Arbeitsjournal

During the development of this project, I faced several technical challenges and decisions. This journal serves as a record of the tasks I worked on, the difficulties I faced, and the hypotheses or solutions I explored. It allows the reader to trace the progression of the project and gain insight into the reasoning behind the decisions I made. A table showcasing the development timeline can be found at the end of this document. For a more in-depth view of the development process, including specific changes, you can refer to the commit history in the GitHub repository: https://github.com/mdbrnd/Quick21. The accompanying paper is also of great help, available at https://github.com/mdbrnd/Maturarbeit.

On November 1st, 2023, I met with my supervisor to discuss potential topics for my Matura paper. The discussion revolved around two main options: developing a blackjack game or creating a programming language. After weighing the pros and cons, I chose the blackjack game because it offered more practical challenges and allowed me to apply a broader range of skills. This meeting also helped clarify the evaluation criteria for the project, with 60% of the focus on coding and 40% on the accompanying paper.

A couple of months later, on January 5th, 2024, I set up the initial framework for the frontend. After evaluating popular frontend frameworks like Angular and Vue, I chose to use React mainly because of its popularity, strong community support, and the component-based architecture that simplifies the management of reusable elements. While Angular and Vue also use component-based architectures, I prefer React's approach. Learning any framework is not necessarily easier than not using a framework, but React’s flexibility and preferred component-based structure fit the needs of the project more than vanilla HTML, CSS and JavaScript. I chose Create React App (CRA) to initialize the frontend project because it simplified the project setup by offering features like hot reload and built-in webpack configuration. Although my experience in web development was limited, I had experimented with some test apps using Create React App a few months earlier, which made it a familiar and accessible choice. This familiarity, combined with its out of the box features, allowed me to quickly move forward. I also uploaded the project to GitHub using Git, a tool I was already familiar with, which helped me manage version control efficiently throughout the development process.

With the framework in place, I spent the next few hours that same day developing the basic client skeleton using React and some basic CSS. These early stages were mostly experimental, so not much substantial progress was made. Initially, I built the app using JavaScript on both the client and server sides to get a feel for the development process. However, once I moved on to the actual application, I switched to TypeScript, as originally planned, for its strong type-checking and scalability. My prior experience with statically typed programming languages such as C# or Dart made TypeScript even more appealing.

