Philipps-Universität Marburg

Fachbereich 12 - Mathematik und Informatik

Bachelorarbeit

Webbasierte Multiplayer Schach-App

von
Jasper Paul Fülle

Mai 2023

Betreuer:
Prof. Dr. Thorsten Thormählen

Arbeitsgruppe Grafik und Multimedia Programmierung

Erklärung

Ich, Jasper Paul Fülle (Wirtschaftsinformatikstudent an der Philipps-Universität Marburg, Matrikelnummer 3367654), versichere an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die hier vorliegende Bachelorarbeit wurde weder in ihrer jetzigen noch in einer ähnlichen Form einer Prüfungskommission vorgelegt.

Marburg, 24. Mai 2023

Jasper Fülle

Kurzzusammenfassung

Viele der in der Computergrafik verwendeten 3D-Modelle werden mit Hilfe von Dreiecksnetzen repräsentiert. ... (max. 1 Seite)

Abstract

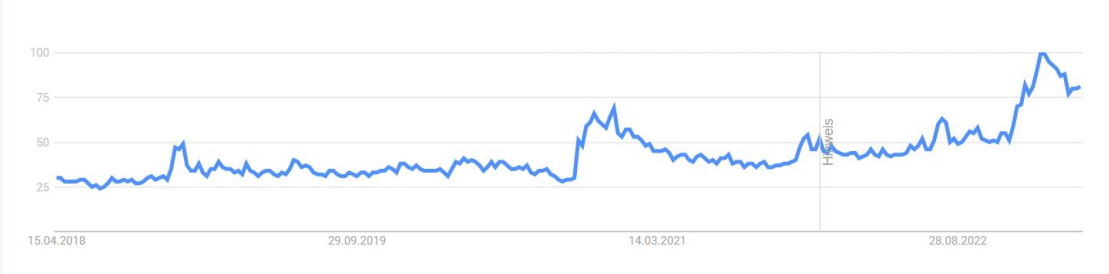
text text text text text text text text text text text text text text text text text text text text text text (exakte englische Übersetzung der deutschen Kurzzusammenfassung)

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Schach ist ein traditionsreiches und abwechslungsreiches Brettspiel, dessen Ursprung nicht genau bestimmt werden kann. Es wird vermutet, dass das erste schachähnliche Spiel *Tschaturanga* seinen Ursprung in Nordindien um 600 n. Chr. hatte [**?**]. Im Laufe der Jahrhunderte hat Schach eine bedeutende Rolle in der Kultur und Geschichte eingenommen. So wurde beispielsweise die Schach-WM 1972 eine Art Machtkampf im kalten Krieg zwischen der UdSSR, die den damaligen Schach dominierte, und der USA[1]. Schach bleibt bis heute ein beliebtes Spiel, das 2020 durch die Netflix Serie *Damengambit* und 2022 durch den Betrugsvorwurf von Magnus Carlsen an seinen 19-jährigen Gegner Hans Niemann[2] eine größere Aufmerksamkeit erhielt (siehe Abbildung 1.1).



Quelle: `https://trends.google.de/`

Abbildung 1.1: Relatives Suchinteresse des Wortes *Chess* auf Google in den letzten 5 Jahren.

Darüber hinaus hat Schach im digitalen Zeitalter eine neue Popularität erreicht. Online-Schachplattformen wie `chess.com` verzeichnen Milliarden von Live-Partien[3], während Schach Live-Streams auf Plattformen wie `twitch.com` Millionen von Followern anziehen[4].
Die Entwicklung einer webbasierten Multiplayer-Schach-App bietet eine einzigartige Gelegenheit, ein traditionsreiches und beliebtes Spiel im digitalen Zeitalter weiterzuentwi-

[1] Quelle: `https://www.geo.de/magazine/geo-epoche/19054-rtkl-schachweltmeisterschaft-wie-ein-schachspiel-zum-wettstreit-der` am 22. April 2023
[2] Quelle: `https://www.sportschau.de/schach/magnus-carlsen-hans-niemann-ermittlungen-100.html` am 22. April 2023
[3] Quelle: `https://www.chess.com/forum/view/general/weve-reached-3000000000-live-chess-games` am 22. April 2023
[4] Quelle: `https://www.twitch.tv/chess` am 27. April 2023

ckeln. Meine Motivation für diese Arbeit besteht darin, eine App zu entwickeln, die die Grundlagen einer Schach-App enthält und gleichzeitig eine solide Basis für zukünftige Erweiterungen und Verbesserungen bietet. Insbesondere plane ich in Zukunft, innovative Funktionen zu integrieren, die bislang in den gängigen Schach-Apps nicht vorhanden sind, wie z.B. die Möglichkeit, unterschiedliche Schachfiguren und -bretter als Belohnungen freizuschalten oder mit Freunden eine Gruppe zu gründen, welche in einer Liga auf- und absteigen kann. Durch die Entwicklung einer Schach-App mit neuen Funktionalitäten kann ich dazu beitragen, die Popularität von Schach zu steigern und vor allem das Spiel einem breiteren Publikum zugänglich zu machen.

1.2 Zielsetzung

Diese Bachelorarbeit hat das Ziel eine Schach-App zu entwerfen und zu implementieren, die eine intuitive User Experience und ein ansprechendes User Interface mit vielen nützlichen Funktionen beinhaltet.
User Experience (kurz UX) bezieht sich darauf wie ein Nutzer sich auf einer Anwendung bewegt und wie einfach und angenehm es für den Nutzer ist, die Funktionen der Anwendung zu verwenden.
Unter User Interface (kurz UI) versteht man die visuelle und interaktive Gestaltung von Benutzeroberflächen. Es umfasst die Gestaltung von Buttons, Formularen und anderen visuellen Komponenten, sowie das Feedback dieser Komponenten, wie zum Beispiel die Rückmeldung einer fehlgeschlagenen Anmeldung. Zusammengefasst beschäftigt sich UX damit, wie man eine Anwendung verwendet und UI damit, wie die Benutzeroberfläche der Anwendung aussieht.[**?**]
Funktionen der Schach-App sind unter anderem das Registrieren und Anmelden, das Versenden, Annehmen und Ablehnen von Freundschaftsanfragen, das Zuschauen bei laufenden Spielen, das Herausfordern von Freunden zu Schachspielen und natürlich das Spielen von Schachpartien mit einem Chat und verschiedenen Einstellungsmöglichkeiten der Schachuhren selbst. Dabei wird besonderer Wert auf die Verwendung moderner Web-Technologien wie React, Node.js, Socket.IO, Redis und PostgeSQL gelegt, um eine optimale Benutzererfahrung und Skalierbarkeit zu gewährleisten. Darüber hinaus soll die Arbeit einen Überblick über die technischen Herausforderungen und Lösungen im Zusammenhang mit der Implementierung einer solchen Schach-App bieten.

1.3 Aufbau der Arbeit

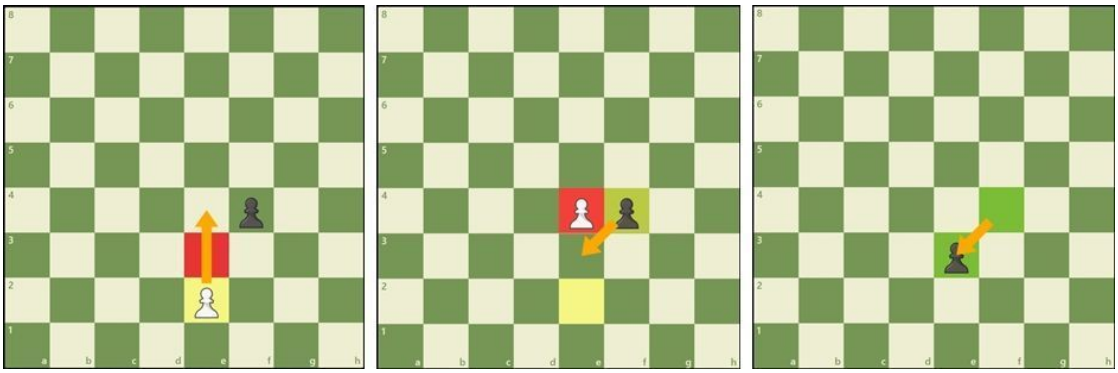
Diese Bachelorarbeit gliedert sich in sechs Hauptkapitel, die jeweils unterschiedliche Aspekte der Entwicklung und Implementierung der Schach-App behandeln.
Im ersten Kapitel, der *Einleitung*, werden die Motivation für die Entwicklung der Schach-App, die Zielsetzung der Arbeit und der Aufbau der Arbeit selbst vorgestellt.
Das zweite Kapitel, *Theoretische Grundlagen*, erläutert die Grundlagen von Schach als Spiel sowie die verwendeten Web-Technologien wie Node.js, Express, Socket.io, React und PostgreSQL, die für das Verständnis der nachfolgenden Kapitel wichtig sind.

Im dritten Kapitel, *Systemarchitektur*, wird die Gesamtarchitektur der Schach-App beschrieben, einschließlich der Unterteilung in Frontend und Backend, der Datenbankstruktur und der Kommunikation zwischen den verschiedenen Komponenten.
Das vierte Kapitel, *Implementierung*, geht auf die praktische Umsetzung der Schach-App ein, indem es die Entwicklungsprozesse für das Frontend und das Backend sowie die Integration der Datenbanken erläutert.
Das fünfte Kapitel, *Tests und Evaluation*, behandelt die verschiedenen Tests, die durchgeführt wurden, um die Funktionalität, Usability, Performance und Sicherheit der Schach-App zu bewerten.
Im abschließenden sechsten Kapitel, *Fazit und Ausblick*, werden die Ergebnisse der Arbeit zusammengefasst, eventuelle Limitationen diskutiert und mögliche Erweiterungen und Weiterentwicklungen für die Schach-App vorgeschlagen.
Die Arbeit endet mit dem *Anhang*, der zusätzliche Grafiken und die Liste der verwendeten Literatur enthält.

2 Theoretische Grundlagen

2.1 Schach

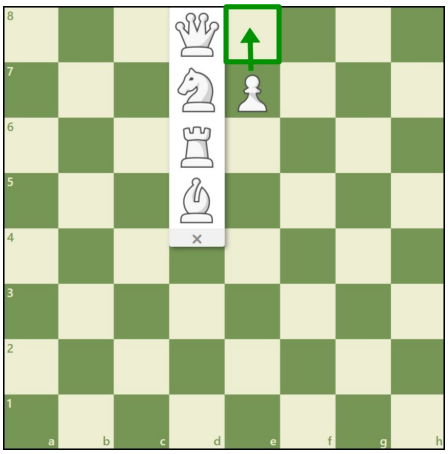
Schach ist ein strategisches Brettspiel für zwei Spieler, welches auf einem quadratischen Spielfeld mit 64 Feldern gespielt wird. Jeder Spieler beginnt mit 16 Figuren und das Ziel des Spiels ist es, den König des Gegners schachmatt zu setzen, indem man ihn bedroht, ohne dass der Gegner den Angriff verhindern kann.
Wie Figuren sich bewegen und andere Figuren schlagen erkläre ich nicht explizit, lediglich zwei Sonderregeln des Schachs und Schachuhren werde ich genauer erklären, da diese bei der Umsetzung des Spiels gesondert gehandhabt werden müssen.
Die erste ist die so genannte *en passant*-Regel. Dabei ist es einem Bauern möglich einen gegnerischen Bauer diagonal zu schlagen, falls dieser zwei Felder gezogen ist und nun auf der gleichen Höhe wie der eigene Bauer steht (siehe Abbildung 2.1).
Die zweite Zusatzregel ist die *Bauernumwandlung*. Sie besagt, dass falls ein Bauer die gegnerische Grundreihe erreicht, dieser Bauer in eine Dame, einen Springer, einen Turm oder einen Läufer umgewandelt werden kann (siehe Abbildung 2.2).



Quelle: `https://www.chess.com/de/schachregeln`

Abbildung 2.1: Die Zusatzregel *en passant*

Die Notation von Schachuhren besteht aus zwei Zahlen, die mit einem Plus getrennt werden, wie zum Beispiel „10 + 5“. Hier steht die erste Zahl, in diesem Fall 10, für die Gesamtzeit, die jedem Spieler zur Verfügung steht, also 10 Minuten. Die zweite Zahl, hier 5, wird als Inkrement bezeichnet. Dies bedeutet, dass nach jedem Zug eines Spielers diesem Spieler zusätzlich 5 Sekunden hinzugefügt werden. Dadurch haben Partien mit Inkrement mehr Zeit, je mehr Züge gespielt werden.
Bei Online-Schachpartien ist es zusätzlich üblich, eine bestimmte Startzeit für den ersten



Quelle: `https://www.chess.com/de/schachregeln`

Abbildung 2.2: Die Zusatzregel *Bauernumwandlung*

Zug jedes Spielers verstreichen zu lassen. Dies gewährleistet, dass die Uhren erst zu laufen beginnen, wenn beide Spieler bereit sind und mitbekommen haben, dass das Spiel gestartet wurde.

2.2 Web-Technologien

2.2.1 Node.js und Express

Node.js und seine Vorteile

Node.js ist eine JavaScript-Laufzeitumgebung, welche erstmals 2009 angekündigt wurde[1] und speziell für die Entwicklung von skalierbaren Netzwerkanwendungen entworfen wurde [**?**]. Skalierbarkeit bedeutet, dass mit steigender Benutzeranzahl der Ressourcenverbrauch idealerweise linear steigt. Zu den relevanten Ressourcen von Webanwendungen gehören Rechenleistung, Ein-/Ausgabeoperationen (kurz I/O) und Arbeitsspeicher, wobei Node.js vor allem die Skalierbarkeit von I/O intensiven Anwendungen verbessert [**?**]. I/O-Zugriffe sind beispielsweise Zugriffe auf Datenbanken, Webservices oder auf das Dateisystem. Node.js setzt dabei vollständig auf asynchrone Zugriffe. Dabei wartet der Thread nicht auf das Ergebnis eines I/O-Zugriffs, sondern führt andere Aufgaben aus, bis das Ergebnis verfügbar ist. Anschließend wird eine zuvor definierte Callback Funktion (siehe Code Ausschnitt 2.1) durchgeführt. Bei einem synchronen Zugriff, wie es bei einigen anderen Laufzeitumgebungen der Fall ist, würde der Thread auf das Ergebnis warten und dieses anschließend weiterverarbeiten, wobei jedoch sein Speicherplatz zum Teil belegt bleibt.[**?**] Die Vorteile hinsichtlich der Skalierbarkeit werden jedoch erst bei einer hohen Anzahl von Zugriffen erkennbar.

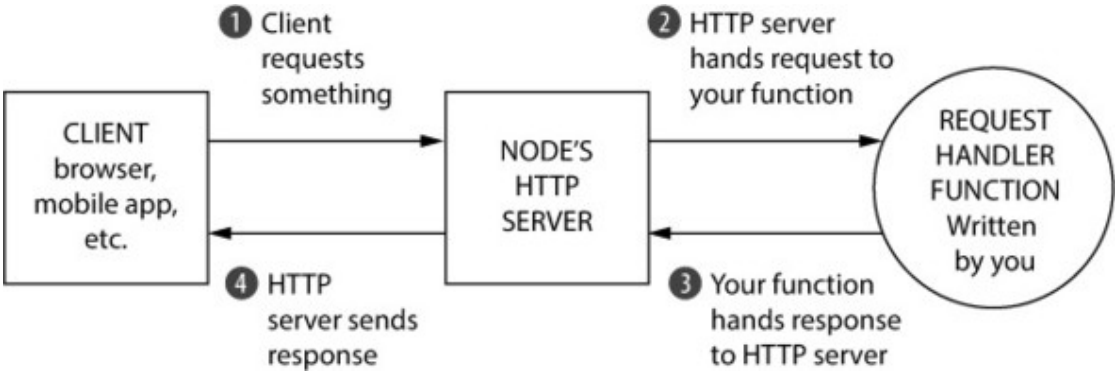
```
1 database.query( "SELECT * FROM user", function(result) {
```

[1] Quelle: `https://www.youtube.com/watch?v=EeYvFl7li9E` am 22. April 2023

```
2 result...
3 });
```

Code Ausschnitt 2.1: Beispiel einer Callback Funktion **Quelle:** [?]

Ein weiterer Vorteil der Nutzung von Node.js ist die Nutzung der gleichen Programmiersprache für Frontend und Backend. In einem Team-Projekt kann das besonders hilfreich sein, da Kommunikationsbarrieren durch unterschiedliche Programmiersprachen von Frontend und Backend niedriger sind. Natürlich versteht deshalb der Frontend-Entwickler nicht alles was der Backend-Entwickler macht und umgekehrt, jedoch gibt es eine gemeinsame Grundlage. Neben den Kommunikationsvorteilen ermöglicht die Verwendung von der gleichen Programmiersprache im Frontend und Backend das Teilen von Code. So ist es zum Beispiel möglich Callback Funktionen vom Frontend an das Backend zu senden und dort aufzurufen.
Node.js basiert auf der Verarbeitung von Requests vom Frontend und dem zurück senden von einem Result mittels dem HTTP-Protokoll. Das HTTP-Protokoll verwendet verschiedene Methoden wie GET, POST, PUT und DELETE, um unterschiedliche Aktionen durchzuführen [?]. Zum Beispiel wird GET zum Abrufen von Informationen verwendet, während POST zum Senden von Daten verwendet wird. Die Art und Weise wie ein Request verarbeitet werden soll ist dabei selbst zu definieren (siehe Abbildung 2.3). Dabei kann der Request sein, eine bestimmte Seite zu laden, womit dann mit der entsprechenden HTML-Datei geantwortet wird, oder es kann als API genutzt werden, um Beispielsweise Daten einer Datenbank zu übermitteln. Um die Verarbeitung dieser Anfragen weniger komplex zu gestalten gibt es die Erweiterung *Express* für Node.js.



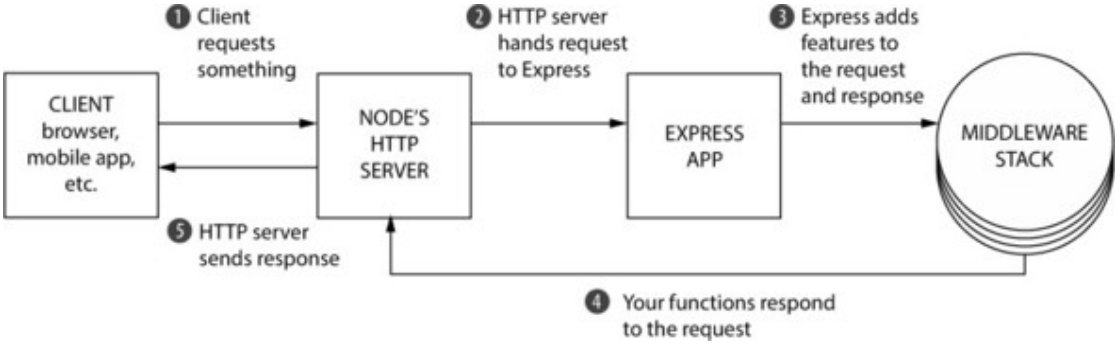
Quelle: [?]

Abbildung 2.3: Ablauf einer Anfrage an einen Node.js Server

Express

Express ist ein leichtgewichtiges und sehr beliebtes Web-Frameworks, welches unter Node.js zur Verfügung steht. Es dient zur Vereinfachung der API von Node.js und stellt hilfreiche Funktionen bereit [?]. Es ermöglicht beispielsweise die Verwendung von *Middleware* und *Routing*.

Middleware ermöglicht, dass eine Anfrage an den Node.js Server nicht ausschließlich von einer Funktion bearbeitet werden muss, welche das Ergebnis zurücksendet, sondern von mehreren Funktionen, die sich um verschiedene Teile der Request kümmern (siehe Abbildung 2.4). Diese Funktionen heißen *Middleware*. Dabei gibt es eine von uns definierte Reihenfolge der Middlewares. Zum Beispiel können wir definieren, dass zu erst der Request von einer Middleware geloggt werden soll, anschließend soll der Benutzer authentifiziert werden und falls der Benutzer eine URL aufrufen möchte, für die er keine Berechtigung hat wird eine „not authorized“ Seite zurück gesendet und die nächste Middleware wird nicht aufgerufen. Ansonsten wird die nächste Middleware der Kette ausgeführt, wie zum Beispiel das senden von Informationen (siehe Abbildung 2.5). Ein Vorteil der Nutzung von Middlewares ist, dass es bereits viele vordefinierte Middlewares (auch von Dritten) gibt, welche nützliche Funktionalitäten mitbringen. Die Anfrage des Frontends in mehrere kleinere Funktionen aufzuteilen, anstatt eine Funktion zu schreiben, welche sich um all dies kümmert verringert die Komplexität und erhöht die Modularität.



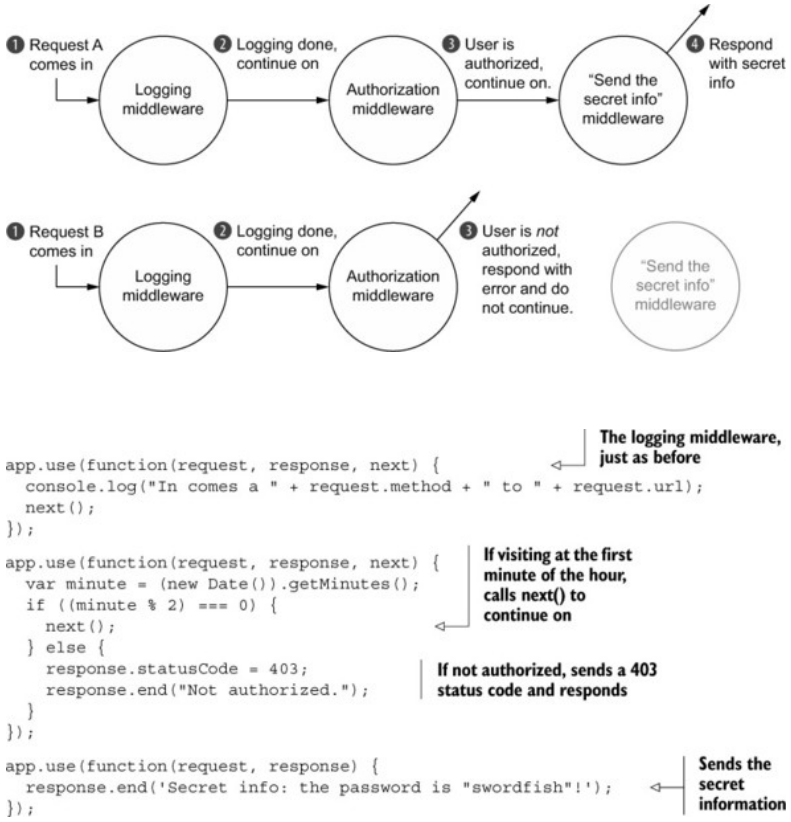
Quelle: [?]

