an event listener that reacts to updates received from the server:

```
1 // client/src/pages/GameScreen.tsx
2
3 socket.on("game-state-update", (newGameState: any) => {
4   newGameState = ClientGameState.fromDTO(newGameState);
5   updateGameState(newGameState);
6 });
```

Figure 4: The game state listener

The deck is initialized as a standard 52-card deck, which is shuffled before each round to ensure fairness and randomness. If the deck runs out of cards during gameplay, a new one is generated to maintain a continuous game flow.
The betting system is implemented using a `Map<Player, number>` structure, which associates each player with their respective bet amount. A `place-bet` event is emitted to the server once a bet is placed. Once all bets are placed, the game transitions to the playing phase, during which players receive their initial cards and make decisions such as hitting or standing.
The following interfaces outline the structure of the `Player` and `Card` entities:

```ts
// server/src/models/player.ts
interface Player {
  socketId: string;
  name: string;
  balance: number;
  userId: number;
}

// server/src/models/card.ts
interface Card {
  value: string;
  suit: string;
}
```

Figure 5: The Player and Card models

To keep the game fair, all the main game logic, like card shuffling, dealing, and determining outcomes, is done on the server. This approach prevents any manipulation from the client side or interference by third parties. As mentioned earlier, the core game logic is implemented in the `Game` class on the server side. One such method in the `Game` class is the `hit` function, which handles drawing additional cards for players:

```ts
// server/src/game.ts

public hit(playerSocketId: string) {
    const player = this.getPlayerHandBySocketId(playerSocketId);
    if (player) {
      // if deck is empty, create a new deck
      if (this.state.deck.length === 0) {
        this.initializeDeck();
      }
      player.push(this.state.deck.pop()!);
    }
}
```

Figure 6: The hit function in the Game class

The deck shuffling is performed using the Fisher-Yates shuffle[44] algorithm, which iterates over the cards in the deck and swaps each card with another randomly selected one. The selection of a random index is based on the `Math.random()` function provided by JavaScript, which uses a pseudo-random number generator (PRNG). The latest JavaScript implementation of `Math.random()` relies on the xorshift128+ algorithm for random number generation[45]. For more information on the PRNG algorithms used in JavaScripts `Math.random()`, refer to Guo, 2015.

[44]Bostock, 2012; Wikipedia, n.d.-b.
[45]Guo, 2015; MDN Web Docs, 2023.

While xorshift128+ produces highly random sequences, it is important to note that this is still pseudo-random, meaning that it follows a deterministic process that can be reproduced if the same initial seed is used. It is not "truly random" in a mathematical sense, but for most practical purposes, including deck shuffling in a game, the level of randomness provided by this algorithm is sufficient[46].

```typescript
// server/src/game.ts

// Generated with the help of GitHub Copilot
private shuffleDeck(deck: Card[]): Card[] {
  for (let i = deck.length - 1; i > 0; i--) {
    const j = Math.floor(Math.random() * i);
    const temp = deck[i];
    deck[i] = deck[j];
    deck[j] = temp;
  }

  return deck;
}
```

Figure 7: The function used to shuffle the deck

3.6 User Interface and Animations

The user interface (UI) of the game is designed to be both intuitive and visually appealing, though it is not the primary focus of the project. The layout is crafted with simplicity in mind, making it easy for players to navigate through the game. Key elements, such as player actions and game state indicators (e.g., hand values, dealer's cards), are placed to offer clarity during gameplay. To further enhance the visual experience, animations are used throughout the game. For example, cards are dealt with smooth transitions, making the game feel more natural and polished.

[46] Guo, 2015; MDN Web Docs, 2023; TypeScript Team, n.d.-b.

