

# 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**

Das Interesse an Schach hat in den letzten Jahren immer mehr zugenommen und Online-Schachplattformen verzeichnen aktuell Rekorde an Benutzern und täglichen Spielen. Dies bietet die attraktive Möglichkeit eine Schach-App zu entwerfen und zu implementieren, deren Konzept Elemente aufgreift,

welche bei bisherigen Schachplattformen kritisiert werden (kann ich eigentlich nicht schreiben, weil das nur meine Meinung ist ;())

welche mehr Anreize schaffen Partien zu spielen und soziale Interaktionen mit Fremden aber auch innerhalb einer Freundesgruppe zu fördern.

Diese Bachelorarbeit hat das Ziel eine Webbasierte Multiplayer Schach-App zu entwerfen, die die Basis für Erweiterungen bildet, um konkurrenzfähig gegenüber den bisherigen Schachplattformen zu sein. Der Fokus liegt dabei besonders auf soziale Interaktionen, eine ansprechende Benutzererfahrung und Sicherheit hinsichtlich Benutzerdaten.



## **Abstract**

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

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Zielsetzung . . . . .	2
1.3	Aufbau der Arbeit . . . . .	2
1.4	Verwandte Arbeiten . . . . .	3
1.4.1	Chess.com . . . . .	3
1.4.2	Lichess . . . . .	4
<b>2</b>	<b>Theoretische Grundlagen</b>	<b>7</b>
2.1	Schach . . . . .	7
2.2	Web-Technologien . . . . .	8
2.2.1	Node.js und Express . . . . .	8
2.2.2	Socket.io . . . . .	10
2.2.3	React . . . . .	14
2.2.4	PostgreSQL und Redis . . . . .	17
2.2.5	Weitere verwendete Bibliotheken . . . . .	19
<b>3</b>	<b>Systemarchitektur und Konzeption</b>	<b>23</b>
3.1	Einführung . . . . .	23
3.2	Architekturübersicht . . . . .	24
3.3	Konzeption der Schachuhren . . . . .	24
3.4	Frontend-Architektur . . . . .	26
3.4.1	React-Komponenten . . . . .	26
3.4.2	Authentifizierung . . . . .	30
3.4.3	Das Schachspiel . . . . .	34
3.4.4	Verwaltung von befreundeten Personen . . . . .	40
3.4.5	Anzeigen und navigieren zu aktiven Partien . . . . .	42
3.5	Backend-Architektur . . . . .	42
3.5.1	Authentifizierung . . . . .	43
3.5.2	Das Schachspiel . . . . .	45
3.5.3	Verwaltung von Freunden . . . . .	49
3.6	Datenbankstruktur . . . . .	51
3.6.1	PostgreSQL Datenbank . . . . .	51
3.6.2	Redis . . . . .	52
3.7	Testen der Anwendung . . . . .	53

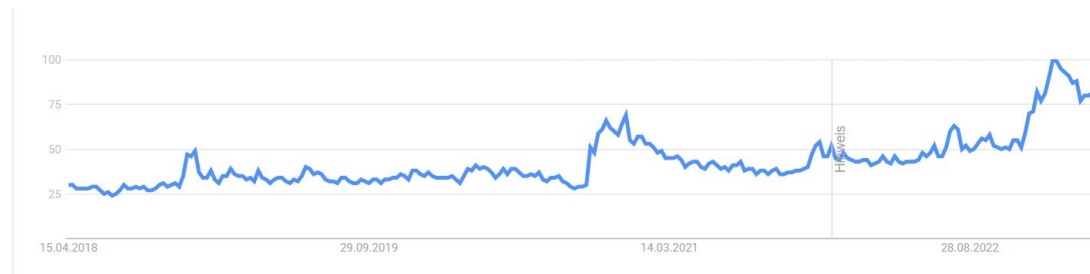
---

<b>4</b>	<b>Implementierung</b>	<b>55</b>
4.1	Frontend-Entwicklung . . . . .	55
4.1.1	Authentifizierung . . . . .	55
4.1.2	Das Schachspiel . . . . .	61
4.1.3	Verwaltung von Freunden . . . . .	67
4.1.4	Design . . . . .	69
4.2	Backend-Entwicklung . . . . .	71
4.2.1	Authentifizierung . . . . .	71
4.2.2	Das Schachspiel . . . . .	77
4.2.3	Verwaltung von Freunden . . . . .	85
4.3	Datenbank Integration . . . . .	86
4.3.1	Zugriff auf Redis mit dem redisController . . . . .	87
<b>5</b>	<b>Fazit und Ausblick</b>	<b>92</b>
5.1	Zusammenfassung der Ergebnisse . . . . .	92
5.2	Herausforderungen . . . . .	92
5.2.1	Schachuhren . . . . .	92
5.2.2	Züge des Schachspiels . . . . .	93
5.3	Zukünftige Erweiterungen und Verbesserungen . . . . .	94
	<b>Literaturverzeichnis</b>	<b>96</b>
	<b>Codeausschnitte</b>	<b>101</b>

# 1 Einleitung

## 1.1 Motivation

Schach ist ein traditionsreiches und abwechslungsreiches Brettspiel, dessen Ursprung nicht genau bestimmt werden kann. Allerdings wird vermutet, dass das erste schachähnliche Spiel *Tschaturanga* seinen Ursprung in Nordindien um 600 n. Chr. hatte [vdL74]. Bis heute bleibt Schach ein beliebtes Spiel, das 2020 durch die Netflix Serie „Damengambit“<sup>1</sup> und 2022 durch den Betrugsvorwurf von Magnus Carlsen an seinen 19-jährigen Gegner Hans Niemann<sup>2</sup> 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 über zehn Millionen Schachpartien täglich<sup>3</sup>, während Schach Live-Streams auf Plattformen wie `twitch.com` Millionen von Followern anziehen<sup>4</sup>.