Abbildung 2.4: Ablauf einer Anfrage an einen Node.js Server mit Express

Routing hilft dabei zu identifizieren bei welchem Request welche Middleware ausgeführt werden soll (siehe Abbildung 2.6). Beispielsweise kann eine Anfrage an die URL `/auth` mit den angegeben Anmeldedaten des Benutzers gesendet werden. Unter diesem Pfad können wir dann bestimmte Middlewares verwenden, welche sich mit der Authentifizierung des Benutzers befassen.

2.2.2 Socket.io

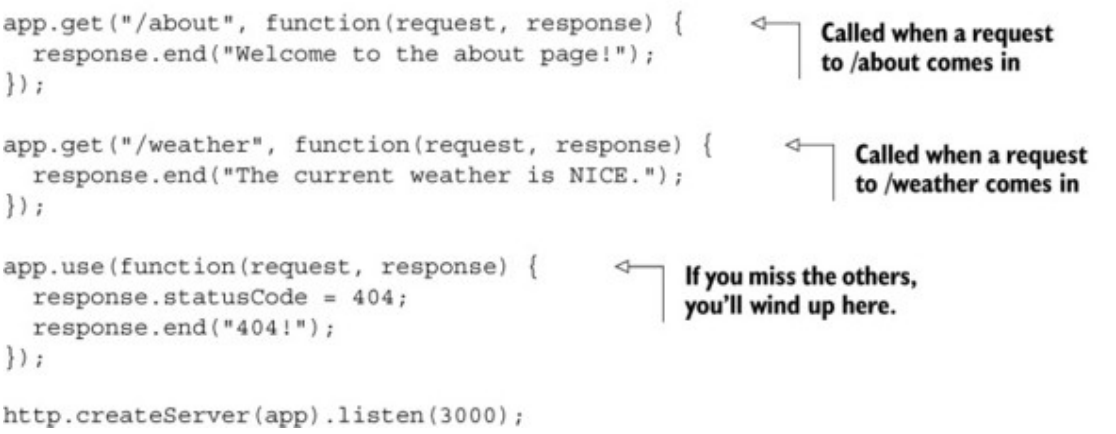
Die Kommunikation mit dem HTTP-Protokoll über Express hat den Nachteil, dass für jeden Datenaustausch eine neue Verbindung aufgebaut und wieder geschlossen wird, was zu einer Latenz führt, welche für Echtzeit-Anwendungen ungeeignet ist. Diese Problematik behebt das Framework *socket.io*.
Es ermöglicht eine direkte, bidirektionale Echtzeitübertragung von Daten mittels Websockets, long-polling und fünf anderen Protokollen zwischen den Clients und dem Server. Diese Echtzeitkommunikation ist für viele Spiele, wie auch für diese Schach-App, essen-



Quelle: [?]

Abbildung 2.5: Beispiel der Nutzung von Middlewares

tiell. So können beim Spielen mit Schachuhr Millisekunden entscheidend sein. Neben der Echtzeitkommunikation überprüft socket.io unter anderem Timeouts, Verbindungsabbrüche, stellt Verbindungen automatisch wieder her und sorgt dafür, dass die Events in der richtigen Reihenfolge beim Server und beim Client ankommen.
Die Kommunikation mit Socket.io läuft ausschließlich über Events. So kann man sowohl beim Client, als auch bei dem Server Eventlistener definieren, die auf ein bestimmtes Event hören und darauf hin eine Funktion auf den übertragenen Daten anwenden. Diese Eventlistener (definiert mit der Funktion `.on()`) haben als ersten Parameter den Namen des Events als String, auf den dieser Listener hören soll und als zweiten Parameter die auszuführende Callback Funktion, welche mit den Parametern aufgerufen wird, die beim senden des Events übertragen wurden.
Events können basierend auf verschiedenen Aktionen wie zum Beispiel dem Drücken eines Buttons im Frontend oder als Reaktion eines eingegangenen Events auf dem Server gesendet werden (mit der Funktion `.emit()`) (siehe Abbildung 2.7). Der erste Parameter der `emit`-Funktion ist wieder der Name des Events als String und im Anschluss kann man beliebig viele Parameter übertragen mit denen die Callback-Funktion des Listeners aufgerufen wird.



Quelle: [?]

Abbildung 2.6: Beispiel der Nutzung von Routing

Ein wichtiges Feature von socket.io sind die Räume[2]. Sockets im Backend können ihnen Beitreten und sie Verlassen. Serverseitig kann man dadurch an alle Sockets, die in einem bestimmten Raum sind etwas senden, ohne es allen einzeln schicken zu müssen (siehe Abbildung 2.8). Hier kann man sich entschließen das Event an alle clients im Raum zu versenden (`io.to(...).emit(...)`) oder an alle, außer den sender (`socket.to(...).emit(...)`) (siehe Code Ausschnitt 2.2).
Des weiteren erhält jede socket beim Verbinden eine eigene ID, die ebenfalls als Raum genutzt werden kann. Dementsprechend ist `io.to(socket.id).emit('hello');` äquivalent zu `socket.emit('hello');`.
Socket.io unterstützt wie Express Middlewares, welche bei einem Verbindungsaufbau ausgeführt werden. Dabei sind die übergebene Argumente die socket und die next Funktion als nächste Middleware. Es bietet sich daher an Authetifikation oder die Initialisierung von Listenern als Middleware zu behandeln.

```
1  //Server
2  io.on("connection", (socket) => {
3    socket.join("Chat");
4    socket.on("message", (text) => {
5      socket.to("Chat").emit("message", text);
6    }
7    //Empfangen der Nachricht und weiterleiten an alle im Raum, ausser
         Sender.
8  });
9
10 //Frontend
11 ...
12 socket.emit("message", "hello world"); //Senden
13 socket.on("message", text => console.log(text)); //Empfangen
14 ...
```

[2]Quelle: `https://socket.io/docs/v4/rooms/` am 22. April 2023

```
import { Server } from "socket.io";

const io = new Server(3000);

io.on("connection", (socket) => {
  // send a message to the client
  socket.emit("hello", "world");

  // receive a message from the client
  socket.on("howdy", (arg) => {
    console.log(arg); // prints "stranger"
  });
});
```

```
import { io } from "socket.io-client";

const socket = io("ws://localhost:3000");

// receive a message from the server
socket.on("hello", (arg) => {
  console.log(arg); // prints "world"
});

// send a message to the server
socket.emit("howdy", "stranger");
```

Quelle:
[?]

Abbildung 2.7: Simples Beispiel der Initialisierung einer socket.io Verbindung und das Senden und Empfangen von Events

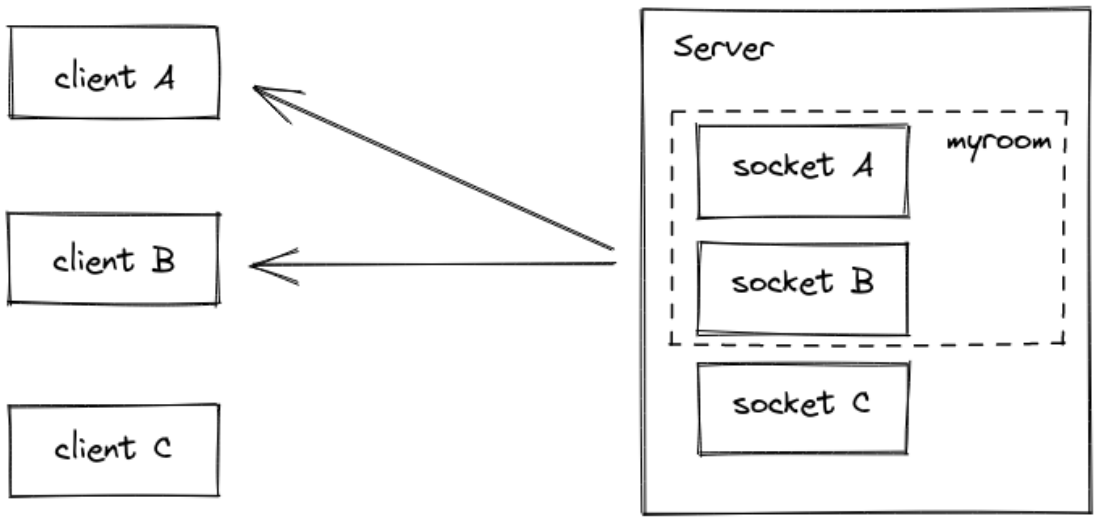
Code Ausschnitt 2.2: Beispiel zum Beitreten Raums und das senden eines Events in diesen Raum

//QUELLE socket.io im nodejsbook

2.2.3 React

React

React ist eine der beliebtesten (nach einer Umfrage von 2022 von Stack Overflow sogar die beliebteste [?]) Frontend Javascript Bibliotheken. Es basiert auf Komponenten, welche wiederverwendbar und kombinierbar sind und vereinfacht die Verwaltung von Interaktionen mit User Interfaces. Dabei benutzt React eine Syntax Erweiterung namens *JSX*. Mit dieser Erweiterung ist es erlaubt HTML Elemente in JavaScript Dateien zu verwenden, welches es ermöglicht die Logik hinter getrennten HTML und JavaScript Dateien in eine Datei zu kombinieren. Die Idee hinter React ist, dass wenn sich nur ein bestimmter Teil des User Interfaces im Vergleich zum aktuell sichtbaren User Interface ändert, auch nur dieser Teil neu gerendert wird und nicht das ganze User Interface. Diese reaktiven Änderungen veranlasst React mittels seiner *Hooks* [?].
Ich werde die Art und Weise wie React und seine Hooks funktionieren an dem Beispiel 2.3 erklären. In diesem Beispiel implementieren wir die Komponente *ExampleComponent*, welche zum Beispiel mittels dem Tag `<ExampleComponent initialCount={10} loadingDelay={3000} />` verwendet werden kann.
Die zwei übergebenen Variablen *initialCount* und *loadingDelay* werden auch **props** genannt, welche in der Komponente verwendet werden können. Eine Komponente ist eine Funktion, welche als Rückgabe den HTML-Code hat, welcher angezeigt werden soll.



Quelle: `https://socket.io/docs/v4/rooms/` am 27. April 2023

Abbildung 2.8: Darstellung eines Raumes *myroom* mit zwei sockets

Die Komponente hat den lokalen State *count*, welchen wir mittels der **useState** Hook initialisieren. Ein State beschreibt einen Zustand der Komponente und eine Änderung veranlasst die Komponente neu zu laden. Die Funktion *useState* nimmt als Argument den initialen Wert des States und gibt uns zwei Elemente zurück, einmal der sich verändernde State *count* und die Funktion *setCount*, um einen neuen Wert in den State *count* zu schreiben. Der zurückgegebenen Funktion *setCount* kann entweder ein konkreter Wert übergeben werden, oder aber eine Funktion welche beschreibt wie der neue Wert sich aus dem alten Wert bilden soll (siehe Zeile 31 in Beispiel 2.3).
Die Hook **useEffekt** nimmt zwei Argumente, eine Funktion und ein sogenanntes *Dependency Array*. Das Array enthält Variablen, deren Wertänderung das Ausführen der übergebenen Funktion auslöst. So wird in unserem Beispiel die Funktion einmal beim ersten Rendern der Komponente und dann bei jeder Änderung von *count* oder dem prop *loadingDelay* ausgeführt. Dadurch bleibt der Titel dieser Beispiel-Webanwendung immer konsistent mit dem aktuellen *count* State. Als Rückgabe kann die übergebene Funktion eine weitere Funktion haben, welche ausgeführt wird, sobald die Komponente *unmounted* wird. Das ist eine Phase im Lebenszyklus einer Komponente, die ausgeführt wird, sobald eine Komponente nicht mehr angezeigt wird, weil zum Beispiel auf eine andere Unterseite navigiert wird. In unserem Fall soll dann der Titel der Webanwendung nicht mehr den aktuellen Count repräsentieren, sondern „React App“.
useCallback ist eine Hook, welche unnötige Code Ausführungen vermeidet und daher ausschließlich performante Vorteile bietet. Sie nimmt die gleichen Argumente wie die *useEffect* Hook und durch sie können wir eine Funktion definieren, welche nur neu initialisiert wird, falls sich eine der Variablen im *Dependency Array* ändert. Würden wir sie als reguläre JavaScript Funktion definieren, würde immer wenn die Komponente

gerendert wird die Funktion neu initialisiert werden.
Ein weiteres wichtiges Konzept in React ist der *Context*. Mit ihm lassen sich Daten über mehrere Ebenen von verschachtelten Komponenten verfügbar machen, ohne dass man sie explizit als Prop an alle Komponenten weitergeben muss. Ein Kontext lässt sich mittels der Hook **useContext** importieren. In unserem Beispiel verwenden wir ihn um ein ThemeContext zu importieren, der die Hintergrundfarbe unserer Komponente bestimmt (siehe Zeile 35 in Beispiel 2.3).
In der Rückgabe der Komponente können wir nicht nur HTML, sondern auch JavaScript innerhalb von geschweiften Klammern verwenden. So prüfen wir in unserem Beispiel mit dem ternären Operator den Wert von *isLoading* und zeigen entsprechende Elemente an (siehe Zeilen 36-44, Beispiel 2.3).

```
1  import React, { useState, useEffect, useCallback, useContext } from
       "react";
2
3  // Beispiel Context
4  const ThemeContext = React.createContext({ theme: "light" });
5
6  // Beispiel Component Props
7  function ExampleComponent({ initialCount, loadingDelay }) {
8    // Beispiel useState
9    const [count, setCount] = useState(initialCount || 0);
10   const [isLoading, setIsLoading] = useState(true);
11
12   // Beispiel useContext
13   const { theme } = useContext(ThemeContext);
14
15   // Beispiel useEffect
16   useEffect(() => {
17     document.title = `Count: ${count}`;
18
19     const timer = setTimeout(() => {
20       setIsLoading(false);
21     }, loadingDelay || 2000);
22
23     return () => {
24       document.title = "React App";
25       clearTimeout(timer);
26     };
27   }, [count, loadingDelay]);
28
29   // Beispiel useCallback
30   const incrementCount = useCallback(() => {
31     setCount((prevCount) => prevCount + 1);
32   }, []);
33
34   return (
35     <div style={{ backgroundColor: theme === "light" ? "#fff" : "#333"
         }}>
36       {isLoading ? (
37         <p>Loading...</p>
```

```
38        ) : (
39          <>
40            <p>Count: {count}</p>
41            <button onClick={incrementCount}>Increment count</button>
42          </>
43        )}
44      </div>
45    );
46  }
47
48  export default ExampleComponent;
```

Code Ausschnitt 2.3: Beispiel einer React Komponente

React Router

React Router ermöglicht die Erstellung von Anwendung mit mehreren Seiten unter verschiedenen Pfaden [**?**]. Das ist sinnvoll um als Benutzer direkt einen Pfad angeben zu können um auf die gewünschte Seite zu kommen oder sie zu teilen. Ein Beispiel der Funktion von verschiedenen Komponenten auf verschiedenen Pfaden finden Sie in Abbildung 2.4.
In diesem Beispiel wird unter dem Pfad „/“ die Komponente `Dashboard` angezeigt, während unter dem Pfad „/orders“ die React Komponente `Orders` gerendert wird. Dafür werden die Komponenten `BrowserRouter`, `Routes` und `Route` des Pakets `react-router-dom` benötigt.

- **BrowserRouter** ermöglicht alle Routing Funktionen und Komponenten zu verwenden.
- **Routes** enthält alle Definition der Pfade. Es kann auch mehrmals verwendet werden um verschiedene Gruppen von Routen zu definieren.
- **Route** legt eine einzelne Route fest. Im `path` kann angegeben werden, welcher Pfad diese Route aktivieren soll und `element` definiert die React Komponente, welche unter diesem Pfad gerendert werden soll.

React Router ermöglicht alledings auch noch ein paar weitere Funktionen, wie zum Beispiel die Hooks `useParams()` und `useNaviagte()` [**?**].
Es ist möglich bei einer `Route` Komponente mit „:“ in einem Pfad einen String zu übertragen. Auf diesen String kann dann mit der `useParams()` Hook zugegriffen werden, wie bei `OrderDetail` in dem Beispiel 2.4.
Mit der `useNavigate()` Hook kann zu einem bestimmten Pfad gesprungen werden. Ein Beispiel dafür ist die `navigateToOrders()` Funktion, welche beim klicken auf den Button in der App Komponente ausgelöst wird (Beipiel 2.4).

```
1  import { BrowserRouter, Routes, Route, useNavigate } from
       'react-router-dom';
2  import { useCallback } from 'react';
```

```
3  import Dashboard from './routes/Dashboard';
4  import Orders from './routes/Orders';
5
6  function App() {
7    const navigate = useNavigate();
8
9    const navigateToOrders = useCallback(() => {
10     navigate('/orders');
11   }, [navigate]);
12
13   return (
14     <BrowserRouter>
15       <Routes>
16         <Route path="/" element={<Dashboard />} />
17         <Route path="/orders" element={<Orders />} />
18         <Route path="/orders/:id" element={ <OrderDetail /> } />
19       </Routes>
20       <Button onClick={navigateToOrders}> To Orders </Button>
21     </BrowserRouter>
22   );
23
24
25 export default App;
26
27 function OrderDetail() {
28
29   const params = useParams();
30
31   const orderId = params.id;
32
33   useEffect(() => {
34   //fetch Data with orderId
35   }, [])
36
37   return (
38   // Show Data
39   );
40 }
41
42 export default OrderDetail;
```

Code Ausschnitt 2.4: Beispiel von verschiedenen Komponenten auf verschiedenen Pfaden
Quelle: [?] (abgewandelt)

2.2.4 PostgreSQL und Redis

PostgreSQL

PostgreSQL ist ein Objektrelationales Open-Source Datenbanksystem, welches erstmals 1989 veröffentlicht wurde [?]. Die Verwaltung von Datenbanken basiert auf sogenannte Datenbankmanagementsystemen (DBMS). Das beliebteste DBMS für PostgreSQL ist

pgAdmin[3]. In relationalen Datenbanken sind Daten in Tabellen organisiert. Zur Bearbeitung und Auswertung von solchen Datenbanken wird die Structured Query Language (**SQL**) verwendet, die in drei Bereiche unterteilt ist [**?**]:

- Data Definition Language (DDL): Um Datenbanken, Tabellen und ihren Strukturen anzulegen, zu ändern und zu löschen.
- Data Manipulation Language (DML): Zum Einfügen, Ändern, Löschen und Aktualisieren von Daten in Tabellen.
- Data Control Language (DCL): Zur Administration von Datenbanken

Tabellen bestehen aus Zeilen, die als Tupel bezeichnet werden, und Spalten, die als Attribute bezeichnet werden. Jedes Attribut hat einen bestimmten, von uns definierten Wertebereich. Beispielsweise kann ein Attribut „Preis“ eine Zahl mit zwei Nachkommastellen oder ein Attribut „Name“ eine Zeichenkette mit maximal 20 Zeichen sein. Attributen können bestimme Restriktionen (auch *Constraints* genannt) zugewiesen werden, wie zum Beispiel die UNIQUE Restriktion, welche definiert, dass jeder Wert des Attributs nur einmal in der Tabelle vorkommen darf. Ein weiteres wichtiges Konzept sind Primär- und Fremdschlüssel. Mit Hilfe von ihnen können Tupel verschiedener Tabellen in Beziehung gebracht werden. Ein Primärschlüssel ist ein Attribut, welches jeden Tupel einer Tabelle eindeutig identifiziert und welches als Fremdschlüssel in anderen Tabellen referenziert werden kann. Als Beispiel könnte eine Artikelnummer in einer Tabelle der Artikel als Primärschlüssel genutzt werden, welche in einer Tabelle Rechnung mit verschiedenen abgeschlossenen Bestellungen als Fremdschlüssel referenziert werden kann.
Zudem ist PostgreSQL ACID-konform. **ACID** steht für folgende Fachbegriffe [**?**]:

- *Atomicity (Atomarität)*: eine Transaktion, wie das Einfügen eines Tupels oder das Erstellen einer Tabelle, wird entweder ganz oder gar nicht ausgeführt.
- *Consistency (Konsistenz)*: Sicherstellung, dass die Datenbank immer in einem konsistenten Zustand ist, auch wenn eine Transaktion unter- oder abgebrochen wird.
- *Isolation*: Während einer Transaktion wird die Datenbank isoliert, da während einer Transaktion ein inkonsistenter Zustand herrschen kann. Diese Isolation wird am Ende der Transaktion aufgehoben.
- *Durability (Dauerhaftigkeit)*: Nach einer abgeschlossenen Transaktion sind die Änderungen an der Datenbank dauerhaft abgespeichert, sodass beispielsweise ein Systemabsturz die Daten nicht gefährden kann.

In Node.js kann auf eine PostgreSQL Datenbank mittels dem Framework *node-postgres* zugegriffen werden[4].

[3] Quelle: `https://www.pgadmin.org/` am 22. April 2023
[4] Quelle: `https://node-postgres.com/` am 22. April 2023

Redis

Redis ist eine No-SQL (*Not only SQL*) Datenbank, welche nicht wie relationale Datenbanken auf Tabellen basieren, sondern in diesem Fall auf *Key-Value*-Paaren. Redis zeichnet sich vor allem durch seine verschiedenen Datentypen und seine schnellen Schreib- und Lesevorgänge aus, welche durch die Speicherung im Arbeitsspeicher resultieren [**?**]. Daher ist es gut für Daten geeignet, welche eine hohe Speicherungs- oder Abruffrequenz haben. Obwohl Redis hauptsächlich im Arbeitsspeicher arbeitet bietet es auch Optionen zur Datensicherung auf der Festplatte, um Datenverluste zu vermeiden. Oft dient Redis als Cache-Speicher, um häufig verwendete Daten temporär zu speichern und dadurch den Zugriff auf die Daten zu beschleunigen[5]. Zu den möglichen Datentypen zählen Strings, Listen, Sets, Hashes, sortierte Sets, Streams und einige weitere[5].
Zudem ermöglicht Redis eine gute Skalierbarkeit indem mehrere Redis Instanzen verbunden werden, anstatt eine Instanz hoch zu skalieren.

2.2.5 Weitere verwendete Bibliotheken

JWT

JWT (*JSON Web Token*) ist ein offener Standard, der eine sichere Möglichkeit bietet Informationen in Form eines JSON Objekts zu übertragen[6]. Diesen Informationen kann vertraut werden, da sie mittels eines privaten Server seitigen secrets mit verschiedenen Algorithmen verschlüsselt (*signiert*) und wieder entschlüsselt werden (siehe Abbildung 2.9). Der meist verwendete Anwendungsbereich für JSON Web Tokens ist die Authentifizierung, bei der der vom Backend generierte Token in einer Session oder einem Cookie gespeichert wird. Dieser kann beim Laden einer Seite aus dem HTTP Request entnommen und mittels des secrets verifiziert werden. Dabei ist wichtig zu beachten, dass der Token nicht Client seitig manipulierbar sein darf, da dies ein Sicherheitsrisiko darstellen kann. Beispielsweise kann das mit einem HTTP-Only Cookie[7] erreicht werden.