Figure 8: The main game screen

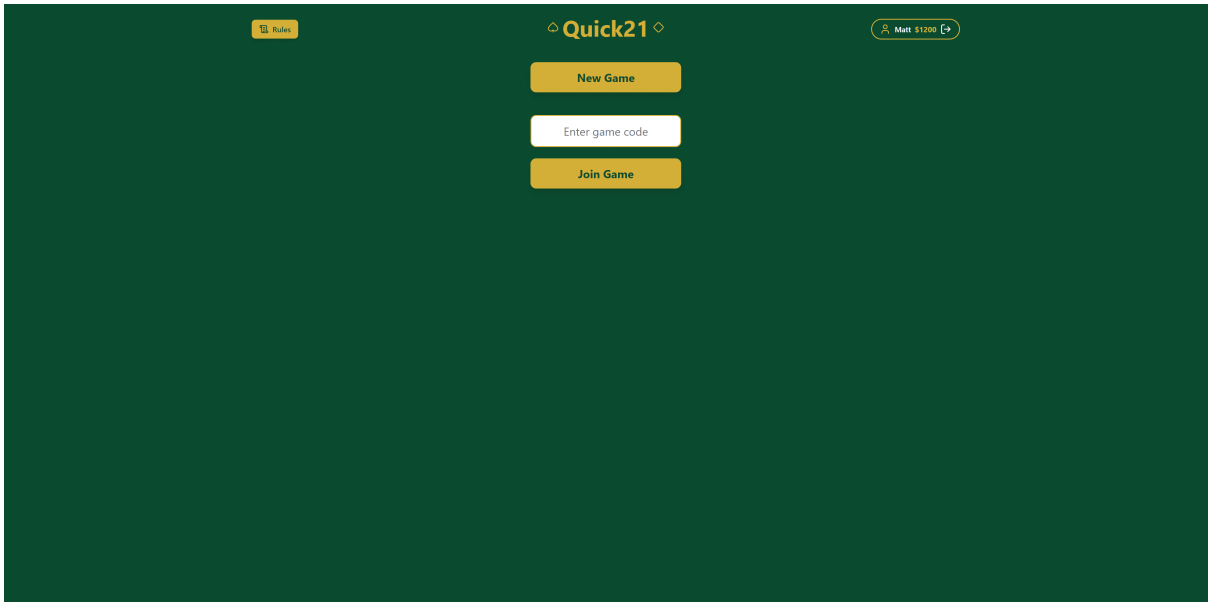

Figure 9: The lobby screen

The logo/favicon for the game was created using generative AI within ChatGPT, which provided the initial concept design. Afterward, the logo was refined and enhanced using photo editing software to remove any artifacts.

Figure 10: The Quick21 logo and favicon

Similarly, the rule images, which visually explain the blackjack game rules, were also generated using AI and then refined.
The styling and animations are implemented with Tailwind CSS[47].
The card images used in the game were sourced from Google[48], providing a set of visually clear and recognizable playing cards that fit seamlessly into the interface.

3.7 Authentication

The authentication system in this project is designed to provide secure user access while maintaining a seamless user experience. As already mentioned, it utilizes JSON Web Tokens for authentication and integrates with Socket.IO for real-time communication.
I chose not to store or ask for users' email addresses because of the scope of this project, so additional user information like an email address was not necessary. Since email is typically used for features like password recovery or notifications, and those features were not relevant for this project, keeping the data minimal with just the username, hashed password, and balance simplified both the user experience and data management, but naturally also made it not directly possible to recover a lost account without contacting an administrator (me).
A key aspect of the authentication system is combining WebSockets with authentication tokens (JWT). When a user registers, their password is hashed using bcrypt[49], a library to hash passwords, before being stored in the database. Unlike encryption, hashing is a one-way process and cannot be reversed[50]. During login, the provided password is compared with the stored hash. Upon successful authentication, a JWT authentication token, containing the user id and name, is generated using a secret key (`JWT_SECRET`) and sent back to the client. The `JWT_SECRET`, which is stored in a `.env` file for security purposes, is essential for signing and verifying the token to ensure that it has not been tampered with. The `JWT_SECRET` is a unique key used in the encryption process to securely sign tokens, ensuring their authenticity. It is crucial that this secret key is stored only on the server side (and also not checked into source control), as exposure of the `JWT_SECRET` would compromise the security of the entire authentication system. If someone gains access to the secret key, they could forge valid tokens, rendering the encryption useless. Since the authentication tokens are not stored in the database server-side, multiple valid authentication tokens can exist at the same time, which is not the most secure approach.

[47] CSS Tricks, 2024; Mozilla Developers Network (MDN), n.d.; Tailwind CSS, 2024; W3Schools, n.d.
[48] Google, n.d.
[49] bcrypt, n.d.
[50] Auth0, 2019.

However, the tokens automatically expire after a fixed period (14 days in this case), mitigating the risk of long-term unauthorized access.

```
// server/src/server.ts

app.post("/login", async (req, res) => {
  // ... (input validation)
  try {
    const user = await dbManager.getUserByName(name);
    const match = await bcrypt.compare(password, user.password);
    if (match) {
      const token = jwt.sign({ id: user.id, name: user.name }, JWT_SECRET, {
        expiresIn: "14d",
      });

      res.send({
        message: "Login successful.",
        token,
        user: { id: user.id, name: user.name, balance: user.balance },
      });
    } else {
      res.status(401).send({ message: "Password is incorrect." });
    }
  } catch (error) {
    // ... (error handling)
  }
});
```

Figure 11: The code used to authenticate the user and generate the token

The JWT is then stored in the browser's local storage on the client side. The client-side handles token storage and management primarily through the `SocketContext` file found at `client/src/SocketContext.tsx`. This token is then used to authenticate subsequent WebSocket and HTTP connections.

```tsx
// client/src/SocketContext.tsx

const connect = (token: string) => {
  const newSocket = io(API_BASE_URL, {
    auth: { token },
  });

  newSocket.on("connect", () => {
    setSocket(newSocket);
    setIsAuthenticated(true);
    localStorage.setItem("authToken", token);
    // ... (fetch user info)
  });
};
```