Die Entwicklung einer webbasierten Multiplayer-Schach-App bietet eine einzigartige Gelegenheit, ein traditionsreiches und beliebtes Spiel im digitalen Zeitalter weiterzuentwickeln. Mein Ziel 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 die Aussicht in Zukunft, innovative Funktionen zu integrieren, die bislang in den gängigen Schach-Apps nicht oder nicht

<sup>1</sup>Quelle: [https://de.wikipedia.org/wiki/Das\\_Damengambit](https://de.wikipedia.org/wiki/Das_Damengambit) am 22. April 2023

<sup>2</sup>Quelle: <https://www.sportschau.de/schach/magnus-carlsen-hans-niemann-ermittlungen-100.html> am 22. April 2023

<sup>3</sup>Quelle: <https://www.chess.com/about> am 12. Mai 2023

<sup>4</sup>Quelle: <https://www.twitch.tv/directory/game/Chess> am 27. April 2023

kostenfrei vorhanden sind motiviert diese Arbeit. 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. Bei der Anwendung soll dabei vor allem soziale Interaktionen im Zusammenhang mit Schach im Vordergrund stehen. Diese Schach-App trägt vorerst den Namen „Gambit“, in der Hoffnung, dass durch die Serie *Damengambit* dieser Name in der Gesellschaft mit Schach verbunden wird.

User Experience (kurz UX) bezieht sich darauf, wie ein/-e Nutzer/-in sich auf einer Anwendung bewegt und wie einfach und angenehm es für den Nutzenden ist, die Funktionen der Anwendung zu verwenden.

Unter User Interface (kurz UI) versteht man die visuelle und interaktive Gestaltung von Benutzeroberflächen<sup>5</sup>. Es umfasst die Gestaltung von Buttons, Formularen und anderen visuellen Komponenten, sowie das Feedback dieser Komponenten, wie zum Beispiel die Rückmeldung einer fehlgeschlagenen Anmeldung. [Rob12]

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 (auch mehrere gleichzeitig und ohne angemeldet zu sein) mit einem Chat und verschiedenen Einstellungsmöglichkeiten der Spielzeiten selbst.

Dabei wird besonderer Wert auf die Verwendung moderner Web-Technologien wie React, Node.js, Socket.IO, Redis und PostgreSQL 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.

Die Anwendung dient hauptsächlich als Basis Schach-App für Erweiterungen. So ist das Webdesign noch nicht responsiv, da mit neuen Erweiterungen die UI und UX ohnehin angepasst werden müsste, um weitere Funktionen zur Verfügung zu stellen.

Mögliche Erweiterungen werden in Kapitel 5.3 erläutert.

## 1.3 Aufbau der Arbeit

Diese Bachelorarbeit gliedert sich in vier Hauptkapitel, die jeweils unterschiedliche Aspekte der Schach-App behandeln.

Das Kapitel **Theoretische Grundlagen** erläutert die grob Grundlagen von Schach als Spiel sowie die verwendeten Web-Technologien wie Node.js, Express, Socket.io, React,

---

<sup>5</sup>In dieser Arbeit wird bei einigen Wörtern wie Benutzeroberfläche, Benutzername und Benutzerdaten nicht gegendert. Es ist jedoch ausdrücklich zu betonen, dass alle Geschlechter gleichermaßen gemeint und angesprochen sind.

der Datenbanken und anderen Bibliotheken die für das Verständnis der nachfolgenden Kapitel wichtig sind.

Im Kapitel **Systemarchitektur** wird die Gesamtarchitektur der Schach-App beschrieben. Das Kapitel behandelt neben einer gesamten Übersicht die Unterteilung in Frontend- und Backend-Architektur einschließlich der Datenbankstruktur und der Kommunikation zwischen den verschiedenen Komponenten. Abläufe und Konzepte der Anwendung werden in Aktivitäts- und Sequenzdiagrammen verdeutlicht.

Das Kapitel **Implementierung** geht auf die praktische Umsetzung der Schach-App ein, indem es die Entwicklung für das Frontend und das Backend sowie die Integration der Datenbanken erläutert. Komplexere Prozesse und Methoden werden anhand von Codeausschnitten verdeutlicht.

Im abschließenden fünften Kapitel, **Fazit und Ausblick**, werden die Ergebnisse der Arbeit zusammengefasst, Herausforderungen und Alternativen diskutiert und mögliche Erweiterungen und Weiterentwicklungen für die Schach-App vorgeschlagen.

Die Arbeit endet mit dem **Anhang**, der die Listen der verwendeten Literatur, Grafiken und Codeausschnitten enthält.

## 1.4 Verwandte Arbeiten

Es gibt vor allem zwei große online Schachplattformen: `chess.com` und `lichess.org`. Dabei war `chess.com` auf Platz 114 und `lichess.org` auf Platz 209 der Webseiten mit am meisten Web-Traffic weltweit im April 2023<sup>6</sup>.

### 1.4.1 Chess.com

Chess.com zeichnet sich durch vielfältige Funktionen, einer verspielten UI und sehr vielen Benutzenden aus. Was im Vergleich zu Lichess jedoch schnell bemerkbar ist, ist die Kommerzialisierung.

Chess.com hat auf seiner gesamten Anwendung viel Werbung und bietet viele Funktionen nur bei einem monatlichen Abonnement gegen Geld an (siehe Abbildung 1.3). Zu diesen Funktionen gehören Analysen der Schachpartien, mehrere Spiele gleichzeitig zu spielen, der Zugriff auf Lektionen zum Lernen von Schach und vieles weitere.

Die Funktionen die Chess.com zur Verfügung stellt (kostenpflichtige, als auch kostenfreie) sind dafür sehr umfangreich. Es gibt Clubs mit Turnieren, es gibt verschiedene Spielmodi, die beispielsweise ermöglichen zu viert ein Schachspiel zu spielen und Taktikaufgaben zum lösen.