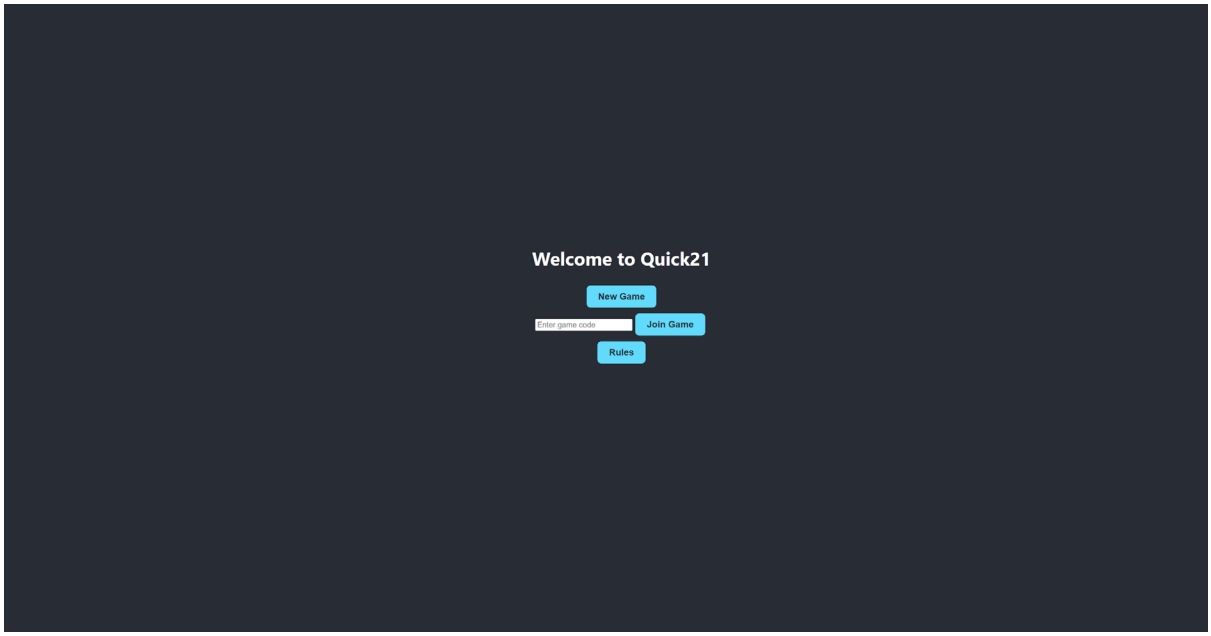

Figure 1: The very first version of the Quick21 lobby screen

During this time, I was also thinking about the structure of my paper and began creating an implementation timeline to guide my progress. Over the next few months, I developed the core game architecture, with significant progress made by April, when I had implemented key features for managing game state and room interactions.

To better understand and communicate the game's flow and structure, I created detailed flowcharts to visualize how data would move through the system and how users would interact with it. These visualizations not only clarified the system for me but also helped anticipate potential challenges in the user interaction flow, ensuring a smoother development process.

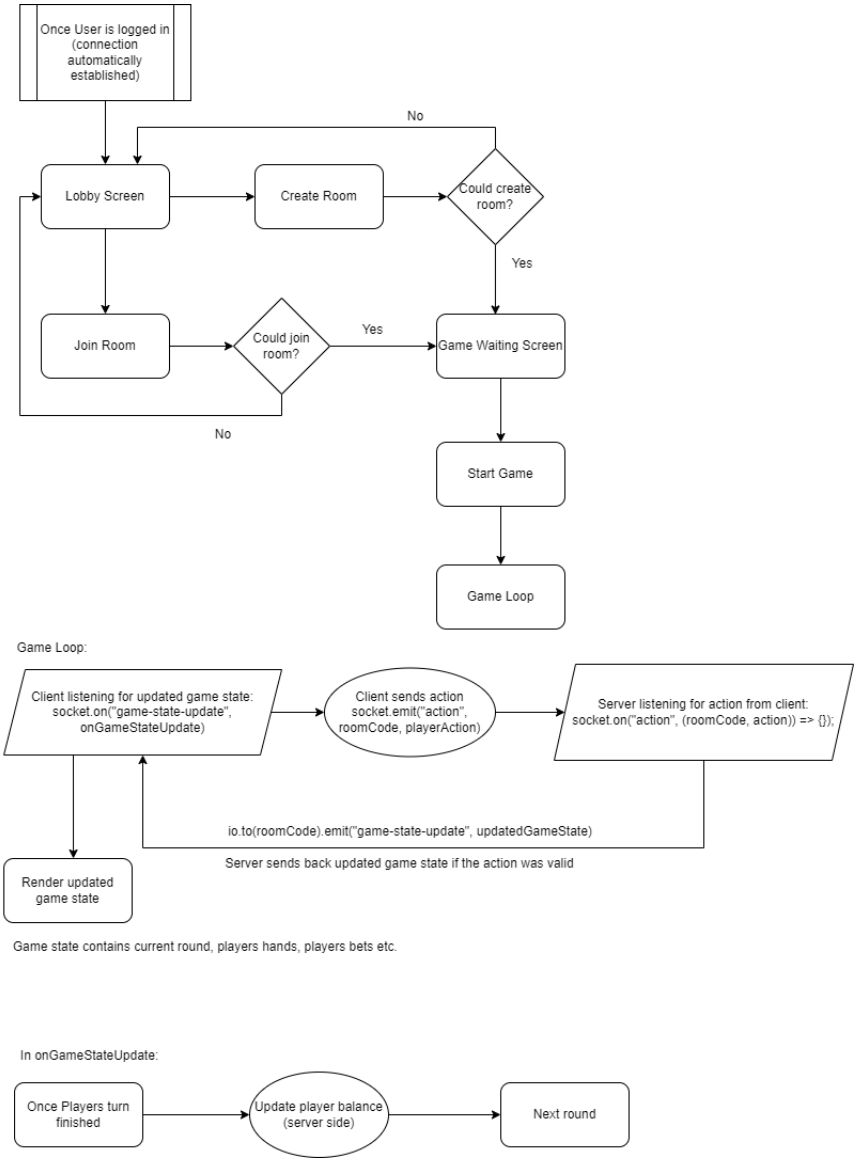

Figure 2: The first draft of the game flow diagram

A few days later, on April 20th, 2024, I returned to the concept stage. I had built a basic UI skeleton (Figure 1) using plain CSS. While this initial layout worked, what truly helped was sketching out the full UI, detailing every screen and interaction.