Figure 12: The the code used to connect authenticated sockets

The client side maintains the authentication state using React's context API. A `SocketProvider` component manages the socket connection and user authentication state. This provider allows components throughout the application to access the authentication state and socket connection using a custom `useSocket` hook. Logging out involves disconnecting the socket, removing the token from local storage, and updating the authentication state. On the server side, a middleware function authenticates the token for each WebSocket connection. This middleware verifies the JWT and attaches the decoded user information to the socket for future use.

```tsx
// server/src/server.ts

const authenticateToken = (socket: any, next: any) => {
  const token = socket.handshake.auth.token;
  if (!token) {
    return next(new Error("Authentication error: Token not provided"));
  }
  jwt.verify(token, JWT_SECRET, (err: any, decoded: any) => {
    if (err) {
      return next(new Error("Authentication error: Invalid token"));
    }
    (socket as AuthenticatedSocket).user = decoded as JwtPayload;
    next();
  });
};

io.use(authenticateToken);
```

Figure 13: The the code used to authenticate tokens and sockets

3.8 Admin Panel

I also decided to add an admin panel to manage the game's users. It serves primarily as a proof of concept and is only accessible by me. Access to the admin panel is restricted and requires authentication via the same JWT mechanism used for general users. The server checks if the username in the JWT matches that of an admin account to grant access. Currently, the panel allows user balances to be updated, but it can easily be extended in the future to manage usernames and passwords as well.
The admin panel is built using React and styled with Tailwind CSS with help from Anthropics LLM Claude[51]. While real-time communication in the game is handled via WebSockets, requests from the admin panel are made using HTTP requests for simplicity and reliability.

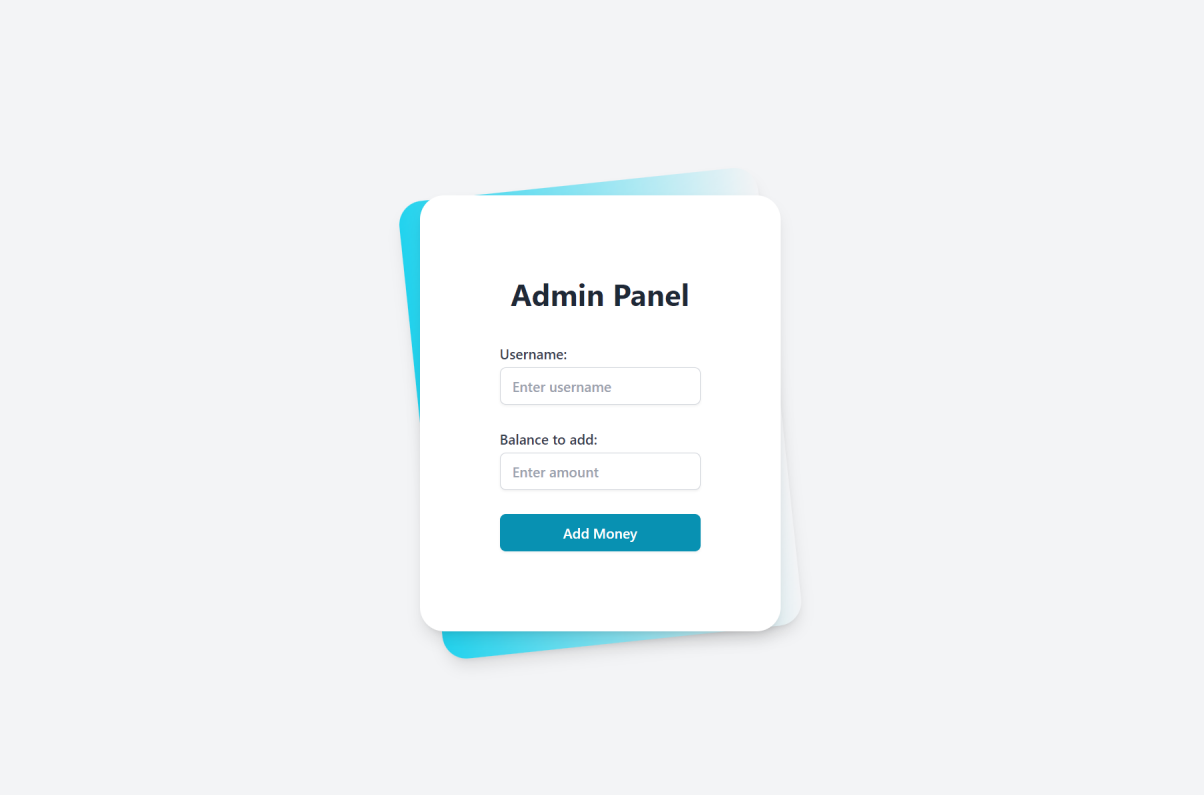

Figure 14: The Quick21 Admin Panel

3.9 Hosting and Deployment

Hosting involves placing the application on a server that can be accessed via the internet, while deployment is the process of transferring the application's code from the development environment to the server[52]. For Quick21 to be playable by others, it needed to be hosted somewhere so users could access the game online. Thus, hosting and deployment are necessary stages in making a web application accessible to users. I used Git[53] and GitHub[54] throughout the project for version control and collaboration to manage code

[51] Anthropic, 2024.
[52] Render, 2024; YouTube User "RoadsideCoder", 2023.
[53] Git, 2024.
[54] GitHub, 2024b.