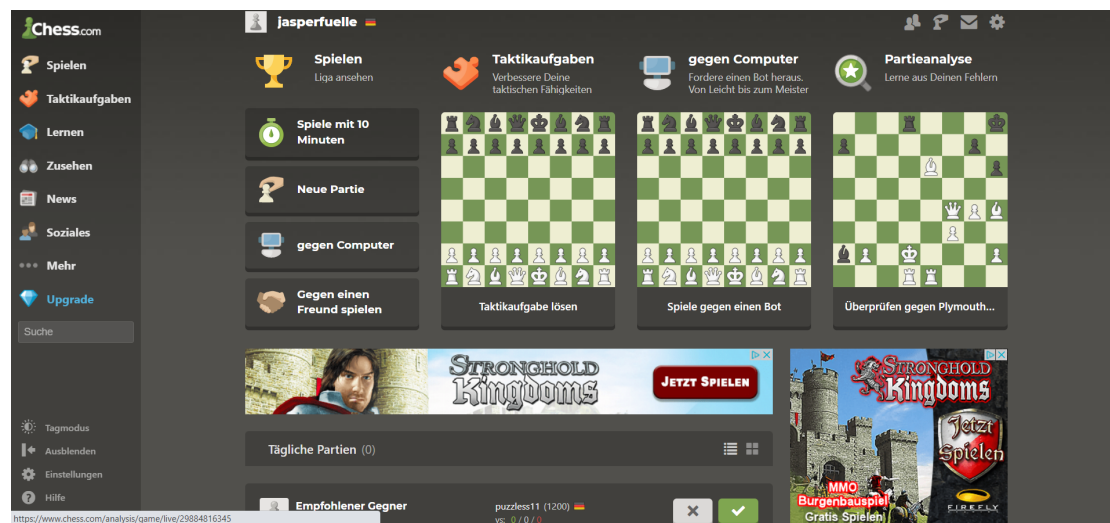
Chess.com zeichnet sich auch durch seine Online-Präsenz auf Plattformen wie YouTube<sup>7</sup> oder twitch<sup>8</sup> aus.

---

<sup>6</sup>Quelle: <https://www.similarweb.com/website/chess.com/vs/lichess.org/#overview> am 12. Mai 2023

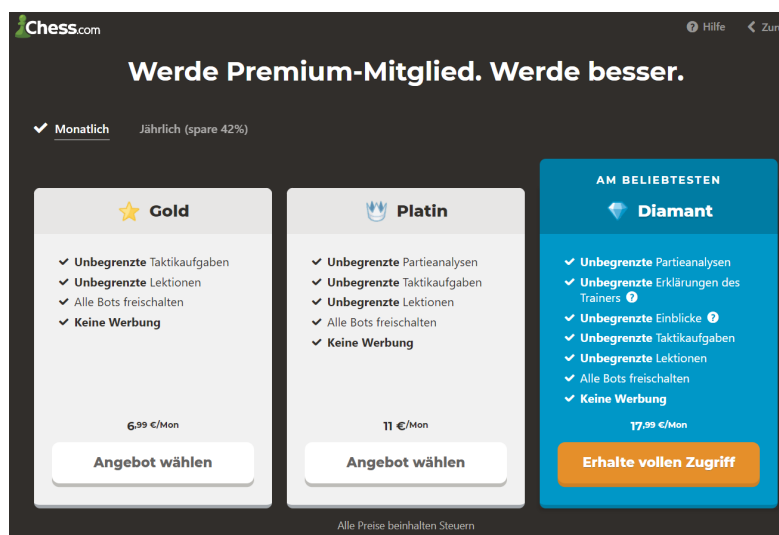
<sup>7</sup>Quelle: <https://www.youtube.com/@chess> am 12. Mai 2023

<sup>8</sup>Quelle: <https://www.twitch.tv/chess> am 12. Mai 2023



Quelle: <https://www.chess.com/home> am 12. Mai 2023

Abbildung 1.2: Home-Bildschirm von Chess.com



Quelle: <https://www.chess.com/membership?c=navbar> am 12. Mai 2023

Abbildung 1.3: Abonnement Möglichkeiten von Chess.com

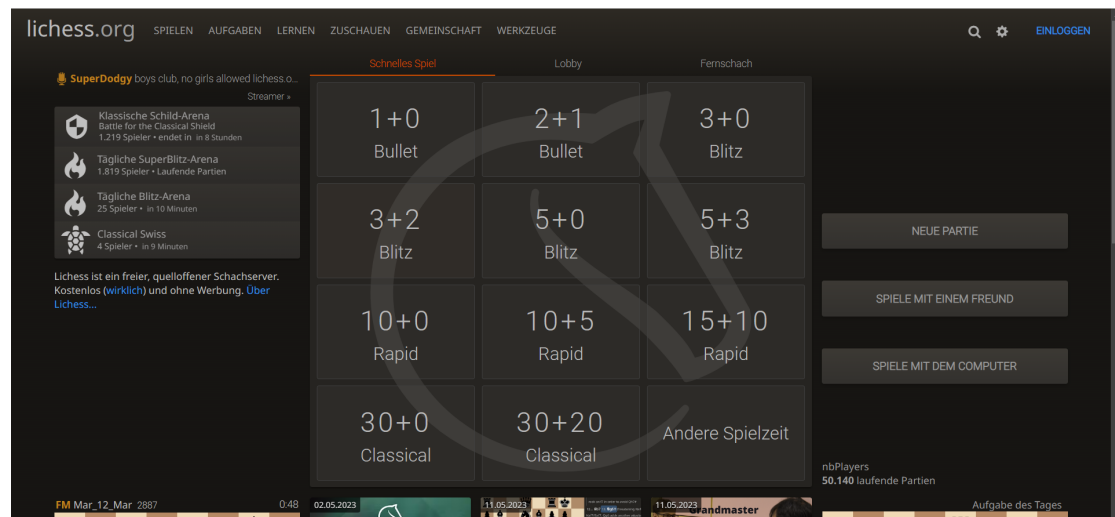
## 1.4.2 Lichess

Lichess wirbt vor allem damit, dass es komplett kostenfrei verwendbar ist, keine Werbung hat und keine Registrierung nötig ist um zu spielen. Auch Funktionen wie Analysen eines Spiels sind für alle Benutzenden uneingeschränkt verfügbar. Die Finanzierung von [lichess.org](https://lichess.org) basiert auf Spenden und die Entwicklung übernehmen Freiwillige<sup>9</sup>. Der

<sup>9</sup>Quelle: <https://lichess.org/about> am 12. Mai 2023

gesamte Code ist Open-Source und für jeden einsehbar.