Chakra UI

Chakra UI ist eine simple Komponenten Bibliothek, welche das designen von React Anwendungen vereinfacht[8]. Es stellt Komponenten zur Verfügung, welchen verschiedene Attribute zugeordnet werden können um diese nach eigenem ermessen zu designen (siehe Code Beispiel 2.5 und Abbildung 2.10).

```
1 import * as React from "react";
2 import { Box, Center, Image, Flex, Badge, Text } from "@chakra-ui/react";
3 import { MdStar } from "react-icons/md";
4
5 export default function Example() {
6   return (
```

[5] Quelle: `https://redis.io/` am 22. April 2023
[6] Quelle: `https://jwt.io/introduction` am 22. April 2023
[7] Quelle: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies` am 22. April 2023
[8] Quelle: `https://chakra-ui.com/` am 22. April 2023

```
7       <Center h="100vh">
8         <Box p="5" maxW="320px" borderWidth="1px">
9           <Image borderRadius="md" src="https://bit.ly/2k1H1t6" />
10          <Flex align="baseline" mt={2}>
11            <Badge colorScheme="pink">Plus</Badge>
12            <Text
13              ml={2}
14              textTransform="uppercase"
15              fontSize="sm"
16              fontWeight="bold"
17              color="pink.800"
18            >
19              Verified &bull; Cape Town
20            </Text>
21          </Flex>
22          <Text mt={2} fontSize="xl" fontWeight="semibold"
                lineHeight="short">
23            Modern, Chic Penthouse with Mountain, City & Sea Views
24          </Text>
25          <Text mt={2}>$119/night</Text>
26          <Flex mt={2} align="center">
27            <Box as={MdStar} color="orange.400" />
28            <Text ml={1} fontSize="sm">
29              <b>4.84</b> (190)
30            </Text>
31          </Flex>
32        </Box>
33      </Center>
34    );
35  }
```

Code Ausschnitt 2.5: Beispiel mit Chakra UI designten React Komponente (siehe Abbildung **Quelle:** `https://chakra-ui.com/` am 27. April 2023

Formik und Yup

Formik ist die beliebteste Open-Source-Bibliothek für Formulare in React [9]. Sie vereinfacht die Handhabung von Formularen und bietet Funktionen wie Validierung der eingegebenen Werte, Fehlermeldungen und Unterstützung für mehrstufige Formulare.
Yup ist eine Bibliothek zur einfachen Definition von Schemata, die von bestimmten Formularen erfüllt werden sollen. Sie ermöglicht die Erstellung komplexer Schemata mit wenig Code[10].
Die Integration von Yup-Schemata in Formik-Formularen ist bereits unterstützt, was eine einfache Handhabung von Überprüfungen und Fehlerbehandlungen bei Benutzereingaben ermöglicht.

[9] Quelle: `https://formik.org/` am 22. April 2023
[10] Quelle: `https://github.com/jquense/yup` am 22. April 2023

chess.js

chess.js ist eine Schach Bibliothek, welche die gesamte Schachlogik zur Verfügung stellt[11]. Es bietet Methoden, welche alle aktuell möglichen Züge ausgibt, einen Zug ausführt und verschiedene Notationen des Zuges zurückgibt, überprüft ob es sich um ein Schachmatt oder Patt handelt, eine Partie mittels einer FEN[12] oder PGN[13] Notation laden kann und vieles weitere.

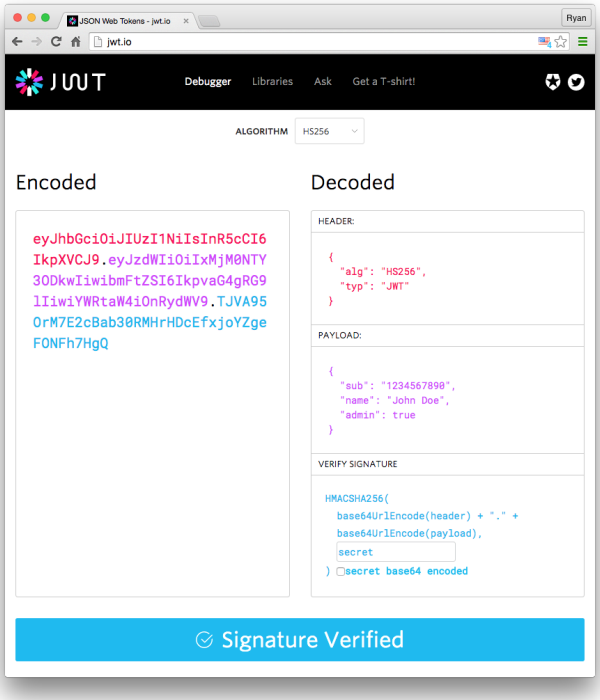
chessground

chessground ist ein Open-Source-Schach-User-Interface, das ursprünglich für die Online-Schachplattform `lichess.org` entwickelt wurde[14]. Es bietet zahlreiche Konfigurationsmöglichkeiten, wie zum Beispiel Animationen beim bewegen von Figuren, Auswahl der anklickbaren und bewegbaren Figuren, Anpassung des Figurendesigns und vieles mehr. Die eigentliche Schachlogik ist nicht enthalten, sodass verschiedene Schachvarianten mit Sonderregelungen implementiert werden können.

bcrypt

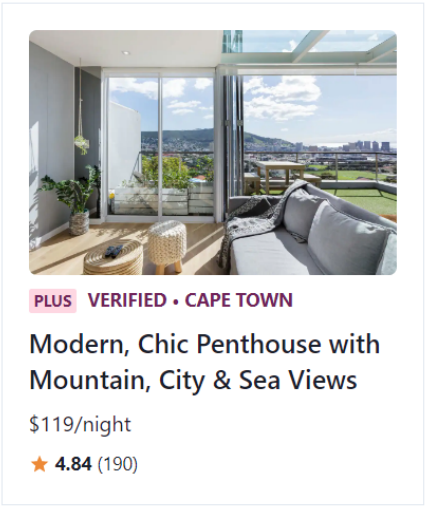
bcrypt ist eine Bibliothek welche entwickelt wurde um Passwörter zu verschlüsseln. Es basiert auf Blowfish, einem Verschlüsselungsalgorithmus, und löst das Problem, dass durch schnellere Hardware Passwörter immer schneller kodiert und dekodiert werden können und dadurch Sicherheitsrisiken entstehen. Es löst dieses Problem dadurch, dass es die Passwörter um einen selbst definierbaren Faktor zeitlich länger kodiert indem es mehrere Iterationen durchführt[15].

[11] Quelle: `https://github.com/jhlywa/chess.js/` am 22. April 2023
[12] Quelle: `https://de.wikipedia.org/wiki/Forsyth-Edwards-Notation` am 22. April 2023
[13] `https://de.wikipedia.org/wiki/Portable_Game_Notation` am 22. April 2023
[14] Quelle: `https://github.com/lichess-org/chessground` am 22. April 2023
[15] Quelle: `https://auth0.com/blog/hashing-in-action-understanding-bcrypt/`



Quelle: `https://jwt.io/introduction` am 27. April 2023

Abbildung 2.9: Beispiel eines verschlüsselten Tokens von JWT



Quelle: `https://chakra-ui.com/` am 27. April 2023

Abbildung 2.10: Darstellung der mit Chakra UI designten React Komponente aus dem Code Beispiel 2.5

3 Systemarchitektur

3.1 Einführung

In diesem Kapitel wird die Systemarchitektur der Anwendung vorgestellt, indem erläutert wird wie die verschiedenen Komponenten und Technologien zusammenarbeiten und miteinander kommunizieren. Die Anwendung ist in zwei Hauptkomponenten unterteilt: das Frontend und das Backend. Das Frontend ist für die Darstellung der Benutzeroberfläche und die Interaktion mit dem Benutzer verantwortlich, während das Backend die Spiellogik, die Verwaltung der Benutzerdaten und die Echtzeit-Kommunikation zwischen den Spielern steuert.
Die Anwendung verwendet moderne Web-Technologien, um eine reaktive und benutzerfreundliche Oberfläche zu schaffen. Das Frontend basiert auf dem React-Framework[1], das es ermöglicht, wiederverwendbare Komponenten zu entwickeln und den Anwendungsstatus effizient zu verwalten. Das User-Interface basiert auf Chakra UI[2], einem modernen und flexiblen Komponenten-Bibliothekssystem, das die Entwicklung von responsiven und zugänglichen Benutzeroberflächen erleichtert. Die Benutzerführung und die Kommunikation zwischen den React-Komponenten sind so gestaltet, dass sie eine nahtlose und intuitive Benutzererfahrung bieten.
Auf der Backend-Seite wird Node.js[3] mit dem Express-Framework verwendet, um einen leistungsstarken und skalierbaren Server bereitzustellen. Die API-Endpunkte und die Echtzeitkommunikation mittels Socket.io ist so konzipiert, dass sie den Anforderungen der verschiedenen Frontend-Komponenten gerecht werden und die Kommunikation zwischen Frontend und Backend erleichtern. Für die Speicherung und Verwaltung der Benutzerdaten zum Anmelden wird eine PostgreSQL[4]-Datenbank verwendet, die aufgrund ihrer Leistungsfähigkeit und Flexibilität ausgewählt wurde. Freundeslisten und Daten aktiver Spiele werden in einer Redis[5]-Datenbank gespeichert, die sich durch hohe Leistung und niedrige Latenz auszeichnet, insbesondere bei Lese- und Schreibvorgängen. Redis, eine In-Memory-Datenstruktur, eignet sich ideal für Anwendungen, bei denen schnelle Zugriffszeiten und Skalierbarkeit wichtig sind. Die Kombination von PostgreSQL und Redis ermöglicht eine effiziente Verwaltung sowohl persistenter als auch flüchtiger Daten und fördert eine optimale Benutzererfahrung.

[1] [?]
[2] [?]
[3] [?]
[4] [?]
[5] [?]

3.2 Architekturübersicht

Das Komponentendiagramm in Abbildung 3.1 visualisiert die Hauptkomponenten und deren Schnittstellen der Kommunikation.
Im Frontend gibt es drei Komponenten, welche die Web-API verwenden: Der *UserContext*, *Login* und *SignUp*. Unsere Web-API verwendet dabei nur die Methoden GET und POST. Der *UserContext* ist verantwortlich für die Verwaltung des Benutzerzustands, während die Komponenten *Login* und *SignUp* das setzen des Benutzerzustands über das Anmelden und Registrieren unterstützen. Der *SocketContext* baut die Socket.io Verbindung für die Echtzeitkommunikation auf und stellt sie den restlichen React Komponenten zur Verfügung, um Events zu senden und zu empfangen.
Die Anfragen über die Web-API werden durch den in *authRouter* definierten Express Router entgegengenommen. Zur Behandlung werden in ihm verschieden Middlewares der Datei *authController* für verschiedene Anfragen festgelegt, welche auf die PostgreSQL Datenbank zugreifen. Die Web-API und die PostgreSQL Datenbank werden lediglich für das Registrieren und Anmelden von Benutzern verwendet.
Beim Herstellen einer Socket.io Verbindung in *SocketContext* werden im Backend die Middlewares aus *socketMiddleware* ausgeführt, welche unter anderem die Listener aus *socketController* und *socketChessContoller* initialisieren. Der Unterschied zwischen den Listenern aus den beiden Dateien ist dabei, dass *socketController* sich um allgemeine Funktionen wie das Versenden von Freundschaftsanfragen oder das senden von Informationen an das Frontend kümmert, während *socketChessController* Listener enthält, welche sich um Funktionen des Schachspiels kümmert, wie zum Beispiel das Behandeln eines neuen Zugs.
Alle Dateien im sockets Package verwenden den *redisController*, um Daten aus der Redis Datenbank abzurufen und zu speichern. Beispielsweise werden Freundschaftsanfragen, Freunde und Daten aktiver Spiele in der Redis Datenbank von dem *redisController* verwaltet, abgerufen und gespeichert.

3.3 Konzeption der Schachuhren

Bei der Konzeption der Schachuhren war ein Aspekt besonders entscheidend: Was passiert, falls ein Spieler vorübergehend keine Internetverbindung beim senden oder beim empfangen hat?
Um den Server zu entlasten wäre natürlich eine Client basierte Lösung ideal, bei der mit einem Zug auch die jeweilige aktuelle Zeit gesendet wird. Wenn ein Spieler allerdings eine schlechte Internetverbindung hat und deshalb der Zug im Backend und bei dem anderen Spieler erst verspätet ankommt können sich die Zeiten der beiden Spieler erheblich unterscheiden. Es stellt sich die Frage, welche dieser Zeiten jetzt gültig ist. Trivialerweise entsteht das gleiche Problem bei einer verzögerten Zustellung. Wenn die Zeit auf einem Client bereits abgelaufen wäre, könnte sie auf dem anderen noch weiterlaufen. Hat der Spieler dann gewonnen oder nicht?
Um dieses Problem zu umgehen liegt die einfachste Lösung darin, eine serverseitige

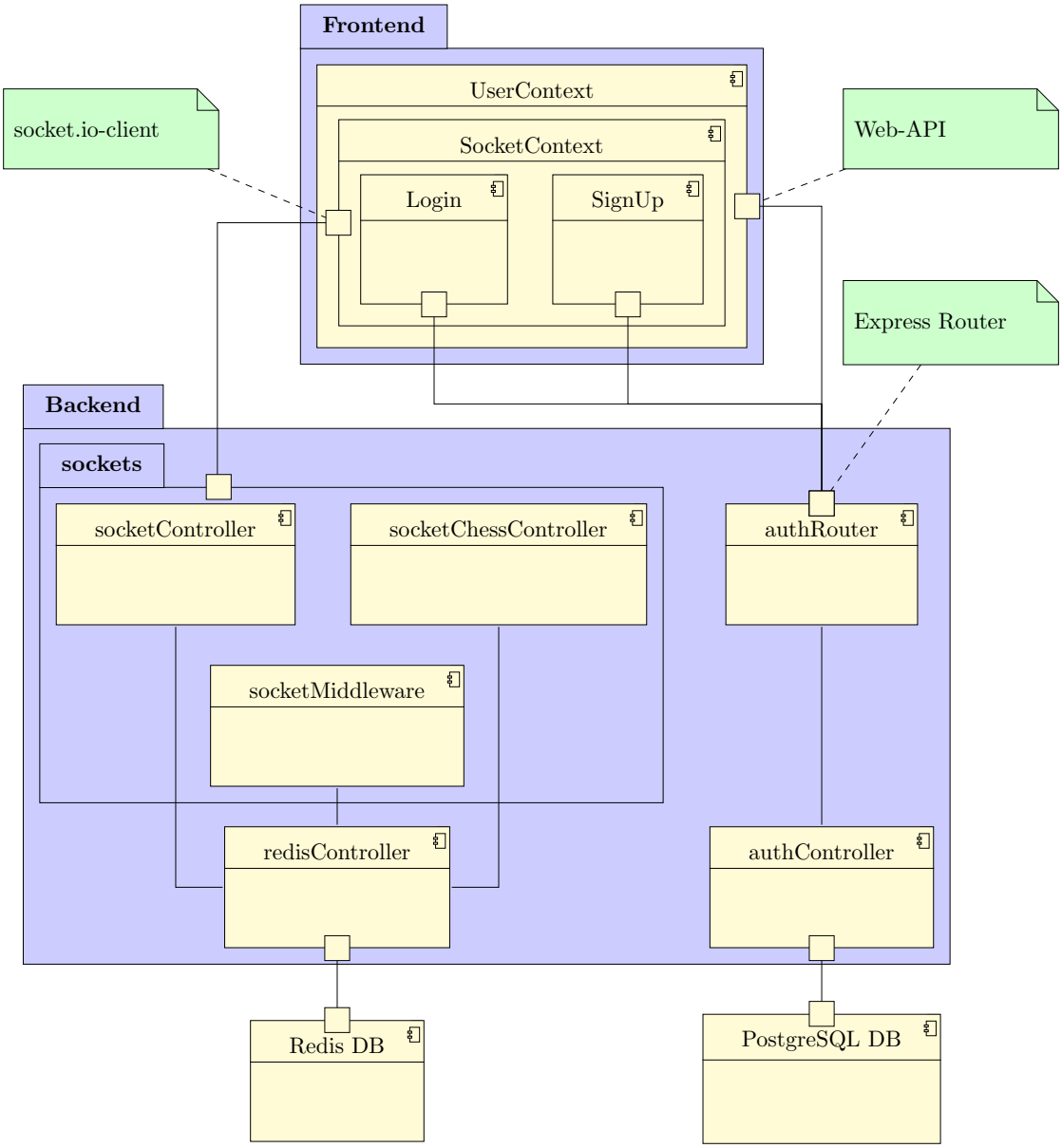


Abbildung 3.1: Komponentendiagramm der Anwenundung

Schachuhr einzuführen. Diese Uhr bestimmt die aktuellen Zeiten. Wenn ein Zug im Backend ankommt, wird die auf dem Server gültige Zeit an die Clients gesendet. Dadurch gibt es keine Unklarheiten hinsichtlich der aktuellen Zeit. Wenn eine Zeit auf dem Server abläuft wird dies durch ein Event mitgeteilt. Es kann zwar vorkommen, dass bei den Clients zu dem Zeitpunkt noch Zeit übrig ist, wenn das letzte Event aufgrund einer schlechten Verbindung verspätet eingetroffen ist, aber dieses Problem ist unvermeidbar. Durch dieses Konzept umgeht man auch das Problem, dass Client seitiger Code im

Browser manipulierbar sein kann und dadurch die Schachuhren beeinflussbar wären. Wie die Schachuhren konkret funktionieren wird in den nachfolgenden Kapiteln behandelt.

3.4 Frontend-Architektur

Das Frontend der Anwendung wurde unter Verwendung von React (Abschnitt 2.2.3), Socket.io (Abschnitt 2.2.2) und weiteren Bibliotheken aus Abschnitt 2.2.5 entwickelt. In diesem Kapitel werde ich die unterschiedlichen React-Komponenten vorstellen und erläutern, wie sie zusammenarbeiten, um bestimmte Funktionen zu erfüllen und sowohl untereinander als auch mit dem Backend kommunizieren.
Die Ordnerstruktur (Abbildung 3.2) ist wie folgt aufgebaut:

- **public:** In diesem Ordner befinden sich statische Ressourcen, wie zum Beispiel Bilder des Logos, die von der Anwendung verwendet werden.
- **components:** Dieser Ordner enthält alle React-Komponenten, die für die Anwendung verwendet werden. Sie sind modular und stellen jeweils nur ein Teil eines User Interfaces dar.
- **contexts:** Hier befinden sich die React Contexts, die zum Verwalten von globalen Zuständen und Kommunikationsschnittstellen verwendet werden.
- **themes:** Dieser Ordner enthält Dateien, die für das Design und die Anpassung des Aussehens der Anwendung mittels Chakra UI verantwortlich sind.
- **utils:** In diesem Ordner befinden sich Hilfsdateien, die ausgelagerte Funktionen zur Verfügung stellen.
- **views:** Dieser Ordner enthält die verschiedenen Seiten der Anwendung. Zu diese Seiten kann mit Hilfe des React-Routers über verschiedene Pfade navigiert werden. Diese Seiten verwenden teilweise die Komponenten aus dem components-Ordner, um eine vollständige Benutzeroberfläche darzustellen.

3.4.1 React-Komponenten

Der hierarchische Aufbau der React-Komponenten in Abbildung 3.3 zeigt die Struktur und Verschachtelung der Anwendung für angemeldete Benutzer. Nicht angemeldete Benutzer sehen lediglich die Komponenten ActiveGames und FriendList nicht, während der restliche Aufbau gleich bleibt. In diesem Abschnitt werden die wichtigsten Komponenten und ihre Funktionen innerhalb der Anwendung erläutert.

- **AccountContext:** Stellt Informationen über den Benutzerstatus allen Folgenden Komponenten mittels eines React-Contexts zur Verfügung. Diese Informationen beinhalten, ob ein Benutzer angemeldet ist und falls er das ist seinen Benutzernamen.

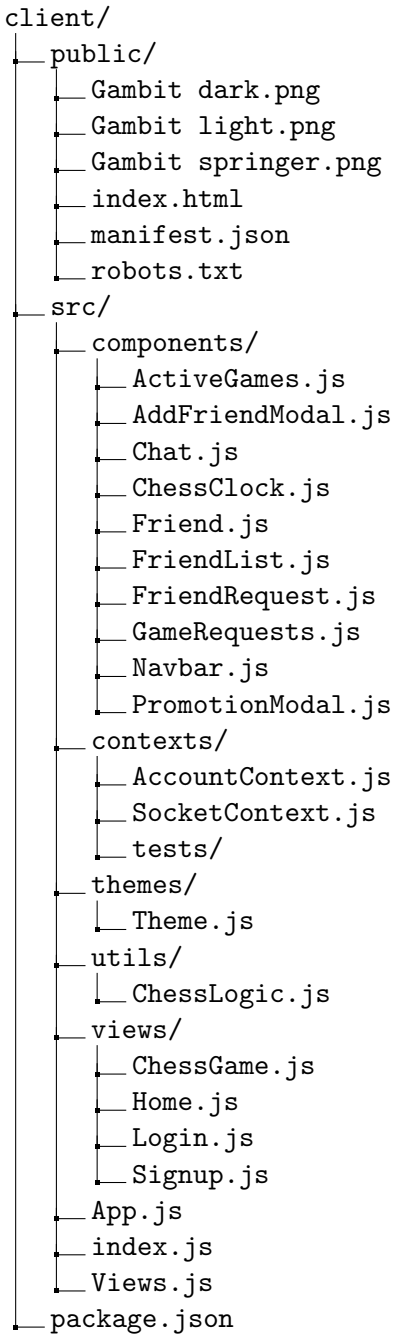


Abbildung 3.2: Ordnerstruktur des Frontends

- **SocketContext:** In diesem React Context wird eine socket.io Verbindung mit dem Server hergestellt und allen darauf folgenden Komponenten bereitgestellt.

- **ChakraBaseProvider und ColorModeScript:** Diese importierten Komponenten von ChakraUI stellen die Funktionen zum designen bereit. Dazu gehören Beispielsweise das Verwenden des globalen Zustands des Farbenschemas (dunkel oder hell) oder das Zugreifen auf definierte Stile.