changes efficiently. My familiarity and prior experience with these platforms made them the natural choice.
The first step in this process was configuring the server to serve the client's build files. Quick21's project structure had separate client and server directories. The server was set up to backtrack into the client folder and access the build directory, allowing it to serve the static files necessary for the frontend. Using Node.js and Express, the server was configured to use the `express.static()` middleware for serving static files.
After ensuring the server could serve the frontend, I evaluated several hosting providers to determine the best option. Some platforms I considered were DigitalOcean[55], Heroku[56], AWS[57] and Render[58]. I ultimately chose Render because it offered the best balance between ease of use and functionality. Render, like most hosting platforms, provided automatic deployment from GitHub, a platform for version control and code hosting that allows developers to manage code repositories and collaborate. With GitHub integration, Render automatically deployed the latest code to the live server whenever changes were pushed to the `main` git branch[59]. This made it so I could commit any changes to my `staging` branch and merge them into the `main` branch whenever I wanted to publish them.
Despite DigitalOcean and AWS technically offering more control and flexibility, they required significantly more manual setup, which was not necessary for a project of this scale. The same can be said for hosting it myself on one of my own devices. Heroku was also a strong contender, but after coming across a tutorial that demonstrated how to deploy an application with a similar structure to mine, I decided to use Render, as it aligned perfectly with the deployment process shown in the video[60].
To host Quick21 on Render, I connected the GitHub repository to the platform, which enabled continuous deployment. I used Node.js to manage API requests and serve the static build files from the client folder. Render took care of the infrastructure setup, ensuring the game was always accessible and up-to-date with the latest code changes, without the need for extensive manual configuration[61].

3.10 Testing and Evaluation

Testing and evaluation were essential to ensuring that Quick21 was both functional and enjoyable. A mix of manual testing, automated tests, and user feedback allowed me to identify and fix bugs, improve performance, and enhance the overall experience.

3.10.1 Manual Testing and Automated Tests

In the early stages, I performed manual tests while developing, focusing on the game's core mechanics, such as dealing cards, calculating hand values, and determining round outcomes. These tests helped uncover issues in how hands were handled and how rounds progressed, especially in edge cases like immediate wins or busts.

[55] DigitalOcean, 2024.
[56] Heroku, 2024.
[57] Amazon, 2024a.
[58] Render, 2024.
[59] Git, 2024.
[60] YouTube User "RoadsideCoder", 2023.
[61] Render, 2024; YouTube User "RoadsideCoder", 2023.

To add more precision, I used Jest, a JavaScript testing framework, to automate some of these checks[62]. Jest allowed me to write tests that ensured the game's logic, particularly the calculation of hand values and round transitions, functioned correctly. For example, the Jest tests confirmed that card totals were accurate and that the game properly handled win, lose, or tie scenarios. Despite the number of automated tests being limited, they helped verify key parts of the game.

3.10.2 User Feedback

Once the game was hosted, I expanded testing to include my dad and friends. Their feedback was crucial in spotting issues that had not been apparent in solo testing. By observing how they played, I discovered bugs (mostly related to authentication), potential UI/UX improvements, and areas where the gameplay flow could be improved. Direct conversations helped me identify what worked well and where improvements were needed, such as simplifying the interface and ensuring the game stayed fast-paced.
The feedback loop was critical in refining Quick21, making it both more intuitive and enjoyable. Iterative improvements, based on these tests, led to a smoother experience, resolving issues like UI overlaps and unclear game state transitions.

3.11 Future Scalability Considerations

Quick21 currently uses SQLite for data storage, which is well-suited for small-scale applications like this, where the game is intended for a small group of players with minimal traffic. SQLite's simplicity and portability make it an ideal choice for current needs, requiring no configuration and operating efficiently in this context.
However, if the player base were to grow significantly, migrating to a more robust database system, such as PostgreSQL, may become necessary. Unlike SQLite, which can encounter limitations due to file locking in high-concurrency environments, PostgreSQL offers better performance for handling larger datasets and higher traffic[63].
The same applies to WebSockets. While they can introduce scalability challenges, they are sufficient for the current needs, provided the player base does not explode. If the user base expands considerably, moving off from WebSockets or implementing a load balancing[64] system could be required.
That said, these scalability considerations extend beyond the current scope of this project.

[62] JestJS, n.d.
[63] PostgreSQL, 2024.
[64] Amazon, 2024b.

3.12 Libraries Used

Name	Description
React	A JavaScript library with TypeScript support for building user interfaces, chosen for its popularity and performance in developing interactive UIs (React Team, n.d.).
TailwindCSS	A CSS framework for rapidly building custom designs, providing flexibility and simplifying the creation of customizable layouts (Tailwind CSS, 2024).
Socket.IO Client	Enables real-time communication between the frontend and backend, ideal for implementing real-time updates in a multiplayer blackjack application (Socket.io Developers, 2024).
Create React App	A tool to set up a new React project quickly with sensible defaults, chosen because it is a familiar option that simplifies the initial project setup (Create React App, 2024).
lucide-react	A library providing beautiful icons in a React-friendly way, enhancing the UI with clean, modern visuals (lucide-react, n.d.).