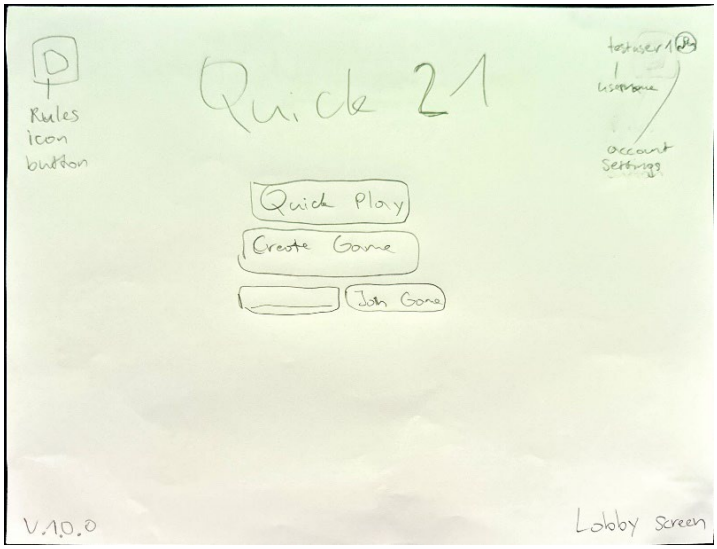

Figure 3: Initial sketch of the lobby screen

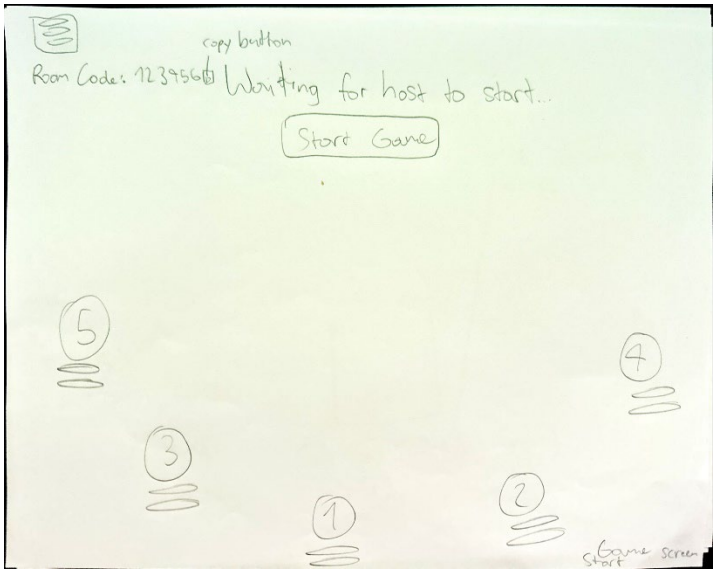

Figure 4: Initial sketch of the game screen

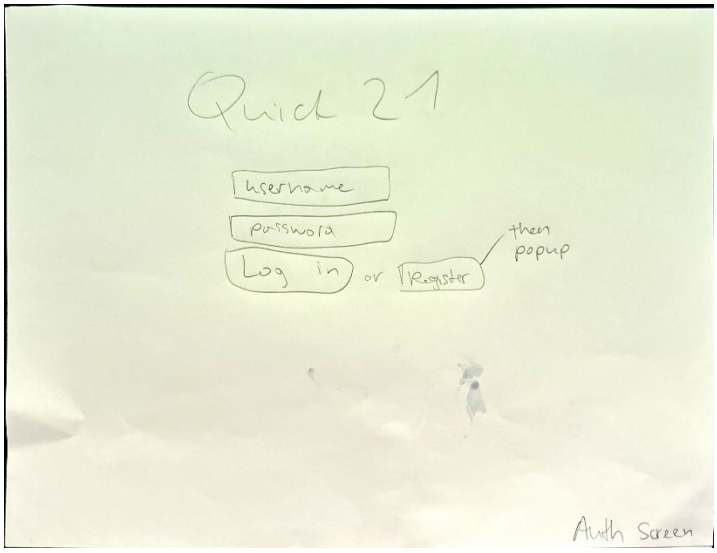

Figure 5: Initial sketch of the authentication screen

These sketches, though quite basic, were extremely helpful throughout the development process. Even though I didn't follow them exactly, they provided a solid foundation and direction for the design.