- **Views:** Diese Komponente beinhaltet das User Interface. Mit Hilfe des React-Routers werden hier die Komponenten des `view`-Ordners unter einem bestimmten Pfad definiert. Des weiteren beinhaltet es die Komponenten GameRequest und Navbar, welche durch die Definition in dieser Komponente auf jedem Pfad vorhanden sind.
  - **Navbar:** Die Navigationsleiste besteht aus dem Logo und einem Button zum wechseln des Farbschemas. Je nachdem, ob ein Benutzer angemeldet ist oder nicht beinhaltet es noch Buttons zum Anmelden, Registrieren oder Abmelden (siehe Abbildungen 3.4 & 3.5).
  - **GameRequests:** Diese Komponente ist dafür Verantwortlich beim Eingang einer Spielanfrage eines Freundes, dieses als Modal darzustellen und bietet die Möglichkeit diese Anfrage zu beantworten (siehe Abbildung 3.6).

- **Home:** Diese Komponente stellt die Startseite dar und enthält die Buttons zum Starten eines Spiels mit verschiedenen Schachuhr Konfigurationen (siehe Abbildung 3.4). Diese Buttons sind in einigen online Schachplattformen (Beispielsweise `lichess.org`) bereits gängig und benötigt keine zusätzliche Erklärung. Ist ein Benutzer angemeldet sind auch noch die Komponenten ActiveGames und FriendList auf der rechten Seite vorhanden (siehe Abbildung 3.5).
  - **ActiveGames:** ActiveGames ist eine Komponente die alle derzeit aktiven Spiele mit den Informationen der Benutzernamen und wer welche Farbe spielt als Buttons darstellt (siehe Abbildung 3.5). Beim klicken auf einen dieser Buttons wird zu der aktiven Partie navigiert.
  - **FriendList:** Diese Komponente verwaltet alle Freunde und Freundschaftsanfragen eines Benutzers, während die Darstellung und Interaktion die Unterkomponenten *Friend* und *FriendRequest* übernehmen. Die Funktionsweise der Komponente mit seinen Unterkomponenten wird in Abschnitt 3.4.4 erläutert.
    - **Friend:** Übernimmt die Darstellung eines Freundes. Mittels eines farbigen Punktes ist erkennbar, ob dieser Freund gerade online ist (grün) oder nicht (rot) (siehe Abbildung 3.5). Ist er online erscheint noch mindestens ein weiterer Button. Es beinhaltet ein Icon in Form von gekreuzten Schwertern und einem Schild. Dies hat die Funktion einen Freund zu einem Spiel herauszufordern. Falls dieser Freund gerade ein aktives Spiel hat erscheint noch ein zweiter Button mit einem Auge als Icon, welches den Benutzer zu dem aktiven Spiel des Freundes als Zuschauer navigiert.
    - **FriendRequest:** Eine Freundschaftsanfrage wird mittels dieser Komponente dargestellt und beantwortet (siehe Abbildung 3.5).

    - **AddFriendModal:** Mit Hilfe dieser Komponente können Freundschaftsanfragen unter Angabe des Benutzernamens versendet werden. // BILD

- **Login & SignUp:** Komponenten, die das Anmelden und Registrieren mittels Formularen mit Formik und Yup (siehe Abschnitt 2.2.5) ermöglichen und mit dem Server zur Authentifizierung kommunizieren (siehe Abbildungen 3.7 & 3.8).

- **ChessGame:** Die Komponente ChessGame ist das Herzstück der Anwendung, da dort das eigentliche Schachspiel stattfindet. Die ChessGame Komponente wird durch den Pfad `/game/:roomId` gerendert und holt sich durch die roomId die Spieldaten vom Backend. In der Abbildung 3.9 ist eine beispielhafte Komponente zu sehen. Auf der rechten Seite neben dem Brett befindet sich die *ChessClock* Komponente und daneben befindet sich die *Chat* Komponente. Das Spielfeld und die Figuren entstehen durch die Bibliothek chessground, während die Spiellogik hinter dem Schachspiel von chess.js verwaltet wird (siehe Abschnitt 2.2.5). Eine detaillierte Beschreibung, was in der Komponente beim spielen oder empfangen eines Zugs passiert befindet sich im Abschnitt 3.4.3.

    - **ChessClock:** ChessClock ist eine Komponente, die die Verwaltung und Darstellung der Schachuhren übernimmt.
    - **Chat:** Die Chat Komponente repräsentiert einen simplen gehaltenen Chat in dem die beiden Spieler kommunizieren können. Zuschauer können ihn lesen, allerdings nichts selber schreiben, da sie Tipps geben könnten.

3.4.2 Authentifizierung

Die Authentifizierung findet in drei Komponenten statt: Dem *UserContext* und den *Login*- und *SignUp*-Komponenten. Gewissermaßen hat *SocketContext* auch etwas mit der Authentifizierung zu tun, da sobald sich der Anmeldezustand sich verändert eine neue socket Verbindung hergestellt wird. Was im Backend bei einer Authentifizierung stattfindet, wird in Abschnitt 3.5.1 erläutert.
Der *UserContext* beinhaltet den State `user` und der *SocketContext* beinhaltet den State `socket`, die den Komponenten zur Verfügung gestellt wird. Nach dem rendern der Komponente wird im *UserContext* eine GET HTTP Anfrage an den Server unter dem Pfad `/auth/login` gesendet. Diese beinhaltet, falls vorhanden, den Cookie mit dem auf dem Server authentifiziert werden kann. Daraufhin sendet der Server die Antwort, ob der Benutzer nun angemeldet ist oder nicht und wenn ja, dann seinen Benutzernamen. Dies wird in den `user`-State gesetzt.
Wurde der `user` gesetzt, wird im *SocketContext* eine socket Verbindung hergestellt. Solange eine der beiden States der Kontexte nicht initialisiert wurde, wird ein Lade-Bildschirm gezeigt, um fehlerhafte Darstellungen oder Funktionen zu vermeiden.
Die Authentifizierung der *Login*- und *SignUp*-Komponenten ist simpel gehalten. Mittels Formik und Yup (siehe Abschnitt 2.2.5) werden die Formulare überprüft und gegebenenfalls als POST HTTP Anfrage unter `/auth/signup` oder `/auth/login` an den Server

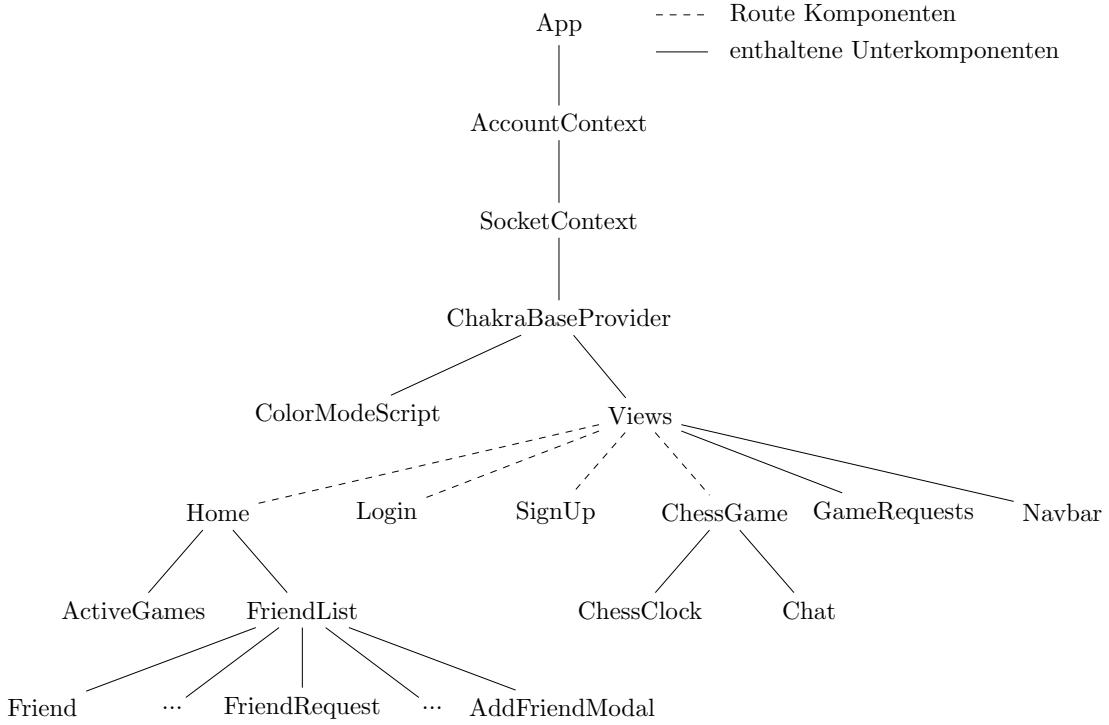


Abbildung 3.3: Aufbau der React-Komponenten für angemeldete Benutzer

gesendet. Falls es dabei auf dem Server einen Fehler gab, wird die Fehlermeldung angezeigt oder man erhält als Antwort die Benutzerdaten, welche in den *UserContext* gespeichert werden (siehe Abbildungen 3.7 & 3.8). Auch hier wird nach dem Ändern des `user` States in der *UserContext*-Komponente eine neue socket Verbindung in *SocketContext* hergestellt.
Konkrete Einzelheiten und Code-Beispiele werden in Abschnitt 4.1.1 erläutert.

3.4.3 Das Schachspiel

In diesem Kapitel werde ich näher darauf eingehen wie das Schachspiel im Frontend verwaltet und aktualisiert wird. Die Vorgehensweise im Backend und im Zusammenspiel befindet sich in Abschnitt 3.5.3.
Das Schachspiel und die zugehörigen Schachuhren sind getrennt gehalten um die Modularität zu erhöhen. Die Schachuhren und das Spiel haben jeweils eigene Events und Listener auf die sie hören.

Das Starten eines Schachspiels

Um eine Schachpartie zu starten können entweder die Buttons in der Mitte des Bildschirms der *Home*-Komponente (siehe Abbildung 3.5) oder die Herausforderung zu einer Partie eines Freundes verwendet werden.

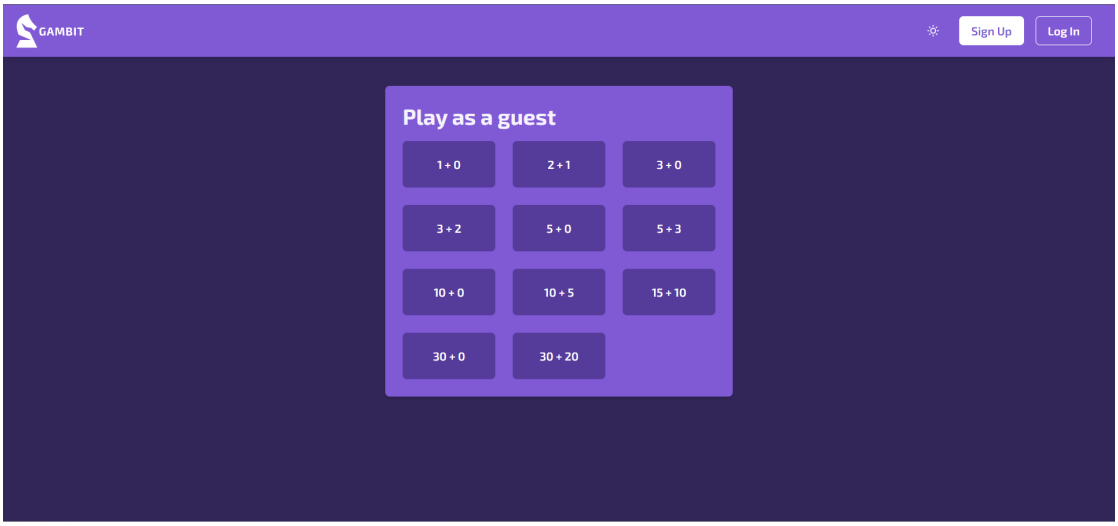


Abbildung 3.4: Home und Navbar Komponente eines nicht angemeldeten Benutzers im dunklen Farbschema

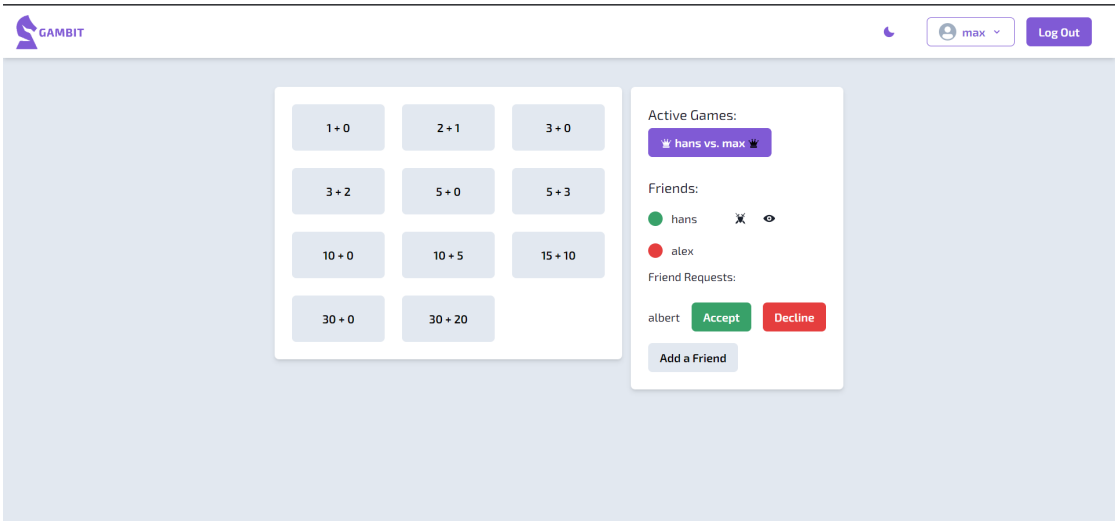


Abbildung 3.5: Home und Navbar Komponente eines angemeldeten Benutzers im hellen Farbschema

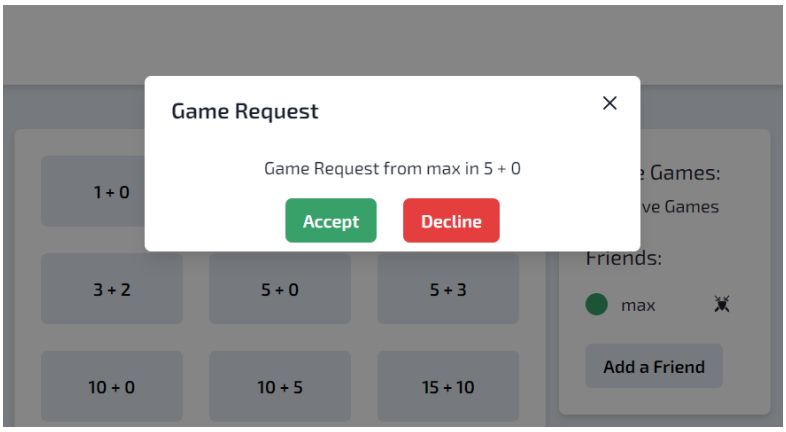


Abbildung 3.6: Das Modal der Komponente GameRequest in hellem Farbschema

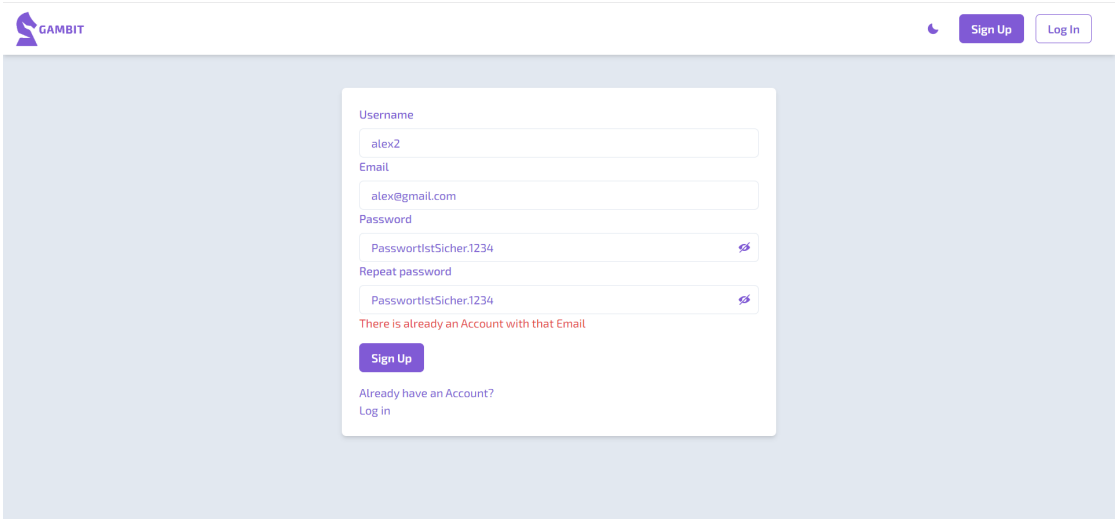


Abbildung 3.7: Beispiel einer SignUp-Komponente mit bereits verwendet E-Mail Adresse

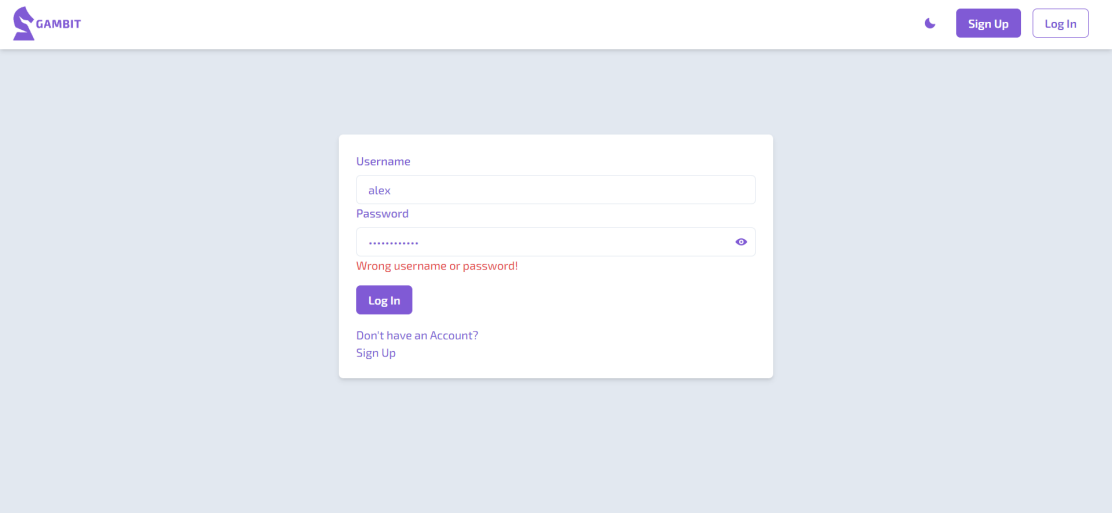


Abbildung 3.8: Beispiel einer Login-Komponente mit ungültigen Daten

Abbildung 3.9: Beispiel einer ChessGame-Komponente

Beim klicken auf eines der Zeitkonfigurations-Buttons in der *Home*-Komponente wird das Event `find_game` mit dem aktuellen Benutzerzustand der *UserContext*-Komponente und der Auswahl der Zeitkonfiguration gesendet. Solange man auf einen zufälligen Gegner wartet erscheint ein Lade-Bildschirm, mit einem Button um den Suchvorgang eines Gegners abzubrechen (siehe Abbildung 3.10). Bei solch einem Abbruch wird das Event `leave_queue` mit der Zeitkonfiguration gesendet.

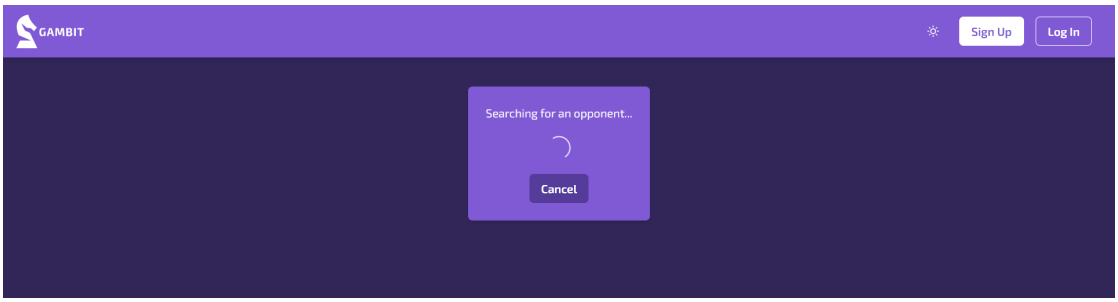


Abbildung 3.10: Lade-Bildschirm beim Starten eines Spiels mit den Buttons der *Home*-Komponente

Wurde ein Gegner für diese Zeitkonfiguration gefunden, wird in der *Home*-Komponente das Event `joined_game` mit einer ID unter der das Spiel stattfindet (In Zukunft wird die ID eines Schachspiels *roomId* genannt) und gegebenenfalls einen Gast Benutzernamen empfangen und man wird zu dem Pfad `/game/roomId` weitergeleitet, auf dem eine entsprechende *ChessGame*-Komponente das Schachspiel initialisiert.
Wie man einen Freund zu einer Partie herausfordert und was passiert wenn der Freund die Anfrage akzeptiert oder ablehnt wird in Abschnitt 3.4.4 erläutert, da dies eine Funktion ist, welche in der *Friend*-Komponente stattfindet.
Wenn man selbst eine Anfrage erhält wird dies mittels der *GameReqeust*-Komponente dargestellt (siehe Abbildung 3.6). Diese besteht aus einer Liste aller gültigen Anfragen zu einer Partie und hat Listener auf die Events `game_request` und `cancel_game_request`. Das Event `game_request` empfängt eine Anfrage zu einer Partie mit den Informationen um welche Zeitkonfiguration es sich handelt und wer einen herausfordert. Diese wird ganz einfach der Liste hinzugefügt.
Das Event `cancel_game_request` wird empfangen, sobald der Freund seine Anfrage zurück zieht. Es enthält den Benutzernamen des Spielers, der die Anfrage gesendet hat und diese Anfrage wird aus der Liste der Anfragen entfernt.
Gesendet werden kann das Event `game_request_response` mit den Informationen um welche Anfrage es sich handelt und ob man die Anfrage akzeptiert oder ablehnt.

Das Spiel

Das Schachspiel findet in der Komponente *ChessGame* statt. Nach dem rendern der Komponente wird das `get_game_data` Event mit der roomId des Spiels und einer Callback Methode gesendet, die die Daten des Spiels beinhaltet, falls dieses existiert. Falls

diese Partie nicht im Backend existiert, also keine Daten gesendet werden, wird der Benutzer darauf hingewiesen (siehe Abbildung 3.11).