Lichess ist dunkel und modern gehalten. Die Funktionen die Lichess anbietet, sind weniger umfangreich als die von Chess.com und sind auf dem Desktop zum Teil schwer zugänglich, während sie auf einem Mobil-Gerät in der App gar nicht verfügbar sind. Dazu zählen auch soziale Interaktionen wie Gruppen. Sie sind wenig ausgebaut und schwierig, bis gar nicht, auffindbar.



Quelle: lichess.org am 12. Mai 2023

Abbildung 1.4: Der Lichess Desktop Home-Bildschirm





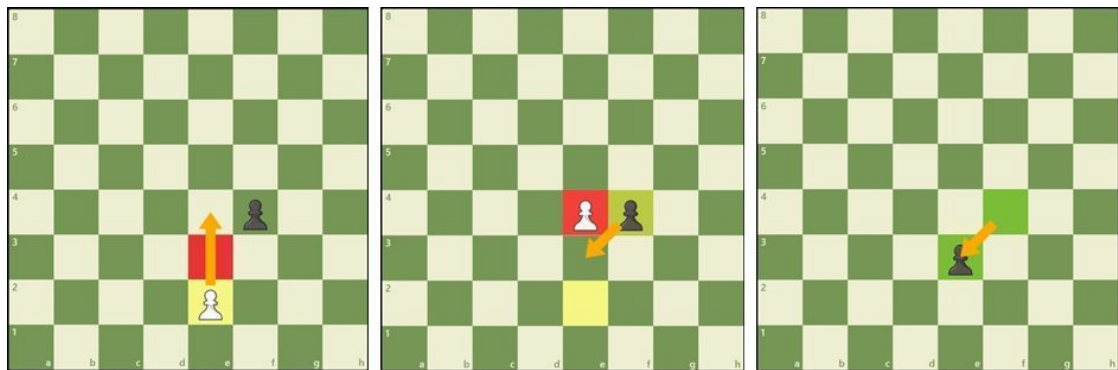
## 2 Theoretische Grundlagen

### 2.1 Schach

Schach ist ein strategisches Brettspiel für zwei Spielende, welches auf einem quadratischen Spielfeld mit 64 Feldern gespielt wird. Jede/-r Spieler/-in 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/die Gegner/-in den Angriff verhindern kann.

Wie Figuren sich bewegen und andere Figuren schlagen wird nicht explizit erklärt, lediglich die Schachuhren und zwei Sonderregeln des Schachs werden genauer erläutert, 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).



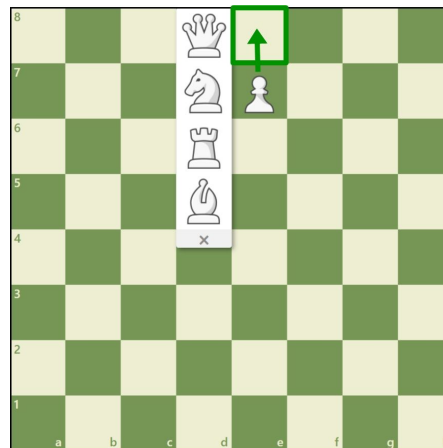
Quelle: <https://www.chess.com/de/schachregeln>

Abbildung 2.1: Die Zusatzregel *en passant*

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).

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 in Minuten, die jedem Spielenden zur Verfügung steht. Die zweite Zahl, hier 5, wird als Inkrement bezeichnet. Dies bedeutet, dass nach jedem Zug diesem Spielenden 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 Spielenden verstreichen zu lassen. Dies gewährleistet, dass die Uhren erst zu laufen beginnen, wenn beide Spielenden 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, die erstmals 2009 angekündigt wurde<sup>1</sup> und speziell für die Entwicklung von skalierbaren Netzwerkanwendungen konzipiert ist [Fou23]. Skalierbarkeit bedeutet, dass mit steigender Anzahl von Benutzenden 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 [Pre15]. 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 Codeausschnitt 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. [Pre15] Die Vorteile hinsichtlich der Skalierbarkeit werden allerdings erst bei einer hohen Anzahl von Zugriffen erkennbar.

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

<sup>1</sup>Quelle: <https://www.youtube.com/watch?v=EeYvF17li9E> am 22. April 2023

```

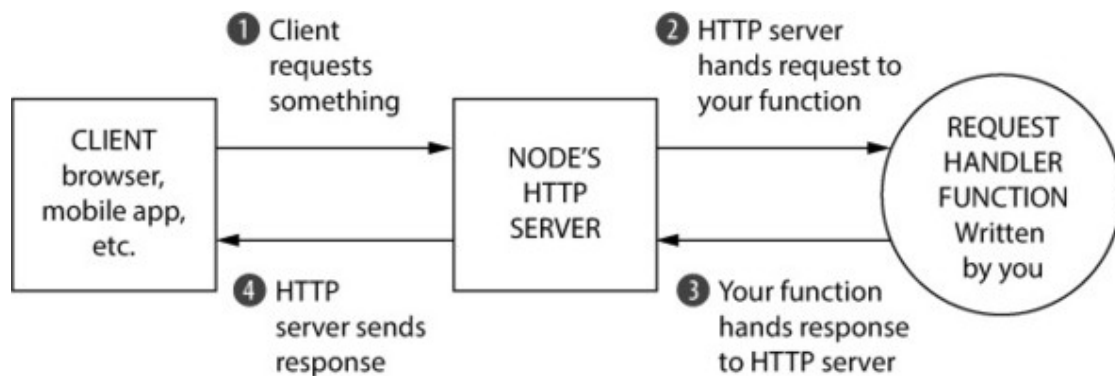
2 | result...
3 | });

```

Codeausschnitt 2.1: Beispiel einer Callback Funktion **Quelle:** [Pre15]

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/die Frontend-Entwickler/-in nicht alles was der/die Backend-Entwickler/-in macht und umgekehrt, aber es gibt 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 zurücksenden von einem Result mittels dem HTTP-Protokoll. Das HTTP-Protokoll verwendet verschiedene Methoden wie GET, POST, PUT und DELETE, um unterschiedliche Aktionen durchzuführen [Hah16]. 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.