Table 1: Tools and Libraries Used in the Frontend

Name	Description
Node.js	A JavaScript runtime environment for server-side development (Node.js Foundation, 2024).
Express	A minimal and flexible Node.js web application framework, simplifies the process of building robust APIs and web applications (Express.js Team, 2024).
Socket.IO	Enables real-time communication between server and clients, crucial for implementing the multiplayer functionality in the blackjack game (Socket.io Developers, 2024).
bcrypt	A library to hash passwords, used for secure storage of user passwords (bcrypt, n.d.; Wikipedia, n.d.-a).
cors	A package for enabling Cross-Origin Resource Sharing, allows the backend to handle requests from the frontend securely (cors, n.d.).
dotenv	Loads environment variables from a .env file into process.env, facilitates configuration management without exposing sensitive information in the codebase (dotenv, n.d.).
jsonwebtoken	A package to create and verify JSON Web Tokens, used for secure user authentication and session management (jsonwebtoken, n.d.).
nodemon	A tool that automatically restarts the server on file changes, enhancing development productivity by automatically restarting the Node.js application (NPM Contributors, 2024).
sqlite3	A self-contained, high-reliability, embedded, full-featured SQL database engine, chosen for its simplicity and efficiency in handling the application's database needs (SQLite, 2024).
ts-node	A TypeScript execution engine and REPL for Node.js, allows direct execution of TypeScript files without precompiling them into JavaScript (ts-node, n.d.).
jest	A testing framework for JavaScript with TypeScript support, providing a robust and easy-to-use environment for testing the backend code (JestJS, n.d.).

Table 2: Tools and Libraries Used in the Backend

4 Conclusion

4.1 Reflective Analysis

Looking back on the development process, the research phase posed one of the biggest challenges. Grasping the intricacies of game design, real-time communication, and security required significant effort. I relied heavily on AI tools[65] like ChatGPT, Claude AI

[65] Anthropic, 2024; GitHub, 2024a; OpenAI, 2024.

and GitHub Copilot, as well as YouTube tutorials[66], to understand these concepts more efficiently. These tools sped up my research, helped me troubleshoot issues, and even provided suggestions for implementing certain features. Rather than acting as a crutch, they served as valuable aids that made the more challenging aspects of the project manageable. I also used ChatGPT to assist in drafting and refining parts of this paper, guiding the structure and grammar while benefiting from the tool's suggestions for clarity.
In addition to research, testing played a major role in ensuring the game functioned properly. I conducted extensive manual testing early on, focusing on core game mechanics such as hand value calculations and round outcomes. Later, I also introduced several Jest tests to automate the validation of these critical components. After hosting the game, I invited my dad and friends to help with testing, which revealed user experience issues and bugs that I had not noticed during solo testing. Their feedback played a crucial role in refining the gameplay, enhancing the UI, and ensuring the game remained fast-paced and engaging.
As the project progressed, using Socket.IO for real-time communication made certain aspects straightforward, but it also added complexity as new features were introduced. Managing the growing number of methods and interactions led to a more cluttered and harder-to-maintain codebase.
As mentioned earlier, I opted for Create React App (CRA)[67] to set up the project due to its simplicity and familiarity having made a small project with it before. However, this choice later proved to be a mistake. For example, I was unable to manually change the `.env` variables in certain cases, which became a frustrating limitation[68]. While CRA's default configurations helped me start quickly, this inflexibility made it difficult to customize key parts of the project when needed.
Another key takeaway from this project was learning the importance of setting realistic goals and acknowledging the limitations of the tools and technologies I chose. While there were obstacles and imperfections in the final product, the experience was invaluable, and it is clear that this project has potential for further refinement and improvement. These challenges and the steps taken to overcome them are discussed in more detail in the Arbeitsjournal (Section A.1).
Ultimately, the project achieved its main goal: to create a quick, fair, and enjoyable blackjack game. It provides a solid foundation to build upon in the future, but it also serves as a reminder of the practical limitations that come with scaling a project of this scope.

4.2 Conclusion and Outlook

This paper investigates the implementation of blackjack in a real-time multiplayer web application. The research is driven by the motivation to address the frustrations experienced in existing blackjack games, particularly the perception of unfair outcomes. The approach involved developing a web-based blackjack game using different technologies to allow for an accurate real-time experience.
The key results of the project include the successful creation of a multiplayer blackjack game that faithfully implements the game's core mechanics while ensuring fairness

[66] Dev.to, 2022; freeCodeCamp, n.d.-a, n.d.-b; YouTube User "Chris Courses", 2023; YouTube User "Dave Gray", n.d.; YouTube User "RoadsideCoder", 2023.
[67] Create React App, 2024.
[68] Facebook, n.d., 2017; Stack Overflow, 2019.

through reliable random number generation and server-side game logic.
Looking ahead, there is potential to continue developing Quick21. Improvements could include expanding the game's features, such as adding a global leaderboard. In my opinion, Quick21 is a solid starting point, and with further development, it has the potential to evolve into a more robust multiplayer game. However, for now, I have decided to pause further development due to a lack of spare time.