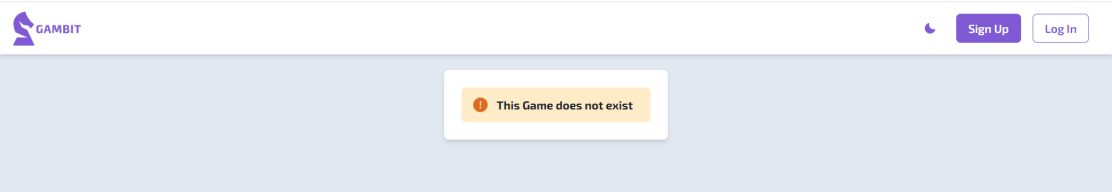


Abbildung 3.11: Benachrichtigung, falls kein Spiel unter der roomId gefunden werden konnte

Die Daten die gesendet werden umfassen folgendes:

- Der Namen des weißen und des schwarzen Spielers.
- Welcher Zeitmodus gespielt wird (z.B.: 5 + 3, 10 + 5, ...)
- Die aktuelle Stellung der Partie
- Die bisherigen Nachrichten im Chat.
- Die aktuelle Phase des Spiels: Dabei gibt es die vier Möglichkeiten, dass die Startzeit von Schwarz oder Weiß läuft oder dass die reguläre Zeit von Schwarz oder Weiß läuft.
- Die aktuellen Zeiten der Spieler.

Diese Informationen werden für den Fall benötigt, dass man die Seite neu lädt, über die `ActiveGames`-Komponente auf die Seite gelangt, über einen Link der Partie beitritt oder ähnliches. Das Einholen dieser Daten gewährleistet, dass man auch in diesen Fällen noch ohne Veränderungen weiterspielen oder zugucken kann.
Aufgrund der gesendeten Namen der Spieler wird entschieden, ob man ein Zuschauer oder ein Spieler ist. Dafür werden die Benutzernamen mit dem eigenen Benutzernamen im *UserContext* verglichen. Doch was passiert wenn man eine Partie als unangemeldeter Benutzer spielt und deshalb keinen Benutzernamen im *UserContext* hat?
Bei dem Event `joined_game` (siehe Abschnitt 3.4.3) wird der Gast-Benutzername des Spielers gesendet und in `location.state` gesetzt[6]. Dadurch kann in der *ChessGame*-Komponente darauf zugegriffen werden und es kann überprüft werden, ob es sich um einen Spieler oder Zuschauer handelt.
Dem entsprechend wird auch bestimmt wie das Schachbrett, die Namen und die Schachuhren ausgerichtet sind. Ist man Zuschauer wird in chessground definiert, dass man keine Figuren bewegen kann und es gibt kein input Feld für den Chat, sodass man keine Nachrichten abschicken kann. Dies verhindert, dass ein Zuschauer Tipps geben könnte.
Zum Spielen der Partie werden folgende Listener definiert:

[6]Quelle: `https://github.com/remix-run/history/blob/main/docs/api-reference.md#locationstate` am 04. Mai 2023

- `opponent_move`: Dient zu Empfangen eines Zugs eines Spielers.
- `checkmate`: Ein Event welches bei Schachmatt mit dem Benutzernamen des Gewinners empfangen wird.
- `time_over`: Ist eine Benachrichtigung, dass die Zeit eines Spielers abgelaufen ist.
- `draw`: Kommuniziert ein Patt der Partie.
- `resigned`: Signalisiert, dass ein Spieler aufgegeben hat.
- `cancel_game`: Das Spiel wird aufgrund der abgelaufenen Start Zeit abgebrochen.

Die Events `checkmate`, `time_over`, `draw`, `resigned` und `cancel_game` beschreiben alle das Ende des Schachspiels. In ihren Listenern wird definiert, dass man keine Figur des Schach Interfaces von chessground mehr bewegen darf und man wird über den Ausgang des Spiels in Form von einem Toast benachrichtigt (siehe Abbildung 3.12).

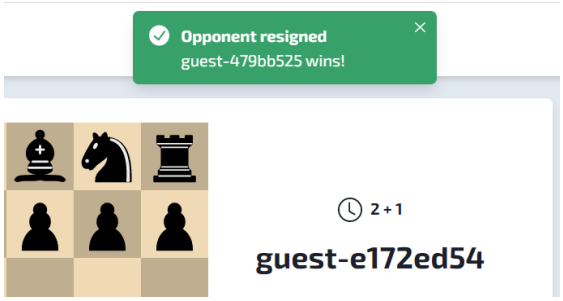


Abbildung 3.12: Beispiel eines Toasts, falls der Gegner aufgegeben hat

Der Ablauf beim Empfangen eines neuen Zugs ist im Aktivitätsdiagramm in Abbildung 3.14 dargestellt. Die Bauernumwandlung und das En Passant müssen separat behandelt werden, da chessground nur das Schach Interface zur Verfügung stellt und bei diesen beiden Zusatzregeln andere Figuren ersetzt oder entfernt werden, als bei regulären Zügen. Das aktualisieren möglicher Züge beinhaltet, dass chessground alle möglichen Züge von chess.js übertragen bekommt, welches zur Folge hat, dass bei einem Klick auf eine Figur korrekt angezeigt wird wohin diese Figur ziehen könnte und auch nur auf diese Felder kann eine Figur bewegt werden (siehe Abbildung 3.9).
Gesendet werden können die Events: `new_move` zum senden eines Zugs, `resign` zum Aufgaben der Partie und `leave_room`, wenn der Spieler die *ChessGame* Komponente verlässt. Ein Aktivitätsdiagramm des Senden eines Zugs befindet sich in Abbildung 3.13. Genau wie bei dem Empfangen eines Zugs wird auch beim Senden zwischen Bauernumwandlung und En Passant unterschieden. Der einzige Unterschied ist, dass bei der Bauernumwandlung nach dem setzen des Zugs noch mittels der *PromotionModal*-Komponente ausgewählt werden muss, in welche Figur sich der Bauer umwandeln soll, bevor der Zug gesendet wird.
Das Event zum Aufgeben wird nach dem klicken des „resign“ Buttons gesendet.

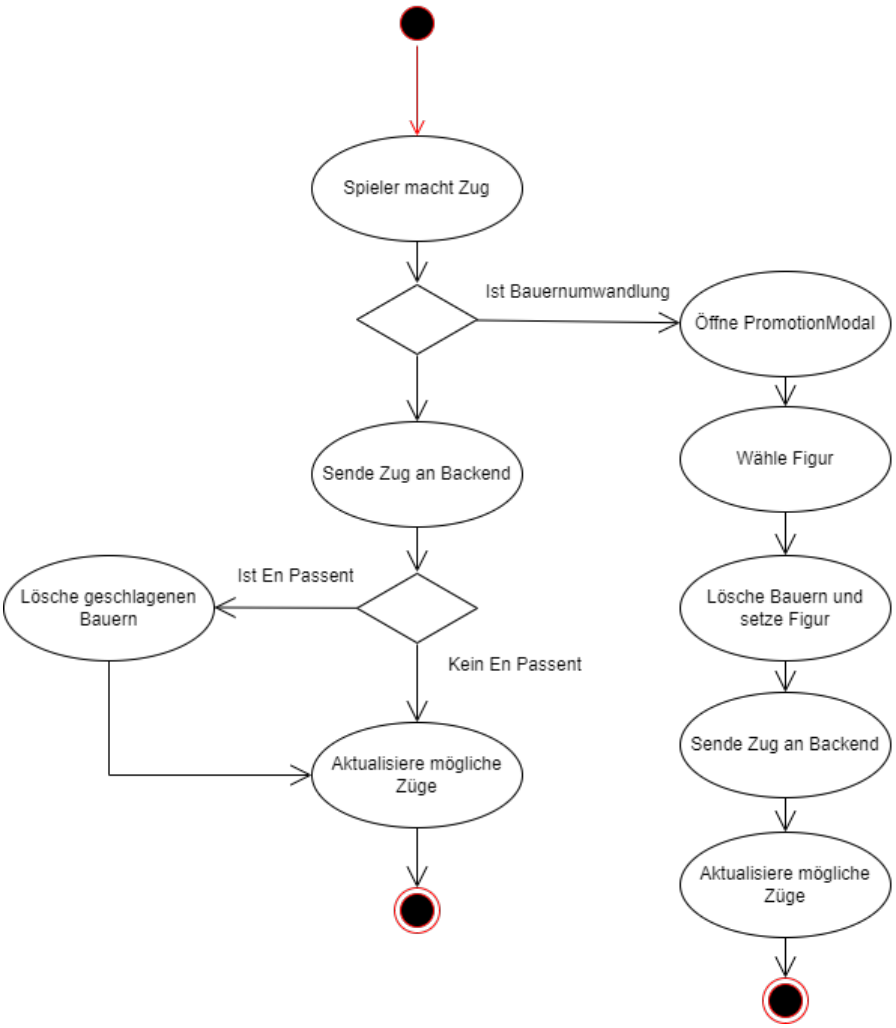


Abbildung 3.13: Aktivitätsdiagramm eines Schachzugs

Die Uhr

Die *ChessClock* Komponente bekommt von *ChessGame* als Props die aktuelle Phase der Partie, die jeweiligen aktuellen Zeiten und die Ausrichtung, welche Zeit oben bzw. unten gezeigt werden soll. Neben der regulären Schach Zeit gibt es noch eine Startzeit, welche während des ersten Zugs jeden Spielers abläuft, um zu gewährleisten, dass das Spiel auch erst wirklich startet sobald beide Spieler bereit sind. Läuft die Startzeit ab wird das Spiel abgebrochen (siehe Abschnitt 2.1). Diese Startzeit ist die lila eingefärbte Zeit in Abbildung 3.9.
Die Komponente sendet keine Events, sondern hört nur auf die folgenden:

- `updated_time`: Dieses Event wird vom Backend gesendet, sobald ein Zug gemacht wurde und enthält die aktuellen Zeiten der Spieler nach dem Zug und welcher

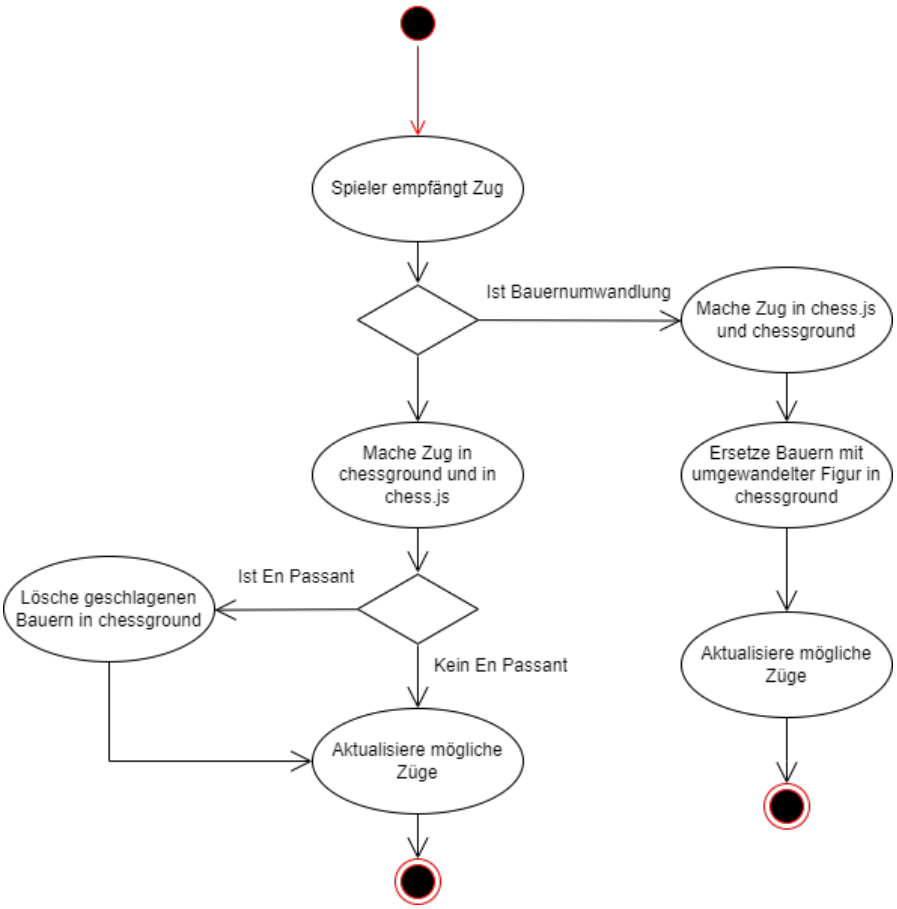


Abbildung 3.14: Aktivitätsdiagramm eines empfangenen Schachzugs

Spieler jetzt am Zug ist. Dementsprechend werden die Zeiten aktualisiert und die Uhr des Spielers, welcher jetzt dran ist wird gestartet.

- `stop_starting_time_white`: Stoppt die Start Zeit des weißen Spielers und startet die Start Zeit des schwarzen Spielers.
- `stop_starting_time_black`: Stoppt die Start Zeit des schwarzen Spielers und lässt die reguläre Zeit des weißen Spielers beginnen.
- `stop_clocks`: Wird bei Beendung des Spiels empfangen und stoppt die aktive Uhr.

Der Chat

Die *Chat*-Komponente bekommt als props alle bisher gesendeten Nachrichten, die roomId unter der das Spiel stattfindet, die Information ob es sich um einen Zuschauer handelt und gegebenenfalls den Gastnamen, falls es sich um einen nicht angemeldeten Benutzer handelt. Handelt es sich um einen Zuschauer wird das Input-Feld der Komponente nicht

angezeigt, um keine Nachrichten schreiben zu können. Die Komponente ist relativ simpel und sendet das Event `send_message`, um eine Nachricht zu senden und empfängt eine Nachricht mit dem `message` Event. Eine Nachricht beinhaltet immer den Namen des Versenders, die Nachricht als String und die roomId des Spiels. Je nachdem ob der Versender der Nachricht mit dem eigenen Namen übereinstimmt wird die Nachricht unterschiedlich ausgerichtet und eingefärbt (siehe Abbildung 3.9).

3.4.4 Verwaltung von Freunden

Die Verwaltung und Darstellung (siehe Abbildung 3.5) von Freunden und Freundschaftsanfragen obliegt der *FriendList*-Komponente. Diese Beinhaltet die Unterkomponenten *Friend*, *FriendReqeust* und *AddFriendModal*.

FriendList-Komponente

FriendList verwendet zwei States in Form von Arrays: `friends` und `friendRequests`. Je ein Element dieser Listen wird durch eine *Friend*-, beziehungsweise *FriendRequest*-Komponente dargestellt und verwaltet. Dabei werden die Freunde je nachdem ob sie online sind oder nicht sortiert. *FriendList* hört dabei auf die folgenden Events:

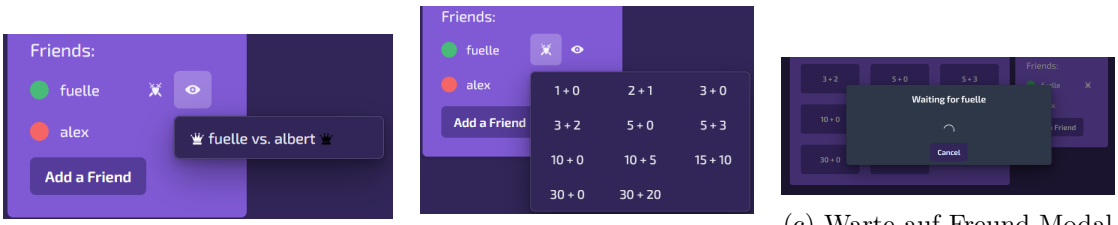
- **friends:** Ein Event welches nach der Anmeldung des Nutzers empfangen wird, um die Liste der Freunde zu bekommen.
- **friend_requests:** Das gleiche Event, nur zum setzen der Freundschaftsanfragen.
- **friend_request_accepted:** Enthält Daten eines neuen Freundes, welcher deine Freundschaftsanfrage angenommen hat. Dieser wird der Freundesliste hinzugefügt.
- **friend_request:** Eingang einer neuen Freundschaftsanfrage. Wird der Liste der Freundschaftsanfragen hinzugefügt.
- **connected:** Dieses Event wird empfangen, falls ein Freund von dir offline, beziehungsweise online geht. Der betreffende Freundes-Eintrag in der Freundesliste wird aktualisiert.

Des weiteren werden die zwei Events `get_friends` und `get_friend_requests` jedes Mal gesendet, falls auf die `Home`-Komponente navigiert wird. Diese beiden Events empfangen mittels Callback-Funktionen alle Daten über die Freunde und Freundschaftsanfragen. Dies ist nötig, damit, falls beispielsweise nach einer Schachpartie wieder auf die *Home*-Komponente navigiert wird, die Daten der Freunde aktualisiert werden.

Friend-Komponente

Diese Komponente stellt einen Freund dar. Es kriegt als props alle wichtigen Daten des Freundes gestellt. Sie hat zwei Grundlegende Funktionen: Das Zuschauen einer Partie eines Freundes und das Herausfordern zu einer Partie. Beim Herausfordern eines Freundes sind genau die gleichen Schachuhren möglich, wie bei der Suche eines zufälligen

Gegners und das Zuschauen ist Aufgebaut wie bei der *ActiveGames*-Komponente. (siehe Abbildung 3.15)
Diese beiden Funktionen sind nur verfügbar, falls der Freund gerade online ist, welches über einen grünen Punkt ersichtlich ist und zum Zuschauen benötigt der Freund trivialerweise ein aktives Spiel.
Die Komponente hat zwei Eventlistener: `game_request_accepted` und `game_request_denied`, welche einen Toast darstellen und den Spieler gegebenenfalls zu dem Spiel navigieren.
Senden tut die Komponente zwei Events: `send_game_request` und `cancel_game_request`.
Wurde eine Herausforderung zu einer Partie versendet erscheint ein Lade-Bildschirm, bis der Spieler die Anfrage angenommen, beziehungsweise abgelehnt hat. Entscheidet sich der Spieler davor doch nicht mehr gegen den Freund zu spielen, kann er auf den „Cancel“ Button klicken und die Spielanfrage wird zurückgenommen.



(a) Aktive Partien eines Freundes

(b) Das Herausfordern eines Freundes

(c) Warte auf Freund Modal

Abbildung 3.15: Das Herausfordern und Zuschauen bei Freunden

FriendRequest-Komponente

Die *FriendRequest*-Komponente stellt eine Freundschaftsanfrage dar und erhält ebenfalls seine Daten von der *FriendList*-Komponente, wozu auch die Funktionen `setFriends` und `setFriendRequest` zählen, um die Listen der *FriendList*-Komponente zu ändern. Es hört auf keine Events, sendet allerdings zwei Events: `accept_friend_request` und `decline_friend_request`. Die Behandlung dieser Events im Backend wird in Abschnitt 3.5.2 erläutert. War das Akzeptieren, beziehungsweise das Ablehnen der Anfrage erfolgreich wird die Freundschaftsanfrage aus der Liste gelöscht und zusätzlich wird gegebenenfalls der neue Freund der Freundesliste hinzugefügt.

AddFriendModal-Komponente

Diese Komponente besteht aus einem Button und einem Modal, falls man den Button anklickt. In diesem Modal kann man einen Benutzernamen angeben, an den die Freundschaftsanfrage verschickt werden soll. Das Versenden einer Freundschaftsanfrage wird mittels des `send_friend_request` Events behandelt. Es beinhaltet eine Callback-Funktion, welche entgegennimmt, ob die Versendung erfolgreich war und wenn nicht, dann eine Fehlermeldung, warum es nicht möglich war die Freundschaftsanfrage zu ver-

senden (siehe Abbildung 3.16). Wie das Backend eine neue Freundschaftsanfrage behandelt wird in Abschnitt 3.5.2 erläutert.

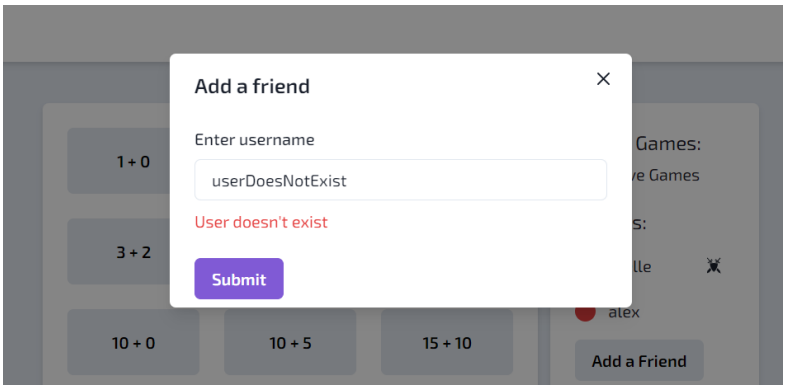


Abbildung 3.16: Das Modal der *AddFriendModal*-Komponente mit Fehlermeldung

3.4.5 Anzeigen und navigieren zu aktiven Partien

Falls man eine aktive Schachpartie spielt und aus Versehen den Browser schließt, auf die Startseite navigiert oder ähnliches gäbe es keine Möglichkeit auf das Spiel zurück zu kehren, es sei denn man hat den Link oder die roomId des Spiels gespeichert. Um es möglich zu machen auf aktive Spiele schnell wieder zurück zu kehren, ohne die roomId der Partie zu kennen, gibt es die Unterkomponente *ActiveGames* in der *Home*-Komponente (siehe Abbildung 3.5). Diese Komponente macht es möglich zu verfolgen welche offenen Spiele man gegen wen in welcher Farbe hat und man kann leicht wieder zu diesen Spielen navigieren, indem man auf den Button klickt. Es besitzt den Listener auf das Event `active_games`, welches nach einer erfolgreichen socket Authentifizierung (siehe Abschnitt 3.5.1) mit einer Liste aller aktiven Spiele gesendet wird. Des weiteren sendet es das Event `get_active_games`, welches mit einer Callback Funktion ebenfalls so eine Liste empfängt. Dieses Event wird, wie bei dem Event `get_friends` von der *FriendList*-Komponente, gesendet, falls wieder auf die *Home*-Komponente navigiert wird, um die Liste der aktiven Spiele zu aktualisieren. Ist die Liste leer, wird nur „No acitve Games“ angezeigt. Ansonsten wird jedes aktive Spiel der Liste mit einem Button dargestellt, auf deren klick man zu dem Spiel navigiert wird.

3.5 Backend-Architektur

Das Backend basiert auf Node.js mit dem Express Framework. Des weiteren werden als Schnittstellen mit dem Frontend eine Web-API für HTTP Anfragen und ein Socket.io Server bereitgestellt. Das Backend kommuniziert mit zwei Datenbanken: einer PostgreSQL Datenbank für die Benutzerverwaltung und eine Redis Datenbank für häufig aktualisierte und angefragte Daten.
Die Ordnerstruktur des Backends (Abbildung 3.17) ist auf dieser Weise aufgebaut:

- **auth:** Dieser Ordner beschäftigt sich sowohl mit der Schnittstelle der Web-API-Kommunikation mit dem Frontend, als auch deren Behandlung und dem Austausch mit der PostgreSQL Datenbank. Diese Schnittstellen dienen bloß der Anmeldung und Registrierung eines Benutzers.
- **chess:** Stellt ein chess.js Schachspiel und die serverseitige Schachuhr zur Verfügung.
- **redis:** Dient als Schnittstelle und Verwalter von Operationen auf der redis Datenbank.
- **sockets:** Stellt Middleware für die Verbindungsherstellung und Listener für die Kommunikation zwischen Frontend und Backend bereit.
- **index.js:** Initialisiert den Server mit seinen Schnittstellen.