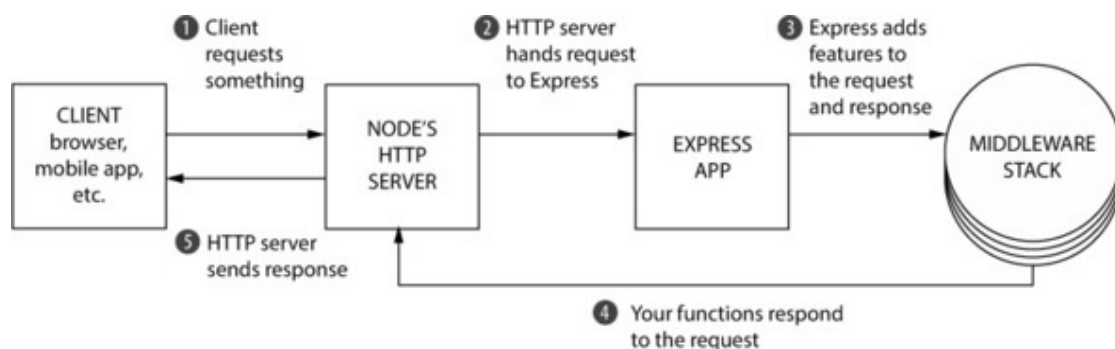
**Quelle:** [Hah16]

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

## Express

Express ist ein leichtgewichtiges und sehr beliebtes Web-Framework, welches unter Node.js zur Verfügung steht. Die API von Node.js wird durch Express vereinfacht und es stellt hilfreiche Funktionen bereit [Hah16]. Beispielsweise ermöglicht es die Verwendung von Middleware und Routing.

**Middlewares** sind Funktionen, die ermöglichen, 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). Dabei gibt es eine von uns definierte Reihenfolge der Middlewares. Zum Beispiel kann man definieren, dass zu erst der Request von einer Middleware geloggt werden soll, anschließend soll die benutzende Person authentifiziert werden und falls sie einen Pfad aufrufen möchte, für die sie 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 [Hah16].



Quelle: [Hah16]

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

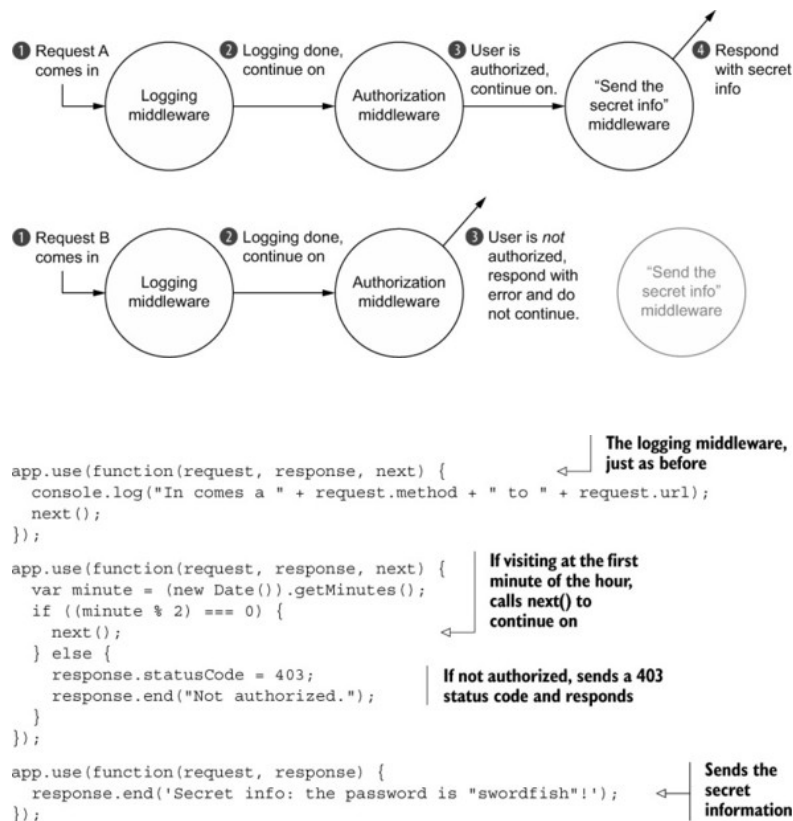
**Routing** hilft dabei zu identifizieren bei welchem Request welche Middleware ausgeführt werden soll (siehe Abbildung 2.6). Beispielsweise kann eine Anfrage an den Pfad `/auth` mit den angegebenen Anmeldedaten des Benutzenden gesendet werden. Unter diesem Pfad kann man dann bestimmte Middlewares verwenden, welche sich mit der Authentifizierung des Benutzenden befassen [Hah16].

### 2.2.2 Socket.io

Die Kommunikation über das HTTP-Protokoll von Express hat den Nachteil, dass der Server von sich aus keine Daten an einen Client senden kann, sondern nur Daten aufgrund einer Anfrage dem Client übermitteln werden können. Diese Problematik löst das Framework Socket.io [Pre15].

Es ermöglicht eine direkte, bidirektionale Echtzeitübertragung von Daten mittels Websockets, long-polling und anderen Protokollen zwischen den Clients und dem Server<sup>2</sup>.

<sup>2</sup>Quelle: <https://socket.io/docs/v4/> am 15. Mai 2023



Quelle: [Hah16]

Abbildung 2.5: Beispiel der Nutzung von Middlewares

Diese Echtzeitkommunikation ist für viele Spiele, wie auch für diese Schach-App, essenziell. 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 [Pre15].