As expected, development came with its share of bugs. Near the end of April, I focused on fixing several client-side issues, particularly with room management. One notable bug was that the room owner wasn't joining the correct room when creating a game. I also updated the client models to match the backend and began laying the groundwork for authentication features.

The next major phase of development took place during the spring holidays, between April 25th and May 14th, 2024, where I spent around 12 hours fixing various bugs and adding core functionalities. A major issue I faced was ensuring the game state transitioned correctly, which led to debugging data serialization problems in the backend. I couldn't figure out why the game state wasn't transferring correctly, especially as it had worked before to transfer info such as whether the game had started, until I found an important detail in an older version of the Socket.IO documentation: "Map and Set are not serializable and must be manually serialized" (Socket.IO, 2024, https://socket.io/docs/v3/emitting-events/#:\~:text=Note:%20Map%20and%20Set%20are%20not%20serializable%20and%20must%20be%20manually%20serialized). As I came to know, Maps and Sets are not serializable by default in JavaScript.

This meant I had to manually serialize the data. Fortunately, the solution was straightforward—I cast the existing Maps (like *bets: Map<Player, number>*) to arrays using *Array.from*. ChatGPT helped me find this approach, and this simple fix resolved the serialization issue, allowing the game state to transfer correctly.

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

const serializeMap = (map: Map<any, any>) => {
  return Array.from(map.entries());
};

const deserializeMap = (array: any[]) => {
  return new Map(array);
};
```

Figure 6: The code used to serialize and deserialize maps

By June, most of the game loop was complete, and I shifted my focus to implementing the visual elements of the game. I started by adding card images to the UI to visually represent the current game state. Over the next few weeks, I continuously worked on improving the UI. This was also when I realized that my original plan to finish everything by July was a bit too ambitious. In coordination with my supervisor, I decided to push the due date for the first version to after the summer holidays and revised the timeline accordingly.

One particularly challenging aspect was CSS card stacking and animation. My goal was to display each player's hand as a stack of cards, while smoothly animating new cards as they were added. The first challenge I faced was aligning the cards with the correct spacing to create a visually appealing "stacked" effect. Ensuring each card had just the right amount of overlap while maintaining readability was trickier than expected.

Aligning the cards so that each one had just the right amount of space from the previous card proved to be more challenging than I initially expected. I struggled to position them correctly within the container to achieve a slight overlap, creating a fanned-out effect. My first approach involved applying a negative margin based on each card's index, but this led to issues. Negative margins, which aren't typically used this way, caused only the first two cards to appear stacked, while any additional cards were displayed incorrectly:

Figure 7: An attempt at stacking the cards

After some trial and error, I eventually settled on dynamically calculating the left style for each card using its index in the array, multiplying the card index by 35 pixels. This gave me the spacing I needed, though it required several adjustments to get the balance right between spacing and visibility.

This experience, along with the growing number of custom styles, made me consider switching to Tailwind CSS. Up until this point, I had been using vanilla CSS for all styling. However, as the project grew, maintaining and customizing styles became more time-consuming and complex. Switching to Tailwind CSS allowed me to apply utility classes directly to the components, making the styling process significantly simpler and in my eyes also more efficient. I also took the opportunity to completely redesign the UI and update the color scheme, resulting in a cleaner and more cohesive look, as shown in the screenshots provided.

Figure 8: The old version of the Quick21 UI

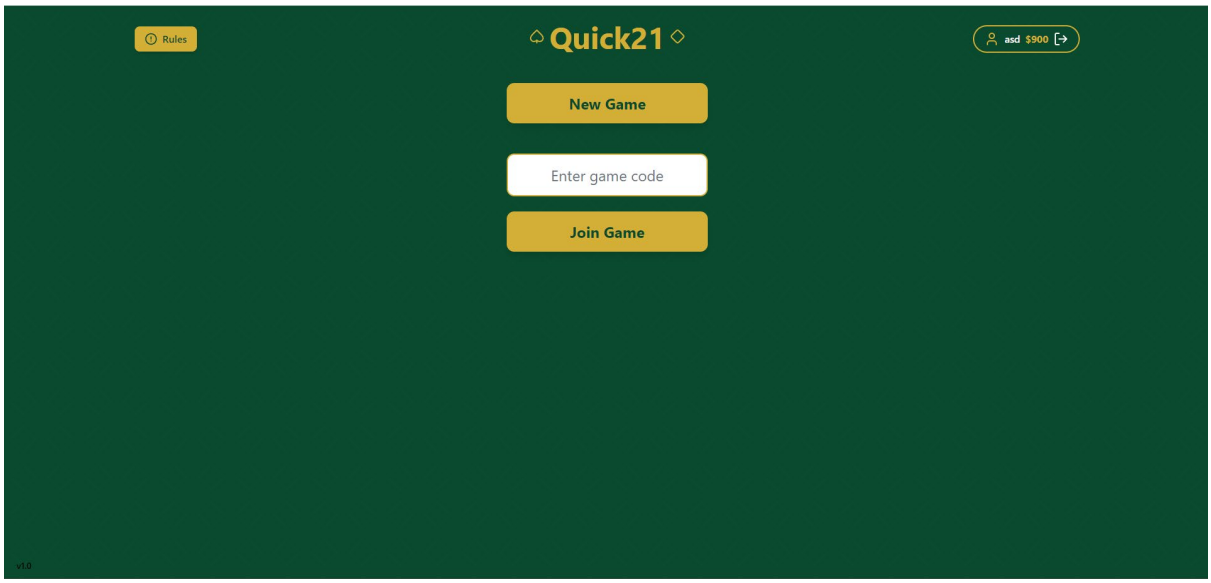

Figure 9: The updated version of the Quick21 UI

After implementing Tailwind CSS, I moved on to tackle the other card UI challenge: animating the cards as they entered. I wanted the most recent card to appear with a smooth entrance, but I had limited experience with CSS animations at the time. It took some research (mainly https://developer.mozilla.org/en-US/docs/Web/CSS and https://www.w3schools.com/css/) to figure out how to conditionally apply an animation class to the last card in the array, while ensuring that the animation wouldn't be triggered again for cards that had already been displayed.