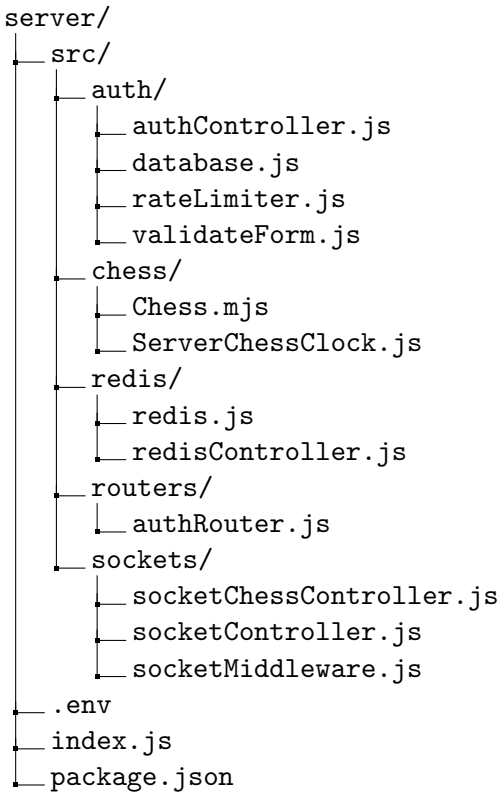


Abbildung 3.17: Ordnerstruktur des Backends

3.5.1 Authentifizierung

Die Authentifizierung eines Benutzers läuft über HTTP Anfragen an die Web-API und einer PostgreSQL Datenbank. Nachdem die erste Authentifizierung stattgefunden hat

wird die Socket.io Verbindung des Clients hergestellt, in der die Socket Verbindung nochmals Authentifiziert wird. Es gibt drei verschiedene Möglichkeiten wie ein Benutzer authentifiziert werden kann: Durch das Anmelden, durch das Registrieren oder durch das Lesen des Cookies beim Aufruf der Seite vom Client.
Die Authentifizierung im Frontend abläuft wird in Abschnitt 3.4.2 erläutert.

Authentifizierung mit der Web-API und PostgreSQL

Das Anmelden und Registrieren mittels Formular läuft über eine POST Anfrage des Clients an den Pfad `/auth/login`, beziehungsweise `/auth/signup`, die die angegebenen Formulardaten beinhaltet. Bei der Verarbeitung der Anfrage werden mittels des Express Routing verschiedene Middlewares verwendet.
Eine Middleware stellt sicher, dass die Anzahl der Anfragen über eine IP-Adresse in einer bestimmten Zeit begrenzt wird. Dies verhindert sogenannte Denial-of-Service (kurz: DoS) Attacken[7], bei denen probiert wird den Server mit so vielen Anfragen zu belasten, dass dieser außer Betrieb gesetzt wird.
Anschließend überprüft eine Middleware, ob die angegebenen Daten mit dem Schema übereinstimmen.
Treten bei diesen beiden Middlewares keine Fehler auf wird beim Anmelden überprüft, ob dieser Benutzer in der Datenbank existiert und es wird mittels bcrypt überprüft ob die Passwörter übereinstimmen. Ist dies der Fall, wird ein JWT-Token mit den Benutzerinformationen erstellt und als Cookie in den Browser des Clients gesetzt. Des weiteren wird dem Benutzer geantwortet, dass die Anmeldung erfolgreich war mit der Übermittlung des Benutzernamens.
Beim Registrieren wird überprüft, ob bereits ein Nutzer mit dem Benutzernamen oder E-Mail existiert und anschließend wird ein neuer Tupel in der PostgreSQL Datenbank erstellt. Das Passwort wird dafür mittels bcrypt verschlüsselt und es wird eine individuelle userid generiert. Diese dient zur Socket.io-Kommunikation.
Bei dem ersten Aufruf der Seite vom Client wird eine GET Anfrage an `/auth/login` gestellt. Bei dieser wird ebenfalls die Middleware gegen DoS-Attacken verwendet und anschließend wird überprüft, ob er einen gültigen JWT-Token im Cookie hat und ihm wird dem entsprechend geantwortet. Das setzen des Tokens im Cookie hat den Vorteil, dass bei einem neuen Aufruf der Seite, solange der Cookie noch gültig ist, der Benutzer automatisch angemeldet wird, ohne seine Anmeldedaten nochmals einzugeben.
Anschließend stellt der Client eine Socket.io Verbindung her.

Authentifizierung und anschließende Middleware mit Socket.io

Bei der Verbindungsherstellung des Clients mit dem Socket.io Server durchläuft die socket verschiedene Middlewares.

[7] Quelle: `https://www.bsi.bund.de/DE/Themen/Verbraucherinnen-und-Verbraucher/Cyber-Sicherheitslage/Methoden-der-Cyber-Kriminalitaet/DoS-Denial-of-Service/dos-denial-of-service_node.html` am 27. April 2023

Die erste Middleware Authentifiziert die Socket des Benutzers. Sie liest aus dem Cookie, der auch in der socket mitgesendet wird, den JWT-Token, falls dieser existiert. Die Daten die in dem Token kodiert sind werden dann in der Socket als Attribute gesetzt, sodass anschließend immer mittels `socket.user` darauf zugegriffen werden kann.
Wenn der Benutzer keinen gültigen JWT-Token hat, werden trotzdem alle Middlewares ohne Fehler durchlaufen. Dies liegt daran, dass bei einem fehlgeschlagenen Middleware-Prozess die Socket.io-Verbindung abgelehnt werden würde. Es soll allerdings auch das Spielen einer Schachpartie als Gast möglich sein.
Als zweite Middleware wird der Benutzer mit Daten versorgt, er tritt dem Raum seiner userid bei und wird als online vermerkt. Der beitritt in den Raum seiner userid dient dazu, dass man anschließend nur diese ID braucht um ein Event an den Benutzer zu senden. Dabei wird in Redis `user:username` mit seinen Informationen und `connected` auf `true` gesetzt. Anschließend werden alle Freunde des Benutzers aus der Redis Datenbank geholt und an alle das Event `connected` gesendet, das signalisiert, dass der Benutzer online ist.
Beim Schließen der Anwendung oder Abmelden des Benutzers wird dementsprechend in Redis `connected` auf `false` gesetzt und die Freunde werden mit dem Event `connected` darüber in Kenntnis gesetzt, dass der Benutzer nicht mehr online ist.
Beim Authentifizieren werden auch Informationen bezüglich aktiver Spiele und Freundschaftsanfragen aus Redis abgerufen und mittels den Events `friends`, `friend_requests` und `active_games` werden die Informationen über Freunde, Freundschaftsanfragen und aktive Partien an den Benutzer gesendet.
Als letzte Middleware werden alle nötigen Listener sowohl für das Schachspiel, als auch für sonstige Funktionen initialisiert.

3.5.2 Hinzufügen von Freunden

In den Aktivitätsdiagrammen in Abbildung 3.18 ist der Ablauf des Versenden und Akzeptieren einer Freundschaftsanfrage abgebildet. Die verschiedenen Datentypen der Speicherung in Redis befindet sich in Abschnitt 3.6.2.

Versenden einer Freundschaftsanfrage

Beim Versenden einer Freundschaftsanfrage wird vom Frontend das Event `send_friend_request` mit dem angegebenen Benutzernamen versendet. Um zu überprüfen, ob eine Freundschaft der beiden erlaubt ist, wird kontrolliert, ob es der eigene Benutzername ist, ob der Nutzer nicht existiert und ob die beiden Benutzer schon befreundet sind. Ist eines davon der Fall, wird mit einer entsprechenden Fehlernachricht dem Sender mittels einer Callback Funktion geantwortet. Ist die Freundschaft erlaubt wird zusätzlich noch überprüft ob es noch eine offene Freundschaftsanfrage zwischen den beiden gibt und falls dies der Fall wird, wird der Benutzer ebenfalls darauf hingewiesen. Ansonsten wird die Freundschaftsanfrage in Redis gespeichert und ein Event mit der Freundschaftsanfrage wird an den betreffenden Spieler gesendet und der Sender wird darüber informiert, dass die Freundschaftsanfrage versendet wurde.

Akzeptieren und Ablehnen einer Freundschaftsanfrage

Beim Akzeptieren einer Freundschaftsanfrage wird wie bei dem Versenden nochmals überprüft, ob diese Freundschaft erlaubt ist. Anschließend wird die Freundschaftsanfrage gelöscht, die beiden Benutzer werden in die entsprechende Freundesliste gesetzt, die restlichen Daten der beiden Spieler werden eingeholt und an den jeweils anderen Benutzer werden alle Informationen, wie ob er/sie online ist oder ob er/sie aktive Spiele hat, gesendet. Dies stellt sicher, dass die Freunde im Frontend direkt richtig angezeigt werden können.
Beim Ablehnen einer Freundschaftsanfrage wird diese einfach nur aus Redis gelöscht.

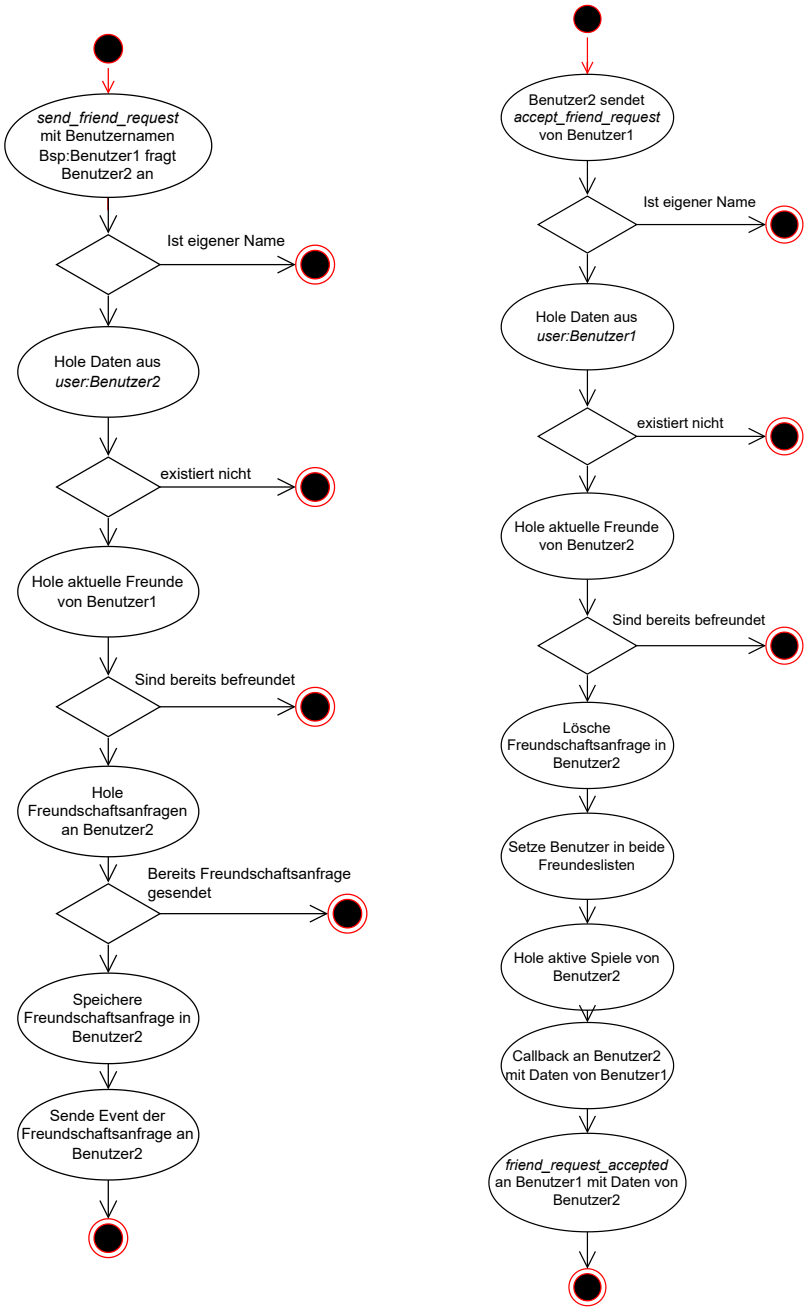
3.5.3 Das Schachspiel

In diesem Abschnitt erkläre ich, was genau beim Finden eines Gegners, Initialisieren, bei neuen Zügen und neuen Nachrichten im Chat eines Schachspiels passiert.

Finden eines Gegners

Es gibt 3 verschiedene Arten ein Schachspiel zu starten: Ein angemeldeter Benutzer sucht einen zufälligen Gegner, ein angemeldeter Benutzer spielt ein Spiel gegen einen Freund und ein unangemeldeter Benutzer spielt gegen einen zufälligen Gegner.
Das Suchen eines Spiels mit zufälligem Gegner wird mittels des Events `find_game` mit den Benutzerdaten und der gewählten Zeitkonfiguration vom Frontend gesendet. Ein Sequenzdiagramm des starten eines Spiels mit zufälligem Gegner befindet sich in Abbildung 3.19. Dies beinhaltet folgende Schritte:

- Das Event `find_game` mit der Zeitkonfiguration und Benutzerdaten wird gesendet (siehe Abschnitt 3.4.3).
- Falls der Benutzer nicht angemeldet ist, wird ihm ein zufälliger username für dieses Schachspiel zugewiesen, der mit „guest-“ startet. Seine userid wird als seine socket id festgelegt.
- Daraufhin wird im Server ein Spieler aus der entsprechenden Warteschlange aus Redis genommen (siehe Abschnitt 3.6.2).
- Falls dabei kein Spieler entnommen werden konnte, wird der Benutzer selbst in die Liste geschrieben und wartet bis er von einem anderen Benutzer aus der Liste genommen wird.
- Falls ein Spieler aus der Liste entnommen werden konnte, wird das Spiel mit einer *roomId* als Identifikator initialisiert und in Redis gespeichert.
- An die beiden Spieler wird das Event `joined_game` mit der roomId und gegebenenfalls dem Gast-Benutzernamen gesendet, woraufhin sie zu dem Pfad `/game/roomId` navigieren, auf der sich die *ChessGame*-Komponente befindet.



(a) Senden einer Freundschaftsanfrage

(b) Akzeptieren einer Freundschaftsanfrage

Abbildung 3.18: Aktivitätsdiagramme des Versenden und Akzeptieren von Freundschaftsanfragen

- Die *ChessGame*-Komponente sendet das `get_game_data` Event. Daraufhin wird der aktuelle Zustands der Partie aus Redis geholt und an das Frontend zurück gesendet. Ein Ablauf was im Frontend bei einer Schachpartie passiert befindet sich im Abschnitt 3.4.3.

Falls es sich um eine Partie handelt, die aus einer Herausforderung eines Freundes resultiert, wird natürlich in keine Warteschlange nach einem Gegner gesucht, anstatt dessen wird mit den Events `send_game_request`, `game_request` und `game_request_response` (siehe Abschnitt 3.4.4, 3.4.3) die Anfrage versendet und beantwortet. Anschließend wird das Spiel initialisiert und an die beiden Spieler das Event `game_request_accepted` mit der roomId gesendet.

Initialisieren eines Spiels nach gefundenem Gegner

Zunächst wird per Zufall entschieden welcher Spieler weiß und welcher Spieler schwarz spielt. Anschließend wird ein neues Schachspiel der Klasse chess.js (siehe Abschnitt 2.2.5) erstellt und in ihr festgehalten welcher Spieler weiß, beziehungsweise schwarz, spielt und das Datum an dem dieses Spiel stattfindet. Des weiteren wird eine zufällige roomId generiert, unter der das Schachspiel stattfindet.
Der weitere Ablauf der Initialisierung wird im Sequenzdiagramm in Abbildung 3.20 dargestellt.
Dem *redisController* wird aufgetragen das Spiel in Redis zu initialisieren. Dafür bekommt er die Informationen darüber welcher Spieler welche Farbe spielt, unter welcher roomId das Spiel stattfindet und die bisherige PGN-Notation aus dem chess.js Objekt. Diese Informationen speichert er in Redis unter `game:roomId` (siehe Abschnitt 3.6.2) und falls die spieler angemeldet sind wird die roomId auch der Liste `activeGames:username` bei beiden Spielern hinzugefügt.
Anschließend wird ein Objekt der Klasse ServerChessClock erstellt und die Startzeit des weißen Spielers gestartet. Das ServerChessClock Objekt wird in einem Array in der socketChessController Datei gespeichert, sodass mit der roomId darauf zugegriffen werden kann. Das Objekt ServerChessClock kommuniziert mittels Events mit socketChessListener und es werden folgende Eventlistener in socketChessListener definiert:

- `cancel_game`: Das Spiel wird aufgrund einer abgelaufenen Startzeit abgebrochen.
- `time_over_white`: Die Zeit des weißen Spielers ist abgelaufen.
- `time_over_black`: Die Zeit des schwarzen Spielers ist abgelaufen.

Dies sind alles Ausgänge von Schachpartien, die an alle sockets im Raum weitergeleitet werden. Daraufhin wird das Schachspiel beendet (siehe Abschnitt 3.5.3).

ServerChessClock

Die Klasse ServerChessClock definiert einzelne Schachuhr Objekte, welche die Startzeiten und die regulären Schachuhren verwalten.

socketChessController und ServerChessClock kommunizieren mittels Funktionen und Events miteinander, wobei Events genutzt werden um laufende Uhren anzuhalten und zu kommunizieren, dass eine Startzeit oder eine reguläre Schachuhr abgelaufen ist, während die Funktionsaufrufe von ServerChessClock dazu dienen eine bestimmte Zeit zu starten.

Neuer Zug einer Schachpartie

Ein Aktivitätsdiagramm zur Behandlung eines neuen Zugs im Backend ist in Abbildung 3.21 zu finden.
Sobald ein neuer Zug eines Spielers mit dem Event `new_move` ankommt wird die ServerChessClock aus dem Array und das aktuelle PGN aus der Redis Datenbank mittels der roomId geholt. Es wird ein neues chess.js Objekt kreiert, welches das aktuelle PGN importiert und anschließend den neuen Zug macht. Daraufhin wird der Zug in die roomId gebroadcastet. Das bedeutet, es wird an alle sockets im Raum gesendet, außer von dem Sender, da dieser ja bereits den Zug gemacht hat.
Nachdem der Zug gesendet wurde, wird überprüft ob es sich um ein Schachmatt oder ein Patt handelt, falls ja werden die Uhren der ServerChessClock angehalten, das Frontend wird über den Ausgang informiert und das Spiel wird beendet (siehe Abschnitt 3.5.3).
Falls dies nicht der Fall ist wird die Zeit des jetzigen Spielers gestoppt und falls es eine reguläre Zeit ist, wird das Inkrement auf die Zeit gerechnet, die Zeit des anderen Spielers fängt an zu laufen und die aktualisierten Zeiten werden an das Frontend gesendet.
Handelt es sich um die ersten zwei Züge, in denen die Startzeit und nicht die reguläre Zeit der Spieler läuft werden diese gesondert betrachtet und entweder die Startzeit von Schwarz beginnt zu laufen oder die reguläre Zeit von Weiß fängt an zu laufen. Dies wird auch mit entsprechenden Events dem Frontend mitgeteilt.
Als letztes wird noch die PGN-Notation mit dem neuen Zug in Redis gespeichert.

Chat

Der Listener auf das Event `send_message` empfängt und verwaltet die neue Nachricht, die ein Spieler gesendet hat. Die Informationen Benutzernamen, roomId und die Nachricht werden dabei empfangen und weitergeleitet.
Dabei passieren zwei Prozesse:

- Die Nachricht wird in Redis gespeichert. Dafür werden alle alte Nachrichten des Spiels aus `game:roomId` geholt, die neue Nachricht angehängt und anschließend wieder dort gespeichert.
- Die Nachricht wird an alle im Raum mittels des Events `message` gesendet.

Ende einer Schachpartie

Das Ende einer Schachpartie kann durch folgende Situationen stattfinden: Schachmatt, Patt, Startzeit ist abgelaufen, eine reguläre Zeit ist abgelaufen oder ein Spieler hat aufgegeben.

Auf Schachmatt und Patt wird bei neuen Zügen geachtet und an das Frontend gesendet. Wenn das Ende der Partie auf die Schachuhren zurückzuführen ist, wird ein Event von ServerChessClock in socketChessController empfangen und an die Sockets im Raum weitergeleitet.
Bei einer Aufgabe empfängt das Backend das event `resign` mit der Farbe welche aufgibt und der roomId und leitet dies entsprechend weiter.
Bei jedem dieser Ausgänge einer Partie geschieht noch folgendes:
Das ServerChessClock Objekt wird aus dem Array in socketChessController gelöscht.
Die Benutzernamen werden aus dem Redis Eintrag `game:roomId` geholt und falls es sich um angemeldete Benutzer handelt wird das entsprechende `activeGames:username` Element aus der Liste gelöscht. Anschließend wird der Eintrag `game:roomId` gelöscht.
Somit wird kein Speicher mehr von alten Spielen belegt.

3.6 Datenbankstruktur

3.6.1 PostgreSQL Datenbank

Die PostgreSQL Datenbank wird ausschließlich für die Anmeldung und Registrierung genutzt. Sie enthält eine Tabelle mit folgendem Schema:

- **id:** Eine Fortlaufende id, die als Primärschlüssel dient.
- **email:** Die E-Mail, die bei der Registrierung angegeben wurde.
- **username:** Der Benutzername des Benutzers.
- **userid:** Jeder Spieler erhält beim Registrieren seine eigene userid. Diese dient der Kommunikation mit dem Benutzer. Bei einer Verbindung mit Socket.io erhält die socket immer eine andere ID, also wie sendet man ein Event an einen bestimmten Spieler? Diese userid löst das Problem, indem man nach dem anmelden immer dieser ID als Raum beitritt. Somit kann immer an diese ID gesendet werden und man stellt sicher, dass der Client das Event empfängt.
- **password:** Das mit bcrypt verschlüsselte Passwort des Benutzers.

Jedes dieser Attribute, außer das Passwort, hat die Einschränkung, dass es einzigartig sein muss. Die E-Mail wird bisher nicht genutzt, kann aber in Zukunft zum bestätigen der Registrierung oder Einrichtung eines Newsletters genutzt werden.