References

Amazon. (2024a). *Amazon Web Services (AWS)*. Retrieved October 16, 2024, from https://aws.amazon.com/

Amazon. (2024b). *What is Load Balancing?* Retrieved October 16, 2024, from https://aws.amazon.com/

Angular Team. (n.d.). *Angular - web application framework*. Retrieved August 24, 2024, from https://angular.dev/

Anthropic. (2024). *Claude ai* [AI assistant developed by Anthropic]. Retrieved October 15, 2024, from https://www.claude.ai

Auth0. (2019, September). *Hashing passwords: One-way road to security*. Retrieved October 14, 2024, from https://auth0.com/blog/hashing-passwords-one-way-road-to-security/

bcrypt. (n.d.). *A library to help you hash passwords*. Retrieved October 15, 2024, from https://www.npmjs.com/package/bcrypt

Blackjack Apprenticeship. (n.d.-a). *How to play blackjack*. Retrieved October 14, 2024, from https://www.blackjackapprenticeship.com/how-to-play-blackjack/

Blackjack Apprenticeship. (n.d.-b). *Learn blackjack strategy*. Retrieved October 14, 2024, from https://www.blackjackapprenticeship.com/blackjack-strategy-charts/

Blackjack Apprenticeship. (2024a). *Are continuous shuffle machines making card counting obsolete?* Retrieved October 14, 2024, from https://www.blackjackapprenticeship.com/continuous-shuffle-machines/

Blackjack Apprenticeship. (2024b). *How to count cards*. Retrieved October 14, 2024, from https://www.blackjackapprenticeship.com/how-to-count-cards/

Blackjack Champ. (2024). *The ultimate online blackjack guide*. Retrieved October 14, 2024, from https://www.blackjackchamp.com

Blackjack Maverick. (n.d.). *Unveiling the origins: A journey through blackjack's history*. Retrieved October 14, 2024, from https://www.blackjackmaverick.com/guides/history-of-blackjack

Bootstrap. (2024). *Bootstrap - powerful, extensible, and feature-packed frontend toolkit*. Retrieved October 14, 2024, from https://getbootstrap.com/

Bostock, M. (2012, January). *Fisher-yates shuffle*. Retrieved August 24, 2024, from https://bost.ocks.org/mike/shuffle/

Bundesamt für Justiz. (2020). *Swiss money gaming act (geldspielgesetz)*. Retrieved October 14, 2024, from https://www.bj.admin.ch/bj/de/home/wirtschaft/geldspiele/faq.html

C., E. (2022). *How online casinos have evolved with technology throughout history*. Retrieved October 14, 2024, from https://www.throwbacks.com/how-online-casinos-have-evolved-with-technology-throughout-history/

Casino.com. (2022, March). *The history of blackjack: From france to the us and beyond*. Retrieved October 14, 2024, from https://www.casino.com/blog/2022/03/10/history/

Community Service Learning CA. (n.d.). *Comparing regulations in online gambling*. Retrieved August 24, 2024, from https://www.communityservicelearning.ca/comparing-regulations-in-online-gambling/

cors. (n.d.). *Cors middleware for node and express*. Retrieved October 15, 2024, from https://www.npmjs.com/package/cors

Create React App. (2024). *Create react app documentation* [Retrieved on October 15, 2024, from https://create-react-app.dev/ and https://create-react-app.dev/docs/getting-started].

CSS Tricks. (2024, August). *Css tricks website*. Retrieved August 24, 2024, from https://css-tricks.com/

Dev.to. (2022, August). *Article on building a chat app with socket.io and react*. Retrieved August 24, 2024, from https://dev.to/novu/building-a-chat-app-with-socketio-and-react-2edj

Diaconu, A. (2024, May). *The websocket api and protocol explained*. Retrieved October 15, 2024, from https://ably.com/topic/websockets#what-are-the-pros-and-cons-of-web-sockets

DigitalOcean. (2024). *DigitalOcean — Cloud Application Platform*. DigitalOcean, LLC. Retrieved October 16, 2024, from https://www.digitalocean.com/

Django Software Foundation. (2024). *Django documentation*. Retrieved October 15, 2024, from https://docs.djangoproject.com/en/5.1/

dotenv. (n.d.). *Load variables from a .env file*. Retrieved October 15, 2024, from https://www.npmjs.com/package/dotenv

Express.js Team. (2024, August). *Express.js official website*. Retrieved August 24, 2024, from https://expressjs.com/

Facebook. (n.d.). *Adding custom environment variables in create react app*. Retrieved October 13, 2024, from https://create-react-app.dev/docs/adding-custom-environment-variables/

Facebook. (2017). *Github issue: Create react app issue #2880 (environment variables)*. Retrieved October 13, 2024, from https://github.com/facebook/create-react-app/issues/2880