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 daraufhin eine Funktion mit den übertragenen Daten ausführen. Events können mittels Sockets gesendet und empfangen werden, welche als Schnittstelle der Kommunikation von Socket.io dienen. 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

```

app.get("/about", function(request, response) {
  response.end("Welcome to the about page!");
});

app.get("/weather", function(request, response) {
  response.end("The current weather is NICE.");
});

app.use(function(request, response) {
  response.statusCode = 404;
  response.end("404!");
});

http.createServer(app).listen(3000);

```

Called when a request to /about comes in

Called when a request to /weather comes in

If you miss the others, you'll wind up here.

Quelle: [Hah16]

Abbildung 2.6: Beispiel der Nutzung von Routing

man beliebig viele Parameter übertragen mit denen die Callback-Funktion des Listeners aufgerufen wird. [Pre15]

```

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");

```

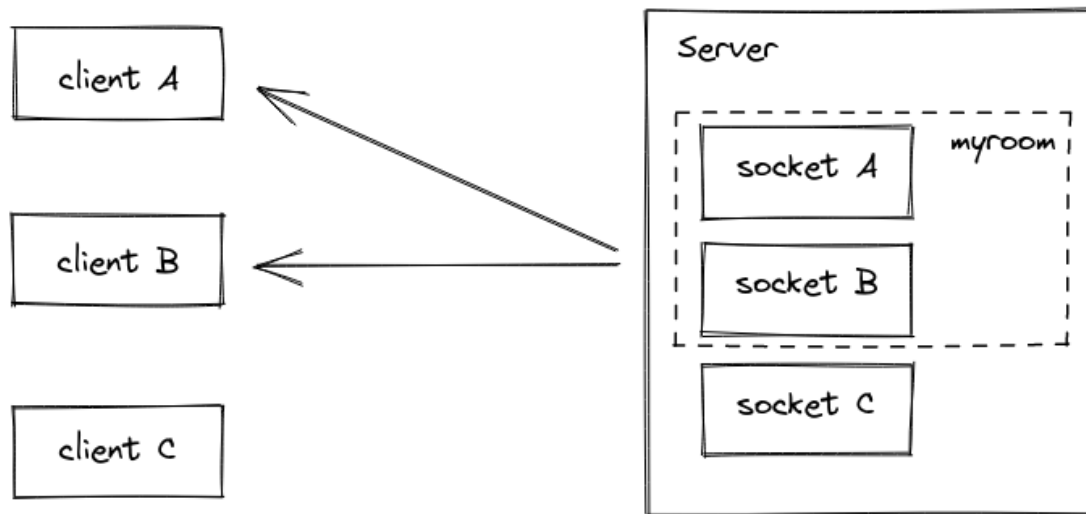
Quelle: [Soc23]

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

Ein wichtiges Feature von Socket.io sind die Räume<sup>3</sup>. 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 Codeausschnitt 2.2).

Des weiteren erhält jede Socket beim Verbinden eine eigene ID, die ebenfalls als Raum ge-

<sup>3</sup>Quelle: <https://socket.io/docs/v4/rooms/> am 22. April 2023



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

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

nutzt 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 übergebenen Argumente die Socket und die next Funktion als nächste Middleware. Es bietet sich daher an, die Authentifizierung oder die Initialisierung von Listnern 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 ...

```

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

### 2.2.3 React

#### React

React ist eine der beliebtesten (nach einer Umfrage von 2022 von Stack Overflow sogar die beliebteste<sup>4</sup>) 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 ermöglicht React mittels seiner *Hooks* [Sch22].

Die Art und Weise wie React und seine Hooks funktionieren wird an dem Beispiel 2.3 erklärt. In diesem Beispiel wird die Komponente *ExampleComponent* implementiert, die beispielsweise mittels des Tags

```
<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.

```
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  }, [count, loadingDelay]);
```

<sup>4</sup>Quelle: <https://survey.stackoverflow.co/2022/#technology-most-popular-technologies> am 27. April 2023



```

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

```

Codeausschnitt 2.3: Beispiel einer React Komponente

Die Komponente hat den lokalen State *count*, welcher mittels der **useState**-Hook initialisiert wird. Ein State beschreibt einen Zustand der Komponente und eine Änderung veranlasst die Komponente neu zu rendern. 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) [Sch22].

Die Hook **useEffect** 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 [Sch22]. 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 verwendet die gleichen Argumente wie die *useEffect* Hook und durch sie kann man eine Funktion definieren, welche nur neu initialisiert wird, falls sich eine der Variablen im *Dependency Array* ändert. Würde man sie als reguläre JavaScript Funktion definieren, würde die Funktion immer neu initialisiert werden, wenn die Komponente gerendert wird [Sch22].

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 **createContext** erstellen und mit der **useContext**-Hook importieren [Sch22]. In unserem Beispiel wird er verwendet, um ein ThemeContext zu importieren, der die Hintergrundfarbe unserer Komponente bestimmt (siehe Zeile 35 in Beispiel 2.3).

In der Rückgabe der Komponente kann man nicht nur HTML, sondern auch JavaScript innerhalb von geschweiften Klammern verwenden. So wird in unserem Beispiel geprüft mit dem ternären Operator den Wert von *isLoading* und zeigen entsprechende Elemente an (siehe Zeilen 36-43, Beispiel 2.3).

## React Router

React Router ermöglicht die Erstellung von Anwendung mit mehreren Seiten unter verschiedenen Pfaden [Sch22]. Das ist sinnvoll, um als Benutzer/-in 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.

```

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 export default App;
25
26 function OrderDetail() {
27
28   const params = useParams();
29   const orderId = params.id;
30
31   useEffect(() => {
32     //fetch Data with orderId
33   }, [])
34
35   return (
36     // Show Data
37   );
38 }
39
40 export default OrderDetail;
```

Codeausschnitt 2.4: Beispiel von verschiedenen Komponenten auf verschiedenen Pfaden  
Quelle: [Sch22] (abgewandelt)

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 allerdings auch noch ein paar weitere Funktionen, wie zum Beispiel die Hooks **useParams()** und **useNavigate()** [Sch22].