3.6.2 Redis

Bemerkung: Bei beispielsweise `user:username` oder `username:userid` wird der `username` immer mit dem richtigen Benutzernamen ausgetauscht. Also die konkreten Zuweisung wäre somit `user:Max` oder `Max:18b06c86-999b-400d-84ce-2bc612b3825d`

user:username

Unter dem Key `user:username` befindet sich ein Redis Hash. Ein Redis Hash besitzt Key-Value Paare, auf welche man zugreifen kann.
In unserem Fall werden folgende Values zu den Keys dort gespeichert:

- **userid:** Auch hier wird die userid gespeichert, da Redis vor allem für socket.io Funktionen verwendet wird und daher eine kurze Abfragezeit benötigt.
- **connected:** Ist „true“ oder „false“, je nachdem ob der Benutzer gerade online ist oder nicht.

game:roomId

Der Redis Hash `game:roomId` verwaltet die Daten einer Schachpartie. Dazu gehören:

- **whitePlayer, blackPlayer:** Benutzernamen des weißen und schwarzen Spielers.
- **time:** Der Zeitmodus welcher gespielt wird (z.B.: 15 + 10, 5 + 3, ...)
- **pgn:** Die Historie aller bisherigen Züge im PGN Format.
- **chat:** Alle bisher geschriebenen Nachrichten im Chat.

activeGames:username

`activeGames:username` beinhaltet eine Liste der aktiven Spiele dieses Benutzers. In der Liste stehen alle roomIds der aktiven Spiele.

friends:username

Der Key `friends:username` verweist auf eine Liste von Freunden. Diese Freunde bestehen aus einem String, der sich zusammensetzt aus `username:userid`. Dies verhindert, dass wenn man ein Event an einen Freund senden will, man einen extra Zugriff auf `user:username` machen muss um die userid zu bekommen.

friend_requests:username

Dies ist eine Liste die genauso aufgebaut ist wie die `friends:username` Liste, außer, dass sie Freundschaftsanfragen verwaltet.

Warteschlangen

Die letzten Elemente sind die Warteschlangen für Spiele. Diese bestehen aus `waitingPlayers:timeMode` und `waitingGuests:timeMode`, je nachdem, ob es sich um einen angemeldeten Benutzer handelt oder nicht. `timeMode` repräsentiert hier alle möglichen Schachuhren, also beispielsweise „10 + 5“, „15 + 10“, ...

Diese Warteschlagen sind Listen, welche aus `username:userid` Einträgen bestehen, um die Anzahl an Abfragen zu verringern.
Durch die Redis Operationen `RPOP` und `LPUSH` lassen sich atomar Einträge hinzufügen oder herausnehmen, welches die Konsistenz der Liste gewährleistet.

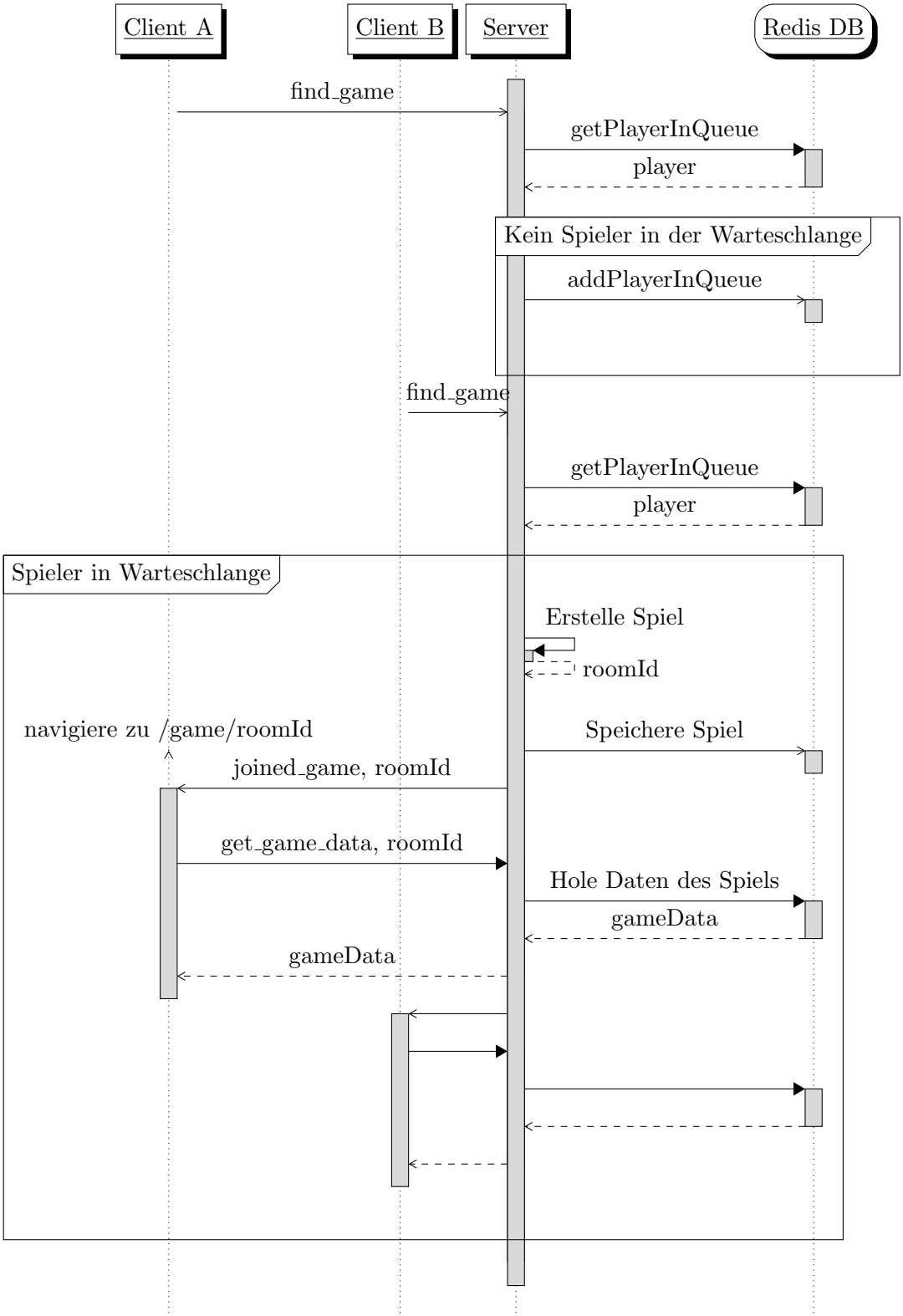


Abbildung 3.19: Sequenzdiagramm des Schachspielstartprozesses mit unbekanntem Gegner

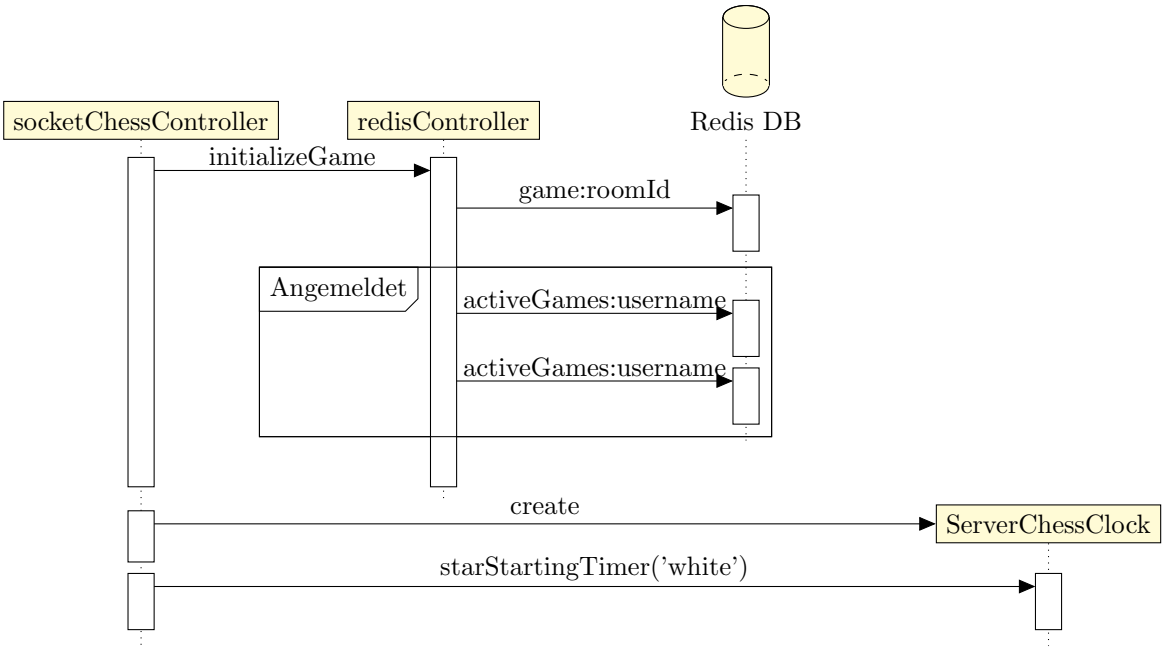


Abbildung 3.20: Sequenzdiagramm des Initialisieren eines Schachspiels

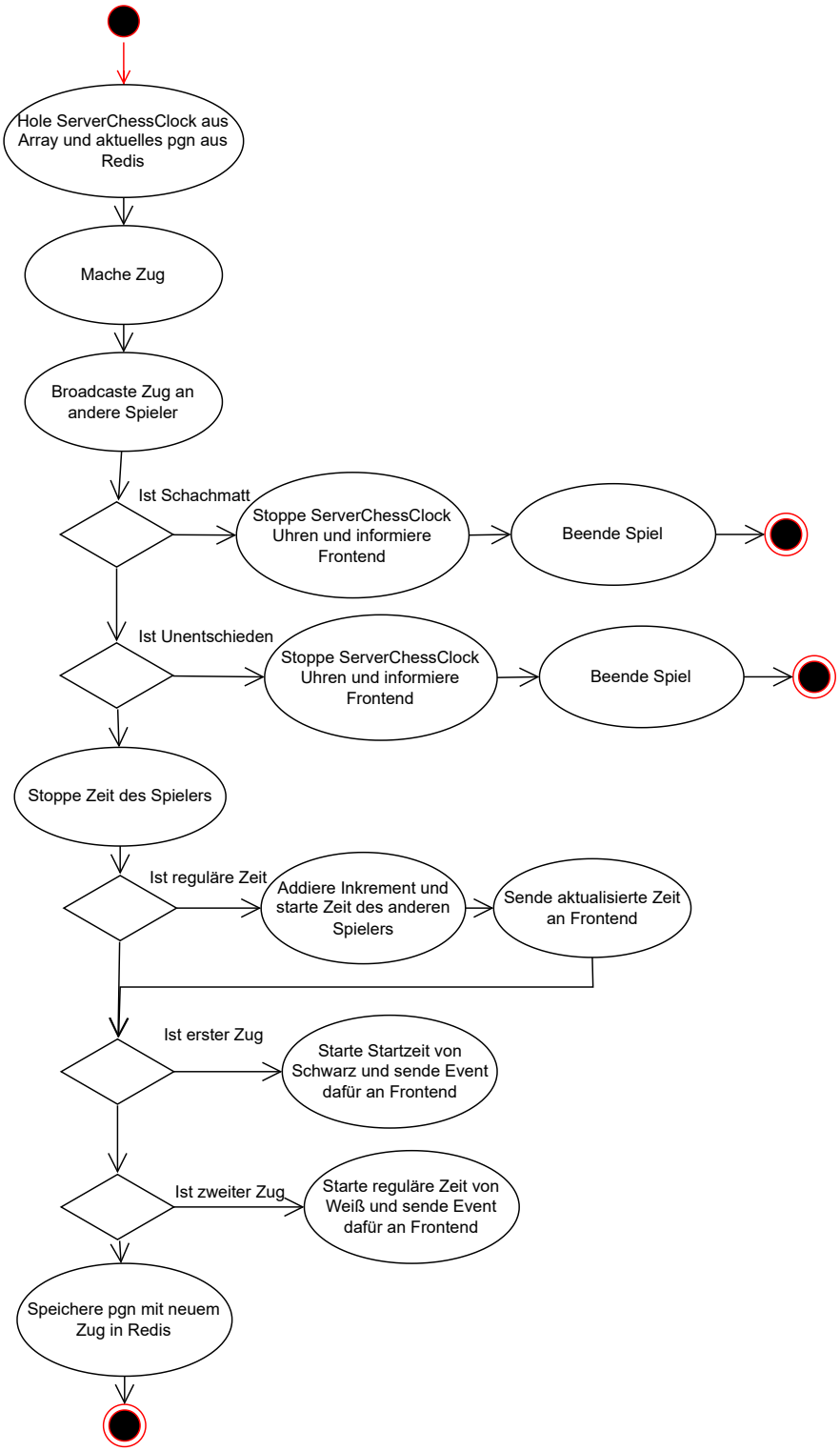


Abbildung 3.21: Aktivitätsdiagramm zur Behandlung eines neuen Zugs im Backend

4 Implementierung

4.1 Frontend-Entwicklung

4.1.1 Authentifizierung

Die verschiedenen Art und Weisen und Abläufe der Authentifizierung wurde im Kapitel 3.4.2 behandelt. In diesem Kapitel werde ich näher auf die genaue Implementierung dieser Abläufe eingehen, erklären wie die Formik Formulare funktionieren und konkrete Code Ausschnitte vorstellen.

Erster Versuch der Authetifizierung mittels Cookie

Im Code Ausschnitt 4.1 ist der Code der gesamten AccountContext Datei zu sehen, welche den *UserContext* zur Verfügung stellt.

```
1  import React, {useEffect, useState, createContext} from "react";
2
3  export const AccountContext = createContext();
4
5  const UserContext = ({children}) => {
6      const [user, setUser] = useState({loggedIn: null});
7
8       /**
9       * Set User with Cookie, if possible
10      */
11     useEffect(() => {
12         fetch(`${process.env.REACT_APP_API_URL}/auth/login`, {
13             credentials: "include",
14         })
15             .catch(err => {
16                 console.log(err);
17                 return;
18             })
19             .then(r => {
20                 if (!r || !r.ok || r.status >= 400) {
21                     return;
22                 }
23                 return r.json();
24             })
25             .then(data => {
26                 setUser({ ...data});
27             });
28     }, []);
29     return (
```

```
30          <AccountContext.Provider value ={{user, setUser}}>
31              {children}
32          </AccountContext.Provider>
33      );
34  }
35  export default UserContext;
```

Code Ausschnitt 4.1: Die AccountContext.js-Datei

In ihr wird der State `user` mit `{loggedIn: null}` initialisiert. Weshalb wir dies tun wird durch den Code Ausschnitt 4.6 und dessen Erklärung in Abschnitt 4.1.1 ersichtlich. Sobald die Komponente gerendert wurde, wird die Funktion der `useEffect`-Hook ausgeführt. Das leere Dependency Array verursacht, dass sie nur dieses eine Mal beim starten der Anwendung ausgeführt wird. Die Funktion schickt eine HTTP GET-Anfrage unter `/auth/login` an den Server mit der Option, dass Credentials mitgesendet werden sollen. Dies stellt sicher, dass der Cookie mit dem JWT-Token an den Server gesendet wird und dort verifiziert werden kann. Der restliche Pfad ist als Umgebungsvariable gesetzt um den Pfad flexibel ändern zu können.
Gibt es einen Fehler oder eine ungültige Antwort wird der Vorgang abgebrochen, ansonsten wird der `user` mit den erhaltenen Daten gesetzt.
Mögliche Optionen sind dabei:

- `{loggedIn: false}`, falls man nicht authentifiziert werden konnte.
- `{loggedIn: true, username: Max}`, bei erfolgreichem Authentifizieren. (Max ist hier nur ein Beispiel als username)

Mehr Informationen benötigt der Benutzer aktuell nicht über sich selbst.

Authentifizierung mittels *SignUp*- oder *Login*-Komponente

Um sich mit Hilfe von den *SignUp*- oder *Login*-Komponenten anzumelden wird eine HTTP POST-Anfrage an den Server unter dem Pfad `/auth/login` oder `/auth/signup` gesendet.
Die Funktion zum Senden der Login-Daten an den Server befindet sich in Code Ausschnitt 4.2.

```
1   /**
2    * Submit the form and send it to the server. Set either the user
         data or an error message based on the response.
3    * @type {function({username, password}, function(boolean): void):
         void}
4    */
5   const submitLogin = useCallback((values, setSubmitting) => {
6       fetch(`${process.env.REACT_APP_API_URL}/auth/login`, {
7           method: "POST",
8           credentials: "include",
9           headers: {
10              "Content-Type": "application/json",
11          },
```

```
12              body: JSON.stringify(values)
13          })
14              .catch(err => {
15                  setLoginError("Please try again later");
16                  setSubmitting(false);
17                  return;
18              })
19              .then(res => {
20                  if (!res || !res.ok || res.status >= 400) {
21                      setSubmitting(false);
22                      setLoginError("Please try again later");
23                      return;
24                  }
25                  return res.json();
26              })
27              .then(data => {
28                  if (!data.loggedIn) {
29                      setLoginError(data.message);
30                      setSubmitting(false);
31                      return;
32                  }
33                  setUser({...data});
34                  setLoginError(null);
35                  navigate('/');
36              });
37      }, [setLoginError, setUser, navigate]);
```

Code Ausschnitt 4.2: Die Funktion zum Senden der Benutzerdaten an das Backend

Die Funktion zum Senden der Registrierungs-Daten sieht genau so aus, nur dass die Anfrage an einen anderen Pfad geht und es mehr Daten enthält, wie zum Beispiel die E-Mail.
Initialisiert wird die Funktion mittels der *useCallback*-Hook um unnötige Neuerstellungen zu vermeiden.Der „Content-Type“ hilft dem Server zu erkennen um was für eine Art von Daten es sich handelt, während im Body die angegebenen Anmeldedaten als String verpackt werden. Falls der Server mit einem Fehler Antwortet, wird dieser in dem State `loginError` erfasst. Ansonsten wird der Benutzerstatus gesetzt und es wird auf die *Home*-Komponente navigiert.
Das Formular wird mittels Formik reaktiv und nutzt Yup Schemata zum überprüfen der eingegebenen Werte.

```
1  ...
2  <Formik
3      initialValues={{
4          username: "",
5          password: "",
6      }}
7      validationSchema={LoginSchema}
8      validateOnChange={true}
9      onSubmit={(values, { setSubmitting }) => {
10         submitLogin(values, setSubmitting);
11     }}
```

```
12 >
13     {({ isValid, isSubmitting }) => (
14         <Form style={{ width: "100%" }}>
15             ...
16             <Field name="password">
17                {({ field, form }) => (
18                   <FormControl isInvalid={form.errors.password &&
                         form.touched.password}>
19                    <FormLabel htmlFor="password">Password</FormLabel>
20                    <InputGroup>
21                         <Input {...field}
22                          id="password"
23                          type={showPassword ? 'text' : 'password'}
24                          autoComplete="current-password" />
25                         <InputRightElement>
26                             <IconButton
27                                 icon={showPassword ? <ViewOffIcon />
                                       : <ViewIcon />}
28                                 onClick={handlePasswordClick}
29                                 variant="ghost"
30                                 _hover={{ bg: hover }}
31                                 aria-label="Toggle password
                                       visibility"
32                             />
33                         </InputRightElement>
34                     </InputGroup>
35                     <FormErrorMessage>{form.errors.password}
36                     </FormErrorMessage>
37                 </FormControl>
38             )}
39             </Field>
40             ...
41             <Button
42                 type="submit"
43                 isDisabled={!isValid || isSubmitting}
44                 ...
45             >
46                Log In
47             </Button>
```

Code Ausschnitt 4.3: Ein Ausschnitt der *Login*-Komponente mit Formik

In diesem Code Ausschnitt ist das von Formik verwaltete Formular der *Login*-Komponente zu erkennen (die Komponente ist in Abbildung 3.8 dargestellt). Als Schema für dieses Formular wird das Yup Schema `LoginSchema` verwendet (siehe Code Ausschnitt 4.4) und es wird definiert, dass bei jeder neuen Änderung der Eingabefelder des Formulars, die Eingaben auf das Schema getestet werden sollen.
`isValid` und `isSubmitting` (Zeile 13) sind props, die man von Formik zur Verfügung gestellt bekommt und die angeben, ob die Eingabefelder mit dem Schema übereinstimmen und ob gerade noch auf die Antwort des Servers gewartet wird. Dies ist auch der Grund, warum wir `setSubmitting` der `submitLogin` Funktion übergeben. Wird noch auf die Antwort des Servers gewartet oder das Formular stimmt nicht mit dem Schema überein,

wird der Button deaktiviert, sodass man ihn nicht mehr klicken kann zum Absenden.
Als Eingabefelder, gibt es den Benutzernamen und das Passwort. Aus Demonstrationszwecken habe ich in den Code Ausschnitt 4.3 nur das Passwort eingefügt.
Die Props `field` und `form` versorgen das Feld mit Informationen und Funktionen für das Feld und über das gesamte Formular. So wird in Zeile 21 das Input Feld mit dem `field` prop verknüpft, welches Attribute wie die Funktion `field.onChange` oder `field.value` besitzt. `form` wird genutzt um Beispielsweise wie in Zeile 35 ein Fehler beim Passwort anzuzeigen, falls dies nicht mit dem Schema übereinstimmt.
Gestaltet wird das Formular mit Komponenten von Chakra UI, welche viele hilfreiche Komponenten wie `FormErrorMessage` oder `IconButton` bereitstellt. Mit Attributen wie `_hover` oder `variant` können diese Komponenten noch weiter individualisiert werden.
Die *Login*-Komponente besitzt den boolean State `showPassword`, welcher angibt, ob das Passwort zu sehen sein soll oder nicht (Zeile 23). Bei einem Klick auf das Icon mit dem Auge wird die Funktion `handlePasswordClick` ausgeführt, welche den `showPassword` State negiert.
Die Formulierung von Schemata mittels Yup ist relativ simpel.

```
1  const LoginSchema = Yup.object().shape({
2      username: Yup.string()
3          .min(3, 'Username has to be at least 3 characters long')
4          .max(20, 'Username cannot be longer than 20 characters')
5          .test('not_guest', 'Username cannot start with "guest"', (value)
               => {
6              return !value.startsWith('guest');
7          })
8          .required('Username is a required field'),
9      password: Yup.string()
10         .min(8, 'Password has to be at least 8 characters long')
11         .minLowercase(1, 'At least one character hat to be in lower
               case')
12         .minUppercase(1, 'At least one character has to be in upper
               case')
13         .minNumbers(1, 'At least on character has to be a number')
14         .minSymbols(1, 'At least one character has to be a symbol')
15         .minRepeating(3, 'It is only allowed to have at most 3 repeating
               characters')
16         .required('Password is a required field'),
17 });
```

Code Ausschnitt 4.4: Yup Schma für das Anmelden

In diesem Beispiel ist das `LoginSchema` zu sehen, welches für die Validierung der Eingabefelder im Anmeldeformular aus Code Ausschnitt 4.3 verwendet wird. Yup ermöglicht es bestimmte Restriktionen an Eingabefelder zu stellen und dabei direkt die Fehlernachricht zu definieren, falls diese Restriktion nicht erfüllt ist. Selbstverständlich sind die Restriktionen für das Registrieren die gleichen, wobei das Registrierungsformular noch ein paar weitere Felder hat.
Ich verwende übliche Restriktionen an Benutzernamen und Passwörter, sodass das Passwort mindestens eine Nummer, ein Symbol, einen Großbuchstaben, ... beinhalten muss.