```css
/* client/src/index.css */

@keyframes cardEnter {
  from {
    opacity: 0;
    transform: translateY(-50px) rotate(-10deg);
  }
  to {
    opacity: 1;
    transform: translateY(0) rotate(0);
  }
}

.animate-card-enter {
  animation: cardEnter 0.5s ease-out;
}
```

Figure 10: The CSS used to animate a card entering the player's hand

I eventually solved this by tracking how many cards had been animated for each player using an *animatedCards* state. This allowed me to apply the animation only to the newly added card, preventing the animation from re-triggering on the already displayed cards. Once the animation was complete, I updated the state accordingly, ensuring a smooth flow. The code for this can be found at *client\src\pages\game\GameControls.tsx*.

The month of August was largely dedicated to implementing the authentication system. Although the blueprint for generating JWT tokens and storing user information in the database was already in place, I needed to connect everything to the frontend. I spent around seven hours refining the authentication logic and further developing my paper. Once the authentication system was integrated, I also added Jest (a testing library) tests to verify the core game logic on the backend and continued improving the UI to enhance the user experience.

By September, the next major milestone was hosting Quick21 so it could be accessed online by others. Hosting the game presented a challenge, as the client and server were two separate projects managed in different folders. Initially, I struggled to find a way to deploy them together, since coordinating two independent deployments seemed overly complicated for such a small-scale project.

After some research (mainly watching YouTube tutorials, the notable one being https://www.youtube.com/watch?v=aQ_2DMXSNwY, on how to deploy similar applications) and asking ChatGPT, I decided to serve a build of the client directly from the server. I did this by adding a custom build script that combined the two projects into a single deployment:

```
// ./package.json

"prebuild": "npm install && cd client && npm install && cd .. && cd server && npm install",
"build": "cd client && npm run build && cd .. && cd server && npm run build"
```

Figure 11: The scripts used to build the project

The build script handled the process of navigating into the client directory, running the build command for the React project, and then moving back to the server directory to prepare the backend build (to transpile the TypeScript to JavaScript).

This approach allowed me to serve the client build from the server:

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

// __dirname = server/src/server.ts for development since its using nodemon (npm run dev)
var distDir = path.join(__dirname, "../../client/build");
if (process.env.NODE_ENV === "production") {
  // __dirname = server/src/dist/server.js
  distDir = path.join(__dirname, "../../../client/build");
}
app.use(express.static(distDir));

app.get("*", (req, res) => {
  res.sendFile(path.join(distDir, "index.html"));
});
```

Figure 12: The code used to serve the client build from the server

This solution simplified the hosting and development process by only needing to run the commands in a single root directory. It was a significant learning experience that helped me understand how to manage multiple projects under one server.

Now that I had this process working, I needed to decide which hosting provider to use—or if I should even use a hosting provider at all. I evaluated several options, including DigitalOcean, Heroku, AWS, and Render, to find the best fit for the project. Ultimately, I chose Render because it offered the ideal balance between ease of use and functionality. Render, like most modern hosting platforms, provides automatic deployment from GitHub. This means the latest code is deployed to the live server whenever changes are pushed to the *main* branch, making it easy to commit updates to a *staging* branch and merge them when I was ready to publish. Render greatly simplified the process of setting the hosting up. Although DigitalOcean and AWS offered more control and flexibility, they required significantly more manual setup, which was unnecessary for a project of this scale. The same applied to the idea of hosting the project myself on my own devices. Heroku was also a strong contender, but after watching a tutorial on deploying an application with a similar structure to mine (https://www.youtube.com/watch?v=aQ_2DMXSNwY), I decided to go with Render as it aligned with the deployment process shown in the video and seemed like the most uncomplicated option.

Setting it up was straightforward. I simply connected my GitHub repository, provided the build and start commands, and set up the necessary environment variables. By September 24, 2024, the website was live at https://quick21.onrender.com. Initially, I used Render's free plan, but since it shut down the server during periods of inactivity, there was a delay of about a minute each time the website was accessed, as the server had to turn on again.

To avoid this, I upgraded to the starter plan for $7 per month, which ensured continuous uptime. This plan also gave me SSH access and persistent disks, meaning user data in my database would be stored consistently, even if the server restarts.