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 (Beispiel 2.4).

## 2.2.4 PostgreSQL und Redis

### PostgreSQL

PostgreSQL ist ein objektrelationales Open-Source Datenbanksystem, welches erstmals 1989 veröffentlicht wurde [Le21]. Die Verwaltung von Datenbanken basiert auf soge-

nannte Datenbankmanagementsystemen (kurz DBMS). Das beliebteste DBMS für PostgreSQL ist *pgAdmin*<sup>5</sup>. 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 [Add21]:

- 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 bestimmte 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 [Add21]:

- *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 (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<sup>6</sup>.

<sup>5</sup>Quelle: <https://www.pgadmin.org/> am 22. April 2023

<sup>6</sup>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 [Nel16]. 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 an, 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<sup>7</sup>. Zu den möglichen Datentypen zählen Strings, Listen, Sets, Hashes, sortierte Sets, Streams und einige weitere<sup>7</sup>. Zudem ermöglicht Redis eine gute vertikale Skalierbarkeit indem mehrere Redis Instanzen verbunden werden<sup>8</sup>.

### 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<sup>9</sup>. Diesen Informationen kann vertraut werden, da sie mittels eines privaten serverseitigen secrets mit verschiedenen Algorithmen verschlüsselt (auch *signiert* genannt) und wieder entschlüsselt werden (siehe Abbildung 2.9). Der meist verwendete Anwendungsbereich für JSON Web Tokens ist die Authentifizierung<sup>10</sup>, bei der der vom Backend generierte Token in einer Session oder einem Cookie gespeichert werden kann. 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 clientseitig manipulierbar sein darf, da dies ein Sicherheitsrisiko darstellen kann. Beispielsweise kann das mit einem HTTP-Only Cookie<sup>11</sup> erreicht werden.

#### Chakra UI

Chakra UI ist eine simple Komponenten-Bibliothek, welche das Designen von React Anwendungen vereinfacht<sup>12</sup>. Es stellt Komponenten zur Verfügung, welchen verschiedene Attribute zugeordnet werden können, um diese nach eigenem Ermessen zu designen (siehe Abbildung 2.10).

<sup>7</sup>Quelle: <https://redis.io/docs/about/> am 22. April 2023

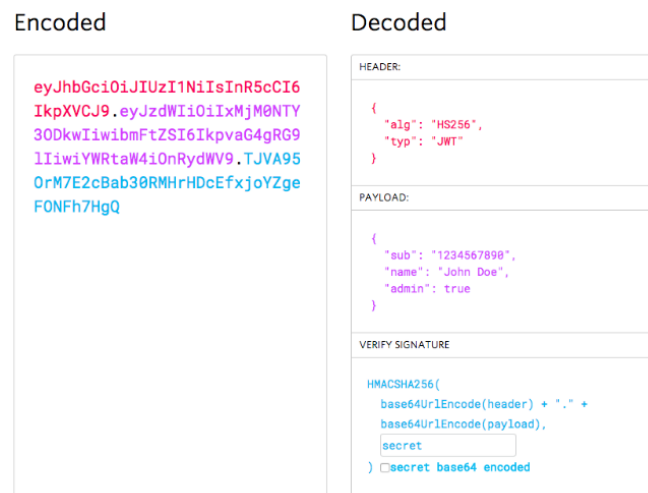
<sup>8</sup>Quelle: <https://redis.io/docs/management/scaling/> am 15. Mai 2023

<sup>9</sup>Quelle: <https://jwt.io/introduction> am 22. April 2023

<sup>10</sup>Quelle: <https://jwt.io/introduction> am 22. April 2023

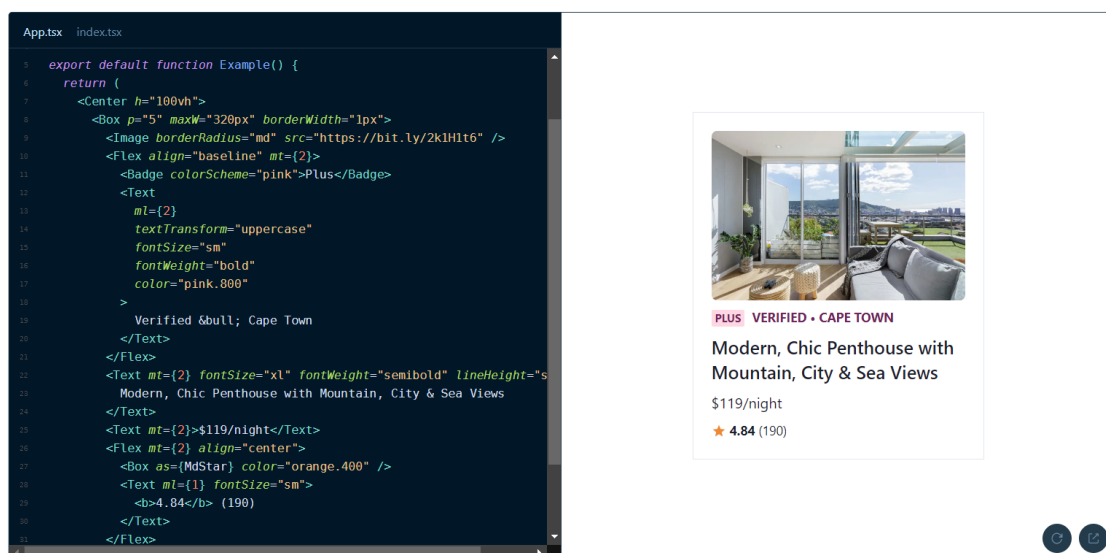
<sup>11</sup>Quelle: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies> am 22. April 2023

<sup>12</sup>Quelle: <https://chakra-ui.com/> am 22. April 2023



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: Beispiel einer mit Chakra-UI designten React-Komponente

## Formik und Yup

Formik ist die beliebteste Open-Source-Bibliothek für Formulare in React <sup>13</sup>. Sie vereinfacht die Handhabung von Formularen und bietet Funktionen wie die Validierung eingegebener Werte, Fehlermeldungen und Unterstützung für mehrstufige Formulare.