Eine Sonderheit ist, dass der Benutzername nicht mit „guest“ beginnen darf, da ich unangemeldete Benutzer bei Schachpartien einen solchen Benutzernamen gebe.

Socket.io Verbindungsaufbau

Immer sobald sich der Benutzerzustand ändert wird eine Socket.io Verbindung hergestellt und den anderen Komponenten zur Verfügung gestellt. Dies hat den Nutzen, dass beim Verbindungsaufbau immer auch die socket auf dem Server authentifiziert werden muss. Dafür wird die *SocketContext*-Komponente verwendet:

```
1  import {io} from "socket.io-client";
2  import React, {useContext, useEffect, useState, createContext} from
       "react";
3  import {AccountContext} from "./AccountContext.js";
4
5  export const SocketContext = createContext();
6
7  function SocketConnectionContext({children}) {
8      const [socket, setSocket] = useState(null);
9      const {user} = useContext(AccountContext);
10
11     /**
12      * New socket connection every time the user changes.
13      */
14     useEffect(() => {
15         if(user.loggedIn !== null) {
16             setSocket(new io(process.env.REACT_APP_SOCKET_URL, {
17                 withCredentials: true
18             }));
19         }
20     }, [user]);
21
22     return (
23         <SocketContext.Provider value={{socket}}>
24             {children}
25         </SocketContext.Provider>
26     );
27 }
28
29 export default SocketConnectionContext;
```

Code Ausschnitt 4.5: Die Datei *SocketContext.js*

Bei der Verbindung wird darauf geachtet, dass die Credentials mitgesendet werden, also beispielsweise auch Cookies. Des weiteren wird erst eine Verbindung aufgebaut, sobald die Antwort des Servers auf die Authentifizierungs-Anfrage mittels Cookie (siehe Abschnitt 4.1.1) empfangen wurde und dadurch das `user.loggedIn` Attribut nicht mehr `null` ist. Dies dient dazu einen unnötigen Verbindungsaufbau zu unterlassen, da nach der Antwort des Servers ohnehin eine neue Verbindung aufgebaut werden würde.
Erst sobald die erste Authentifizierung stattgefunden hat und eine Socket.io Verbindung hergestellt wurde, werden die verschiedenen Elemente und Routen in der `Views`-

Komponente definiert. So lange wird ein Lade-Bildschirm gezeigt (siehe Code Ausschnitt 4.6).

```
1        ...
2         {user.loggedIn === null || socket === null ?
3                 <Flex align="center" justify="center" direction="column"
                      height="80vh">
4                     <Heading as='h2' size='lg'>Loading...</Heading>
5                     <Spinner size='xl' color="purple.500" marginTop="4"/>
6                 </Flex>
7
8                 :
9                 <>
10                    <Navbar/>
11                    ...
```

Code Ausschnitt 4.6: Ausschnitt der *Views*-Komponente

4.2 Backend-Entwicklung

4.2.1 Authentifizierung

Die Vorgehensweise im Backend bei der Authentifizierung des Benutzers ist im Kapitel 3.5.1 geschildert. In diesem Kapitel werde ich Code Beispiele vorstellen und erläutern.

Routing-Middlewares

Der Code Ausschnitt 4.7 zeigt die Reihenfolge der Middlewares bei den Anmelde- und Registrierungsanfragen des Frontends.

```
1  //index.js
2  ...
3  app.use("/auth", authRouter);
4  ...
5
6  //authRouter.js
7  const express = require('express');
8  const router = express.Router();
9  ... //import der Middlewares
10
11 router.use(rateLimiter(60, 10));
12
13 router.route('/login').get(handleLogin).post(validateLogin,
       attemptLogin);
14
15 router.post('/signup', validateSignUp, attemptSignUp);
16
17 router.get('/logout', handleLogout);
18
19 module.exports = router;
```

Code Ausschnitt 4.7: Ausschnitt aus index.js und die Datei authRouter.js

Für jede Anfrage auf den Pfad `/auth` wird die Middleware `rateLimiter` verwendet. Dieser achtet darauf, dass nicht zu viele Anfragen einer IP Adresse gesendet werden.

```
1  const redisClient = require("../redis/redis.js");
2  /**
3   * Rate limiter middleware that limits the number of requests from a
        given IP address within a specified time window.
4   * @param limitAmount - The maximum number of requests allowed within
        the time window.
5   * @returns {function(*, *, *): Promise<void>} - Middleware function
        that either sends an error response if the limit is exceeded or
        calls the next middleware or route handler.
6   */
7  module.exports.rateLimiter = (secondsLimit, limitAmount) => async (req,
       res, next) => {
8      const ip = req.connection.remoteAddress;
9      [response] = await redisClient
10         .multi()
11         .incr(ip)
12         .expire(ip, secondsLimit)
13         .exec();
14     if (response[1] > limitAmount) {
15         res.json({
16             loggedIn: false,
17             message: "Slow down!! Try again in a minute.",
18         });
19         res.sendStatus(429);
20     }
21     else next();
22 };
```

Code Ausschnitt 4.8: Die rateLimiter Middleware

Der `rateLimiter` nimmt die Argumente secondsLimit als Zeitfenster und limitAmount als Anfragelimit an und definiert mit Hilfe von ihnen die Middleware. Dafür nutzt er Redis und setzt mit der IP-Adresse als Key die Zahl der Anfragen als Value. Dieser Eintrag wird nach Ablauf des Zeitfensters gelöscht und falls das Anfragelimit innerhalb dieses Zeitfensters überschritten wurde, wird eine Fehlermeldung mit dem Statuscode 429 gesendet, der für „Too Many Requests“ steht[1]. Andernfalls wird die nächste Middleware aufgerufen.
In unserer Anwendung legen wir fest, dass 10 Anfragen innerhalb von 60 Sekunden gesendet werden dürfen.
Für die erste Authentifizierung im *UserContext* (siehe Abschnitt 4.1.1) wird die Middleware `handleLogin` verwendet.

```
1  const getJwtFromCookie = req => {
2      return req.headers["cookie"] &&
           cookie.parse(req.headers["cookie"])["jwt"];
3  };
4
```

[1]Quelle: `https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/429` am 05. April 2023

```
5  module.exports.handleLogin = (req, res) => {
6      const token = getJwtFromCookie(req);
7      if(!token) {
8          res.json({loggedIn: false});
9      } else {
10         jwt.verify(token, process.env.JWT_SECRET, (err, decodedPayload)
               => {
11             if (err) {
12                 res.json({loggedIn: false});
13             } else {
14                 res.json({loggedIn: true, username:
                       decodedPayload.username});
15             }
16         });
17     }
18 };
```

Code Ausschnitt 4.9: Die handleLogin Middleware zum Authentifizieren mit Cookie

Diese Middleware ließt gegebenenfalls den Cookie mit dem JWT-Token und antwortet entsprechend ob er ihn erfolgreich dekodieren konnte oder nicht.
Bei den POST Anfragen des Frontends, über die das ausgefüllte Anmelde-, beziehungsweise Registrierungsformular gesendet wird, wird vor der Auswertung mit den Middlewares `validateLogin` oder `validateSignUp` überprüft, ob das Formular auch dem entsprechenden Yup Schema übereinstimmt. Anschließend wird mit `attemptLogin` oder `attemptSignUp` versucht den Benutzern anzumelden oder zu registrieren.

```
1  /**
2   * Attempts to log in the user by checking the provided credentials
        against the database.
3   * If the credentials match, creates a JWT token and sets it as a cookie
        in the response.
4   * @param req - The incoming request object containing the user's login
        information.
5   * @param res - The outgoing response object.
6   * @returns {Promise<void>} - The Middleware
7   */
8  module.exports.attemptLogin = async (req, res) => {
9      const potentialLogin = await query(
10         "SELECT id, username, password, userid FROM users u WHERE
               u.username=$1",
11         [req.body.username]
12     );
13     if (potentialLogin.rowCount > 0) {
14         const isSamePassword = await bcrypt.compare(
15             req.body.password,
16             potentialLogin.rows[0].password
17         );
18         if (isSamePassword) {
19             const user = {
20                 username: req.body.username,
21                 userid: potentialLogin.rows[0].userid,
22             }
```

```
23              jwt.sign(
24                  user,
25                  process.env.JWT_SECRET,
26                  {expiresIn: "24h"},
27                  (err,token) => {
28                  if(err) {
29                      res.json({loggedIn: false, message: "Something went
                            wrong, try again later"});
30                  } else {
31                      const jwtCookie = cookie.serialize("jwt", token, {
32                          httpOnly: true,
33                          secure: process.env.NODE_ENV === "production",
                                // Secure flag only in production
34                          maxAge: 24 * 60 * 60, // 24 hours
35                          sameSite: "lax",
36                          path: "/"
37                      });
38                      res.setHeader("Set-Cookie", jwtCookie);
39                      res.json({loggedIn: true,  username: user.username});
40                  }
41              }
42              );
43          } else {
44              res.json({loggedIn: false, message: "Wrong username or
                    password!"});
45          }
46      } else {
47          res.json({loggedIn: false, message: "Wrong username or
                password!"});
48      }
49  };
```

Code Ausschnitt 4.10: Die attemptLogin Middleware zum Anmelden

Die Middleware `attemptLogin` überprüft zunächst, ob ein Benutzer mit diesem Benutzernamen existiert und mittels bcrypt ob die Passwörter übereinstimmen. Ist eines davon nicht der Fall wird das Frontend darüber benachrichtigt. Ansonsten wird ein 24 Stunden haltbarer JWT-Token mit dem Benutzernamen und der `userid` generiert und in einen Cookie mit dem Key „jwt“ gesetzt, welcher ebenfalls 24 Stunden gültig ist. Der JWT-Token beinhaltet die `userid`, da die Socket Verbindung diese Information beim authentifizieren benötigt (siehe Abschnitt 4.2.1). Wichtig dabei ist, dass das Attribut `httpOnly` auf `true` gesetzt wird, um zu verhindern, dass Client seitiger Code auf den Cookie zugreifen könnte. Anschließend wird dem Frontend noch mit dem JSON Objekt geantwortet (siehe Zeile 39).
Beim Registrieren wird ebenfalls bei Erfolg der Cookie mit dem JWT-Token gesetzt, jedoch sind die Anforderungen natürlich andere. Es wird überprüft, ob bereits ein Account mit dem Benutzernamen oder der E-Mail existiert und falls ja wird dem Frontend entsprechend geantwortet. Anschließend wird die `userid` erstellt, das Passwort mit bcrypt verschlüsselt, der Tupel in der Datenbank gespeichert und der Cookie mit dem JWT-Token erstellt.

Abmelden

Da das Abmelden auch gewissermaßen zum Authentifizierungsprozess gehört, wird hier kurz beschrieben, was im Backend dabei passiert.

```
/**
 * Handles user logout by clearing the JWT cookie.
 * @param req - The request object.
 * @param res- The response object.
 */
module.exports.handleLogout = (req, res) => {
    res.setHeader(
        "Set-Cookie",
        cookie.serialize("jwt", "", {
            httpOnly: true,
            secure: process.env.NODE_ENV === "production",
            maxAge: -1,
            sameSite: "lax",
            path: "/"
        })
    );
    res.sendStatus(204);
}
```

Code Ausschnitt 4.11: Middleware zum Abmelden

Eine Anfrage zum Abmelden wird auf den Pfad `auth/logout` gestellt unter der die Middleware aus Code Ausschnitt 4.11 durchlaufen wird. In dieser wird ein abgelaufener „jwt“-Cookie gesetzt, der bewirkt, dass der bisherige „jwt“-Cookie gelöscht wird. Dann wird dem Frontend mit dem Code 204 geantwortet, welcher aussagt, dass die Anfrage erfolgreich bearbeitet wurde, jedoch keine Daten mit der Antwort gesendet werden[2].

Socket.io Authentifizierung und Middleware

Im Code Ausschnitt 4.12 ist ein Ausschnitt der index.js Datei zu sehen, in der der Socket.io Server initialisiert wird und die Middlewares für socket Verbindungen zugewiesen werden.

```
...
const corsConfig = {
    origin: process.env.FRONTEND_URL,
    credentials: true,
}
const server = require('http').createServer(app);
const io = new Server(server, {
    cors: corsConfig
});
io.use(authorizeUser);
io.use(initializeUser);
io.use((socket, next) => {
```

[2]Quelle `https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204` am 05. Mai 2023

```
13      initializeChessListeners(socket, io);
14      initializeListeners(socket, io);
15      next();
16  });
17  ...
```

Code Ausschnitt 4.12: Ausschnitt der index.js Datei mit Socket.io Server Erstellung und Middleware Zuweisung

Die Initilaisierung des Socket.io Servers benötigt die `corsConfig` Konfiguration, um Verbindungen vom Frontend zuzulassen und Header Informationen wie Cookies mitzusenden.
Es werden drei Middlewares bei einem Verbindungsaufbau durchlaufen, die ich in diesem Kapitel mit Code ausschnitten erläutern werde.

```
1   /**
2    * Middleware that authorizes the socket connection user based on the
         JWT in the cookie.
3    * If the user is successfully authorized, their profile is added to the
         socket object.
4    *
5    * @param socket - The socket object representing the connection to the
         client.
6    * @param next - The next middleware function to be called if
         authorization is successful.
7    */
8   module.exports.authorizeUser = (socket, next) => {
9       let token = null;
10      if (socket.request.headers.cookie) {
11          token = cookie.parse(socket.request.headers.cookie).jwt;
12          if (token) {
13              jwt.verify(token, process.env.JWT_SECRET, (err,
                    decodedPayload) => {
14                  if (err) {
15                      next(new Error('Unable to Read Token'));
16                      return;
17                  }
18                  socket.user = {...decodedPayload};
19              });
20          }
21      }
22      next();
23  }
```

Code Ausschnitt 4.13: authorizeUser Middleware für Socket.io Verbinudngen

Die Middleware `authorizeUser` (siehe Code Ausschnitt 4.13 autorisiert den Benutzer. Dabei wird auf den Cookie zugegriffen, der gegebenenfalls beim Authentifizieren gesetzt wurde, um den Benutzernamen und die `userid` als Attribute der socket zu setzen. Diese Middleware ist der Grund, warum immer eine neue Socket.io Verbindung hergestellt werden muss, sobald der Benutzerzustand und damit der Cookie sich verändert. Ansonsten könnte es sein, dass veraltete Informationen in `socket.user` definiert sind.

```
1  /**
2   * Middleware that initializes the user data when they connect to the
        socket server.
3   * The function retrieves the user's friends, friend requests, and
        active games data,
4   * and sends them to the client. It also emits a 'connected' event to
        the user's friends.
5   *
6   * @param socket - The socket object representing the connection to the
        client.
7   * @param next - The next middleware function to be called.
8   * @returns {Promise<void>}
9   */
10 module.exports.initializeUser = async (socket, next) => {
11     if (socket.user) {
12         socket.join(socket.user.userid);
13         await setUser(socket.user.username, socket.user.userid, true);
14         getFriends(socket.user.username).then(async friends => {
15             const parsedFriends = await parseFriendList(friends);
16             const friendRooms = parsedFriends.map(friend =>
                   friend.userid);
17             if (friendRooms.length > 0) {
18                 socket.to(friendRooms).emit("connected", "true",
                       socket.user.username);
19             }
20             socket.emit('friends', parsedFriends);
21         });
22         getFriendRequests(socket.user.username).then(friendRequests =>
               socket.emit('friend_requests', friendRequests));
23         getActiveGamesData(socket.user.username).then(gameData =>
               socket.emit('active_games', gameData));
24     }
25     next();
26 };
```

Code Ausschnitt 4.14: initializeUser Middleware für Socket.io Verbinudngen

In der Middleware `initializeUser` werden Daten an den Benutzer gesendet, er wird als online vermerkt und seine Freunde werden darüber informiert, dass er nun online ist. Dies passiert natürlich nur, wenn in der vorherigen Middleware der Benutzer autorisiert werden konnte. Die Funktionen um Daten aus der Redis Datenbank zu lesen oder zu schreiben werden von der Datei `redisController` bereitgestellt.
Zunächst tritt er dem Raum seiner eigenen `userid` bei um Events wie Freundschaftsanfragen, die an ihn gerichtet sind empfangen zu können (siehe Zeile 12). Anschließend wird der Benutzer in Redis gesetzt und als online vermerkt (Zeile 13). Freunde, Freundschaftsanfragen und aktive Spieler werden an den Benutzer mit entsprechenden Events gesendet und alle Freunde werden darüber benachrichtigt, dass der Benutzer online ist. Die dritte Middleware setzt alle nötigen Listener für Events, die mit dem Schachspiel zusammenhängen, als auch sonstige Funktionen (siehe Code Ausschnitt 4.12).

```
1  /**
```

```
 * Function that handles user disconnection from the socket server.
 * The function updates the user's online status and informs all friends,
 * that the user ist offline
 *
 * @param socket - The socket object representing the connection to the
     client.
 * @returns {Promise<void>}
 */
module.exports.onDisconnect = async (socket) => {
    if (!socket.user) {
        return;
    }
    await setUser(socket.user.username, socket.user.userid, false);
    getFriends(socket.user.username).then(friends => {
        const friendRooms = friends.map(friend => friend.userid);
        if (friendRooms.length > 0) {
            socket.to(friendRooms).emit("connected", "false",
                socket.user.username);
        }
    });
}
```

Code Ausschnitt 4.15: Die onDisconnect Methode für Sockets

Die `onDisconnect` Methode aus Code Ausschnitt 4.15 wird aufgerufen, sobald eine socket Verbindung getrennt wird. Ähnlich wie bei der `initializeUser` Middleware (siehe Code Ausschnitt 4.14) wird dabei in Redis der Spieler als offline vermerkt und seine Freunde werden darüber in Kenntnis gesetzt.

4.3 Das Schachspiel

Das Schachspiel verwaltet die Datei `socketChessController.js`, welche eine der Funktionen zur Listener Initialisierung bereitstellt (siehe `initilializeChessListeners` aus Code ausschnitt 4.12), wobei `ServerChessClock.js` die Schachuhren bereitstellt.

4.3.1 Das Spiel

Initialisierung eines Spiels

Die Suche und das Finden eines Gegners wurde in Abschnitt 3.5.3 erläutert. In diesem Abschnitt möchte ich genauer darauf eingehen, wie ein Spiel nach gefundenem Gegner initialisiert wird.

```
/**
 * Creates a new chess game with two players and initializes the game
     state and chess clock.
 *
 * @param  io - The Socket.IO server instance.
 * @param  username1 - The username of the first player.
 * @param  username2 - The username of the second player.
 * @param  time - The time mode for the chess clock.
```

```
8   * @returns {Promise<string>} - Returns a promise that resolves to the
        new game's room ID.
9   */
10  const createChessGame = async (io, username1, username2, time) => {
11      var roomId = UUIDv4();
12      const [whitePlayer, blackPlayer] = Math.random() < 0.5 ? [username1,
            username2] : [username2, username1];
13
14      const chessInstance = await
            import('../chess/Chess.mjs').then(ChessFile => {
15          return ChessFile.Chess();
16      });
17      const d = new Date();
18      chessInstance.header('White', whitePlayer, 'Black', blackPlayer,
            'Date', d.toDateString());
19      //Store Game in Redis
20      await initializeGame(roomId, whitePlayer, blackPlayer, time,
            chessInstance.pgn());
21      const chessClock = new ServerChessClock(time);
22      currentChessClocks[roomId] = {chessClock};
23
24      chessClock.startStartingTimer('white');
25      chessClock.ChessClockEvents.on('cancel_game', () => {
26          io.to(roomId).emit('cancel_game');
27          io.to(roomId).emit('stop_clocks');
28          endGame(roomId)
29      });
30      chessClock.ChessClockEvents.on('time_over', (color) => {
31          io.to(roomId).emit('time_over', color);
32          io.to(roomId).emit('stop_clocks');
33          endGame(roomId);
34      });
35      return roomId;
36  }
```

Code Ausschnitt 4.16: Die createChessGame Methode zum Initialisieren einer Schachpartie

Die `createChessGame` Methode legt zunächst per Zufall fest, welcher Spieler welche Farbe spielen soll und generiert eine zufällige ID als roomId in der das Spiel stattfindet. Anschließend wird ein chess.js Objekt erstellt. Da es sich bei der Bibliothek chess.js um ein ECMAScript-Module handelt und node.js ein CommonJS-Modulsystem verwendet muss man einen Umweg mittels einer `.mjs`-Datei zum Importieren der Bibliothek verwenden[3] (siehe Code Ausschnitt 4.17).

```
1  import {Chess as ChessGame} from 'chess.js';
2  export const Chess = () => new ChessGame();
```

Code Ausschnitt 4.17: Die Chess.mjs Datei

[3] Quelle: `https://stackoverflow.com/questions/70396400/how-to-use-es6-modules-in-commonjs` am 06. Mai 2023

Mit diesem chess.js Objekt generieren wir eine PGN-Notation, in die die Benutzernamen der Spieler und das Datum eingetragen sind. Diese Informationen werden nun in Redis unter `game:roomId` gespeichert, falls die Spieler angemeldet sind auch in deren `activeGames:username` Listen.
Des weiteren wird ein ServerChessClock Objekt erstellt und in `currentChessClock` mit der roomId als Key gespeichert. Die Kommunikation und der Aufbau von ServerChessClock wird in Abschnitt //REF erläutert. Eventlistener für die Events der ServerChessClock werden definiert. Da die Schachuhren und das Schachspiel möglichst Modular gehalten werden, wird an das Frontend in diesen Listenern immer zwei Events gesendet: `time_over` und `cancel_game` für das Schachspiel und `stop_clocks` für die Schachuhren im Frontend. Bei jeder Möglichkeit wie eine Schachpartie endet wird definiert, dass die Funktion `endGame` ausgeführt werden soll, so auch bei diesen zwei Listenern.

4.3.2 Die Uhr

5 Fazit und Ausblick

5.1 Zusammenfassung der Ergebnisse

5.2 Limitationen

//ServerchessClock

5.3 Potenzielle Erweiterungen und Weiterentwicklung

//Trennung waitingPlayers, waitingGuests

Abbildungsverzeichnis