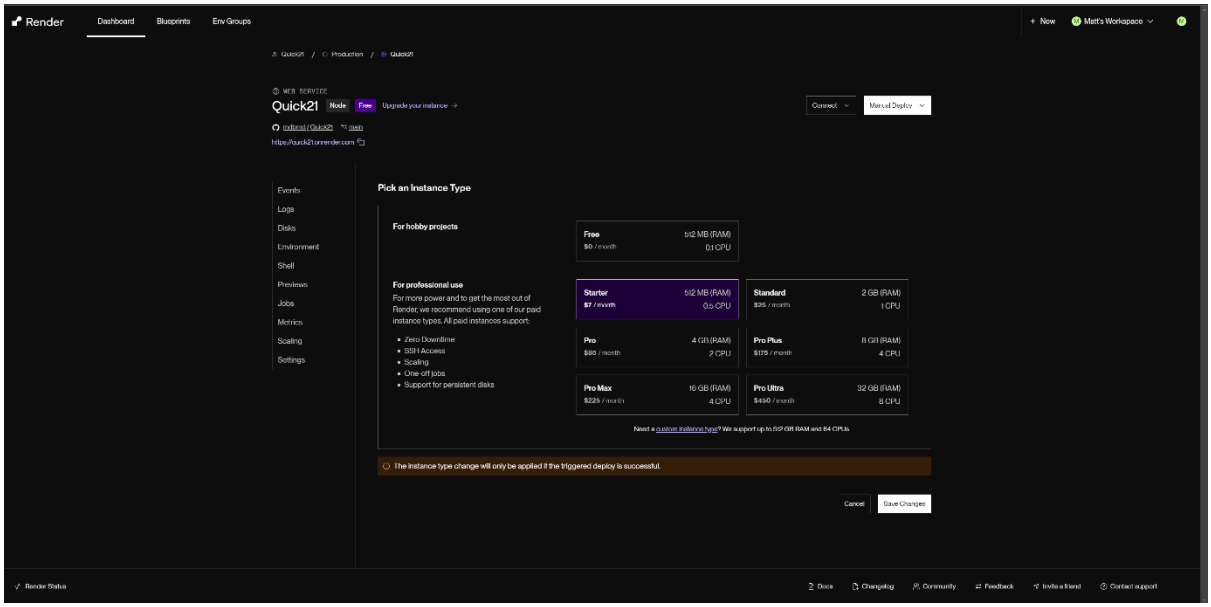

Figure 13: A screenshot of the instance type dashboard in Render

Once the project was hosted, I conducted user tests with my dad and friends, which proved to be incredibly valuable. These tests helped me uncover several bugs, such as issues with authentication token storage not functioning correctly. I also gathered useful feedback about the UI, which led to improvements like adding an indicator for the dealer's hand value and making the game results more clearly visible.

Between October 6th and 20th, 2024, I dedicated many hours to addressing my supervisor's feedback on the paper and making final touches to both the project and the document. I incorporated minor UI improvements, such as balance displays, pulsating indicators for turns and betting phases, and enhanced the betting interface to create a smoother user experience.

Other Challenges Along the Way

I initially chose Create React App (CRA) to create the project because it simplified the project setup and I was somewhat familiar with it, but I later encountered a limitation with managing environment variables. I couldn't manually change the *NODE_ENV* variable, as CRA automatically set it to *production* when running *npm build*. I needed to determine the

development environment in which the frontend was running, as the base URL for requests had to be adjusted accordingly (using localhost during development and the online URL in production). This forced me to create a workaround using two *.env* files—one in the root directory and another in the client folder. The *.env* file in the root couldn't be accessed with *process.env* because the build command was run from the client folder, not the project root where the root *.env* file was located.

This wasn't an issue for the server, as the command to run it was executed from the root directory, allowing the server to handle environment variables directly with *process.env* and read from the root *.env* file. For more info, see https://create-react-app.dev/docs/adding-custom-environment-variables/. In hindsight, I might have been able to backtrack and find a way to read the *.env* file at build time. However, by the time I encountered these limitations, I had already built much of the app and creating a workaround with the second *.env* file was a lot more straightforward than figuring out another way.

Another issue I encountered was sharing models between the client and server. Initially, I wanted to create a shared *models* folder so both the frontend and backend could use the same TypeScript interfaces. However, setting up shared libraries between the two was more complex than I anticipated and required a lot of overhead. After spending time trying to get it working, I decided to simplify things by just copying the models between the client and server projects. While not the most elegant solution, it worked well enough for this project and allowed me to keep moving forward without getting bogged down by complex module setups.

Finally, while testing on another device, I encountered an "address in use" error, which was specific to Windows. This occurred because Windows handles process termination differently compared to UNIX-based systems. On UNIX, signals like SIGINT or SIGTERM are sent to child processes when the parent process (such as the terminal) is closed. However, on Windows, these signals are not always propagated properly. If you close the terminal using the X button, the Node.js process remains running in the background, which causes the "address in use" error when attempting to restart the server.

To avoid this, it's important to terminate the program using Ctrl+C or run it through an integrated terminal in VSCode, which handles these signals more effectively. This issue does not affect macOS systems. For more details, refer to https://learn.microsoft.com/en-us/windows/win32/procthread/terminating-a-process and https://unix.stackexchange.com/questions/149741/why-is-sigint-not-propagated-to-child-process-when-sent-to-its-parent-process. It would have been possible to add a command that automatically kills all processes using the default server port (4000). However, this approach didn't seem ideal, as it could unintentionally terminate other processes running on that port, potentially causing issues for users who rely on that port for unrelated tasks.

Known Bugs

- Occasionally, WebSocket requests fail, causing the player list to not populate when a player joins a room. I have not yet been able to track down the source of the bug.
- In *SocketContext.tsx*, the *isAuthenticated* state sometimes fails to initialize correctly, leading to potential issues with authentication handling. It works correctly on the landing page, so a workaround is to redirect to the landing page if the socket is unauthorized or uninitialized and then redirect back to the lobby.
- On mobile, certain UI elements overlap, affecting the display and usability of the interface.
- Technically, the card area UI could go out of its boundaries a bit if a player has a hand with 5+ cards (very unlikely).