Fiedler, I., Ante, L., Meduna, M., et al. (2023). Simulated gambling: An explorative study based on a representative survey. *Journal of Gambling Studies*, *40*, 255–274. https://doi.org/10.1007/s10899-023-10190-6

freeCodeCamp. (n.d.-a). *Javascript tutorial - create a card game*. Retrieved August 24, 2024, from https://www.youtube.com/watch?v=Bj6lC93JMi0

freeCodeCamp. (n.d.-b). *Typescript project references (shared typescript libraries)*. Retrieved August 24, 2024, from https://www.youtube.com/watch?v=S-jj_tifHl4

GeeksforGeeks. (2024, October). *Node vs express*. Retrieved October 15, 2024, from https://www.geeksforgeeks.org/node-js-vs-express-js/

Git. (2024). *Git: Distributed version control system*. Retrieved October 15, 2024, from https://git-scm.com/

GitHub. (2024a). *Github copilot* [AI-powered code completion tool developed by GitHub]. Retrieved October 15, 2024, from https://github.com/features/copilot

GitHub. (2024b). *Github documentation*. Retrieved October 15, 2024, from https://docs.github.com/en

GitHub User "Polivodichka". (2023). *Github repository for blackjack implementation*. Retrieved August 24, 2024, from https://github.com/polivodichka/blackjack

Google. (n.d.). *Cards image resources*. Retrieved August 24, 2024, from https://code.google.com/archive/p/vector-playing-cards/downloads

Guo, Y. (2015, December). *Math.random() in v8*. Retrieved October 14, 2024, from https://v8.dev/blog/math-random

Heroku. (2024). *Heroku — Cloud Application Platform*. Retrieved October 16, 2024, from https://www.heroku.com/

InsightsSuccess. (n.d.). *The evolution of online casinos: From classic to modern variations*. Retrieved August 24, 2024, from https://insightssuccess.com/the-evolution-of-online-casinos-from-classic-to-modern-variations/

Investopedia. (2024, September). *Why does the house always win? a look at casino profitability*. Retrieved October 14, 2024, from https://www.investopedia.com/articles/personal-finance/110415/why-does-house-always-win-look-casino-profitability.asp

JestJS. (n.d.). *Getting started with jest*. Retrieved October 13, 2024, from https://jestjs.io/docs/getting-started

jsonwebtoken. (n.d.). *An implementation of json web tokens*. Retrieved October 15, 2024, from https://www.npmjs.com/package/jsonwebtoken

KamaGames. (n.d.). Blackjackist game on the app store. https://apps.apple.com/us/app/blackjack-21-blackjackist/id959038492

Kilbride-Singh, K. (2023, December). *Long polling vs websockets - which to use in 2024*. Retrieved October 15, 2024, from https://ably.com/blog/websockets-vs-long-polling

Lane, M. N. (2024). *The evolution of blackjack: From classic to online*. Retrieved October 14, 2024, from https://www.missnikkilane.com/the-evolution-of-blackjack-from-classic-to-online/

lucide-react. (n.d.). *Implementation of the lucide icon library for react applications*. Retrieved October 15, 2024, from https://www.npmjs.com/package/lucide-react

MDN Web Docs. (2023). *Math.random() - javascript — mdn*. Retrieved October 14, 2024, from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/random

Mozilla Developers Network (MDN). (n.d.). *Mozilla developer network (mdn) css documentation*. Retrieved August 24, 2024, from https://developer.mozilla.org/en-US/docs/Web/CSS

NCBI Contributors. (n.d.). *Social and economic consequences of online gambling*. Retrieved August 24, 2024, from https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9414406/

Node.js Foundation. (2024). *Node.js official website*. Retrieved August 24, 2024, from https://nodejs.org/

NPM Contributors. (2024, August). *Nodemon npm package page*. Retrieved August 24, 2024, from https://www.npmjs.com/package/nodemon

OnlineCasinos UK. (2024). *History of blackjack*. Retrieved October 13, 2024, from https://www.onlinecasinos.co.uk/game-guides/blackjack/history-of-blackjack.htm

OpenAI. (2024). *Chatgpt* [Large language model chatbot developed by OpenAI]. Retrieved October 15, 2024, from https://chat.openai.com

Oracle. (2024). *Mysql*. Retrieved October 15, 2024, from https://dev.mysql.com/doc/

OWASP. (n.d.-a). *Query parameterization cheat sheet*. Retrieved October 14, 2024, from https://cheatsheetseries.owasp.org/cheatsheets/Query_Parameterization_Cheat_Sheet.html

OWASP. (n.d.-b). *Sql injection*. Retrieved October 16, 2024, from https://owasp.org/www-community/attacks/SQL_Injection

PostgreSQL. (2024). *Postgresql documentation*. Retrieved October 14, 2024, from https://www.postgresql.org/docs/

PrimeDope. (2024). *Decoding online gambling history: The rise of digital betting platforms*. Retrieved August 24, 2024, from https://www.primedope.com/decoding-online-gambling-history-the-rise-of-digital-betting-platforms/

PubNub. (n.d.-a). *What are websockets?* Retrieved October 14, 2024, from https://www.pubnub.com/guides/websockets/

PubNub. (n.d.-b). *What is long polling?* Retrieved October 15, 2024, from https://www.pubnub.com/guides/long-polling/

React Team. (n.d.). *React documentation*. Retrieved August 24, 2024, from https://react.dev/learn

Render. (2024). *Render documentation* [Retrieved on October 15, 2024, from https://render.com/ and https://docs.render.com/].