Development Timeline

Table 1: The development timeline and key activities for the project

Date	Hours	Category (What)	Content, Steps, Activities
01.11.23	1	Meeting with Supervisor	Discussed topic selection, narrowed down themes, and reviewed evaluation criteria. Main choice was between blackjack game and programming language; opted for blackjack due to its practical challenges and broader skill application.
05.01.24	1	Project Setup	Chose the main framework for the project. Decided to use React after comparing alternatives like Angular and Vue for its simplicity and familiarity. Used CRA to set it up due to familiarity and not having to configure things.
05.01.24	2	Implementation	Started developing the frontend by implementing the basic client skeleton using React and basic CSS.
01.02.24	1	Meeting with Supervisor	Finalized evaluation criteria (60% code, 40% paper), discussed formal requirements and project timeline. Code will be evaluated on quality (consistent style, functions, use of React, use of VCS: Git).
25.02.24 - 28.02.24	1.5	Rough Concept	Outlined the high-level architecture of the project and structure of the paper.
07.03.24	4	Detailed Concept	Defined detailed project expectations, created chapter titles for the paper, set up a detailed timeline.
23.03.24	4	Frontend and Backend Integration	Started implementing backend, connected frontend and backend and implemented a basic draft for room management in the backend

24.03.24	2.5	Backend Models	Created data models for managing the game state, players, and rooms in the backend.
25.03.24	3	Backend Development	Progressed on backend functionalities, added events for room and game state management.
11.04.24	1	Flow Chart Creation	Created detailed flow charts the game loop and flow to visualize data flow and user interactions.
20.04.24	1	Concept Refinement	Created sketches for the UI, refined flow charts and diagrams to improve understanding of the system flow.
24.04.24	1.5	Bug Fixes & Client-Backend Update	Fixed client-side bugs (for example, room owner was not joining the socket room when creating a game), updated client models to match backend, added groundwork for authentication and database in the backend.
25.04.24 - 14.05.24	12	Implementation	Focused on bug fixing (for example, the game state not transferring correctly due to some types not being able to be automatically serialized), added betting functionality, managed game state transitions, and developed the game loop logic.
01.06.24	1	UI Implementation	Added card images to the UI to visually represent game state.
07.06.24	3	UI Improvements	Improved the overall UI layout, fixed CSS issues, and ensured players could only perform one action per turn.
22.06.24 - 24.06.24	4	UI and Logic Fixes	Further UI improvements and fixed some logic issues related to game state transitions and turn-based gameplay.
25.06.24	1	Meeting with Supervisor	Demonstrated progress, discussed issues with card stacking, and explored possible animation solutions.
25.06.24	1	Documentation	Created readme files with installation instructions for both client and server components.
01.07.24 - 01.08.24	14	UI and Paper Writing	Migrated from vanilla CSS to Tailwind CSS while also redesigning the UI, implemented card stacking animation, wrote and documented up to chapter 4 (at the time, chapter 4 was the implementation chapter) of the paper.
05.08.24	1	Authentication Fix	Fixed authentication issues related to token management in the backend.
06.08.24 - 14.08.24	7	Authentication and Paper Refinement	Integrated the existing blueprint for authentication logic into the app, refined the paper. This time was mostly spent setting up the authentication and token storage on the client side while also fetching user info.

22.08.24 - 30.08.24	9	Paper Review and Testing	Reviewed paper chapters, fixed UI bugs, improved user experience. Added jest test for the core game logic in the backend
09.09.24 - 15.09.24	3.5	UI and Hosting Research	Implemented further UI improvements, researched various hosting options (DigitalOcean, Render, AWS) and decided to use Render.
19.09.24	1.5	First Draft Review	Discussed and reviewed the first draft of the paper with the supervisor.
24.09.24 - 27.09.24	7.5	Hosting and Deployment	Unified client and server under a single npm project, deployed on Render, added features like automatic token handling and admin panel. Conducted user testing with my dad and friends for feedback.
01.10.24 - 20.10.24	34	Feedback Implementation and Final touches	Addressed supervisor feedback for first draft, completed minor remaining UI improvements (display balance on the main game screen, pulsating indicators for turns and betting phases, and an enhanced betting interface), reread the entire paper, and consolidated Arbeitsjournal into a final document.