ResearchGate Contributors. (n.d.). *Exploring the growth, regulations, and social impact of the online gambling industry*. Retrieved August 24, 2024, from https://www.researchgate.net/publication/381739302_Exploring_the_Growth_Regulations_and_Social_Impact_of_the_Online_Gambling_Industry

Sharp, R. (2024, August). *Nodemon github repository*. Retrieved August 24, 2024, from https://github.com/remy/nodemon

Skema. (n.d.). *Article on getting started with sqlite in a typescript project*. Retrieved August 24, 2024, from https://www.skema.cloud/en/blog/sagot-dev-2/get-started-with-sqlite-database-in-a-typescript-project-7

Socket.io Developers. (2024). *Socket.io documentation*. Retrieved October 15, 2024, from https://socket.io/docs/

SQLite. (2024). *Sqlite documentation* [Retrieved on October 14, 2024, from https://www.sqlite.org/docs.html and https://www.sqlite.org/about.html].

Stack Overflow. (2017). *Do i commit the package-lock.json file created by npm 5?* Retrieved October 13, 2024, from https://stackoverflow.com/questions/44206782/do-i-commit-the-package-lock-json-file-created-by-npm-5

Stack Overflow. (2019). *How to create and run a development build of an application using create react app*. Retrieved October 13, 2024, from https://stackoverflow.com/questions/58517422/how-to-create-and-run-a-development-build-of-an-application-using-create-react-a

Stack Overflow Contributors. (n.d.). *Stack overflow discussion on sharing code between typescript projects*. Retrieved August 24, 2024, from https://stackoverflow.com/questions/47729344/how-to-share-code-between-typescript-projects

Tailwind CSS. (2024). *Tailwindcss documentation* [Retrieved on October 15, 2024, from https://tailwindcss.com/ and https://tailwindcss.com/docs/installation].

Terra, J. (2024, July). *What is client-server architecture? everything you should know*. Retrieved October 14, 2024, from https://www.simplilearn.com/what-is-client-server-architecture-article

TeX Stack Exchange Contributors. (n.d.). *Tex stack exchange discussion on language options supported in listings*. Retrieved August 24, 2024, from https://tex.stackexchange.com/questions/89574/language-option-supported-in-listings

ts-node. (n.d.). *Typescript execution and repl for node.js*. Retrieved October 15, 2024, from https://www.npmjs.com/package/ts-node

TypeScript Team. (n.d.-a). *Typescript documentation on migrating from javascript*. Retrieved August 24, 2024, from https://www.typescriptlang.org/docs/handbook/migrating-from-javascript.html

TypeScript Team. (n.d.-b). *Typescript official documentation*. Retrieved August 24, 2024, from https://www.typescriptlang.org/docs/

Vercel. (2024). *Next.js by vercel - the react framework*. Retrieved October 15, 2024, from https://nextjs.org/

Vue.js Team. (n.d.). *Vue.js - web application framework*. Retrieved August 24, 2024, from https://vuejs.org/

W3Schools. (n.d.). *W3schools css tutorial*. Retrieved August 24, 2024, from https://www.w3schools.com/css/

Wikipedia. (n.d.-a). *Bcrypt*. Retrieved October 13, 2024, from https://en.wikipedia.org/wiki/Bcrypt

Wikipedia. (n.d.-b). *Fisher–yates shuffle*. Retrieved October 14, 2024, from https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle

Wikipedia. (n.d.-c). *Sql*. Retrieved October 14, 2024, from https://en.wikipedia.org/wiki/SQL

WilliamHill. (n.d.). *What is blackjack insurance and how does it work?* Retrieved October 16, 2024, from https://news.williamhill.com/casino-guides/blackjack-insurance/

YouTube User "Chris Courses". (2023). *Online multiplayer javascript game tutorial - full course*. Retrieved October 13, 2024, from https://www.youtube.com/watch?v=HXquxWtE5vA

YouTube User "Dave Gray". (n.d.). *Socket.io introduction - how to build a chat app*. Retrieved August 24, 2024, from https://www.youtube.com/watch?v=SGQM7PU9hzI

YouTube User "RoadsideCoder". (2023). *Deploying mern stack app to render*. Retrieved October 13, 2024, from https://www.youtube.com/watch?v=aQ_2DMXSNwY

A Appendices

The source code for Quick21 is available at https://github.com/mdbrnd/Quick21. In addition to the cited sources, the bibliography includes materials that were consulted for background information and learning but are not directly referenced in the text.

A.1 Arbeitsjournal

The detailed Arbeitsjournal, which includes a record of the challenges faced and time spent on the project, is available at https://github.com/mdbrnd/Maturarbeit.