---

<sup>13</sup>Quelle: <https://formik.org/> am 22. April 2023

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<sup>14</sup>.

Die Integration von Yup-Schemata in Formik-Formularen ist gut unterstützt, was eine einfache Handhabung von Überprüfungen und Fehlerbehandlungen bei Eingaben des Benutzenden ermöglicht.

### **chess.js**

chess.js ist eine Schach Bibliothek, welche die gesamte Schachlogik zur Verfügung stellt<sup>15</sup>. Sie 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<sup>16</sup>- oder PGN<sup>17</sup>-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<sup>18</sup>. Es bietet zahlreiche Konfigurationsoptionen, 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 auf diesem Feld implementiert werden können.

### **bcrypt**

bcrypt ist eine Bibliothek welche entwickelt wurde um Passwörter zu verschlüsseln. Sie 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. bcrypt löst dieses Problem dadurch, dass es die Passwörter um einen selbst definierbaren Faktor zeitlich länger kodiert, indem mehrere Iterationen durchgeführt werden<sup>19</sup>.

<sup>14</sup>Quelle: <https://github.com/jquense/yup> am 22. April 2023

<sup>15</sup>Quelle: <https://github.com/jhlywa/chess.js/> am 22. April 2023

<sup>16</sup>Quelle: <https://de.wikipedia.org/wiki/Forsyth-Edwards-Notation> am 22. April 2023

<sup>17</sup>[https://de.wikipedia.org/wiki/Portable\\_Game\\_Notation](https://de.wikipedia.org/wiki/Portable_Game_Notation) am 22. April 2023

<sup>18</sup>Quelle: <https://github.com/lichess-org/chessground> am 22. April 2023

<sup>19</sup>Quelle: <https://auth0.com/blog/hashing-in-action-understanding-bcrypt/>





## 3 Systemarchitektur und Konzeption

In diesem Kapitel wird die Systemarchitektur der Anwendung vorgestellt, indem erläutert wird wie die verschiedenen Komponenten und Technologien zusammenarbeiten und miteinander kommunizieren. Die vorhandenen Funktionen der Schach-App werden mit Bildschirmfotos und Erklärungen veranschaulicht und verschiedene Abläufe werden durch Aktivitäts- und Sequenzdiagramme dargestellt.

### 3.1 Einführung

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 oder der Benutzerin verantwortlich, während das Backend die Suche nach Spielenden, die Verwaltung der Benutzerdaten und Freunde und die Echtzeit-Kommunikation zwischen den Spielenden steuert.

Die Anwendung verwendet moderne Web-Technologien, um eine reaktive und benutzerfreundliche Oberfläche zu schaffen. Das Frontend basiert auf dem React-Framework (siehe Kapitel 2.2.3), das es ermöglicht, wiederverwendbare Komponenten zu entwickeln und den Anwendungsstatus effizient zu verwalten. Das User-Interface basiert auf Chakra UI (siehe Kapitel 2.2.5), einem modernen und flexiblen Komponenten-Bibliothekssystem, das die Entwicklung von responsiven und zugänglichen Benutzeroberflächen erleichtert. Auf der Backend-Seite wird Node.js mit dem Express-Framework (siehe Kapitel 2.2.1) verwendet, um einen leistungsstarken und skalierbaren Server bereitzustellen. Die API-Endpunkte und die Echtzeitkommunikation mittels Socket.io (siehe Kapitel 2.2.2) 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-Datenbank (siehe Kapitel 2.2.4) verwendet, die aufgrund ihrer Unterstützung für strukturierte Daten und ihrer Robustheit in der Handhabung von Benutzerdaten ausgewählt wurde. Die Verwaltung von befreundeten Person und Daten aktiver Spiele werden in einer Redis-Datenbank gespeichert, die sich durch hohe Leistung und niedrige Latenz auszeichnet, insbesondere bei Lese- und Schreibvorgängen. Redis 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 die Modularität und Trennung der verschiedenen Funktionsbereiche.

## 3.2 Architekturübersicht

Das Komponentendiagramm in Abbildung 3.1 visualisiert den Zusammenhang der Hauptkomponenten für die Kommunikation.

Wie zu erkennen ist, sind die Datenbanken und deren Funktionen getrennt gehalten. Die Web-API und die PostgreSQL Datenbank werden zum Authentifizieren genutzt, während die Kommunikation mittels Socket.io und die Redis Datenbank für die restlichen Funktionen der Anwendung verwendet werden. Dies erhöht die Modularität und Flexibilität, wodurch die verschiedenen Datenbanken und Funktionen unabhängig voneinander entwickelt, gewartet und skaliert werden können.

Im Frontend gibt es drei Komponenten, welche die Web-API verwenden: Der *UserContext*, *Login* und *SignUp*. Die 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 verschiedene 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 Benutzenden verwendet.

Beim Herstellen einer Socket.io Verbindung in *SocketContext* werden im Backend die Middlewares aus *socketMiddleware* ausgeführt und die Listener aus *socketController* und *socketChessController* initialisiert. Der Unterschied zwischen den Listnern der beiden Dateien ist, 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/-e Spieler/-in vorübergehend keine Internetverbindung beim senden oder beim empfangen von Events hat?

Um den Server zu entlasten wäre natürlich eine clientbasierte Lösung ideal, bei der mit einem Zug auch die jeweilige aktuelle Zeit gesendet wird. Wenn ein/-e Spieler/-in jedoch eine schlechte Internetverbindung hat und daher die Züge im Backend und bei der anderen spielenden Person verzögert ankommen, kann es zu erheblichen Unterschieden

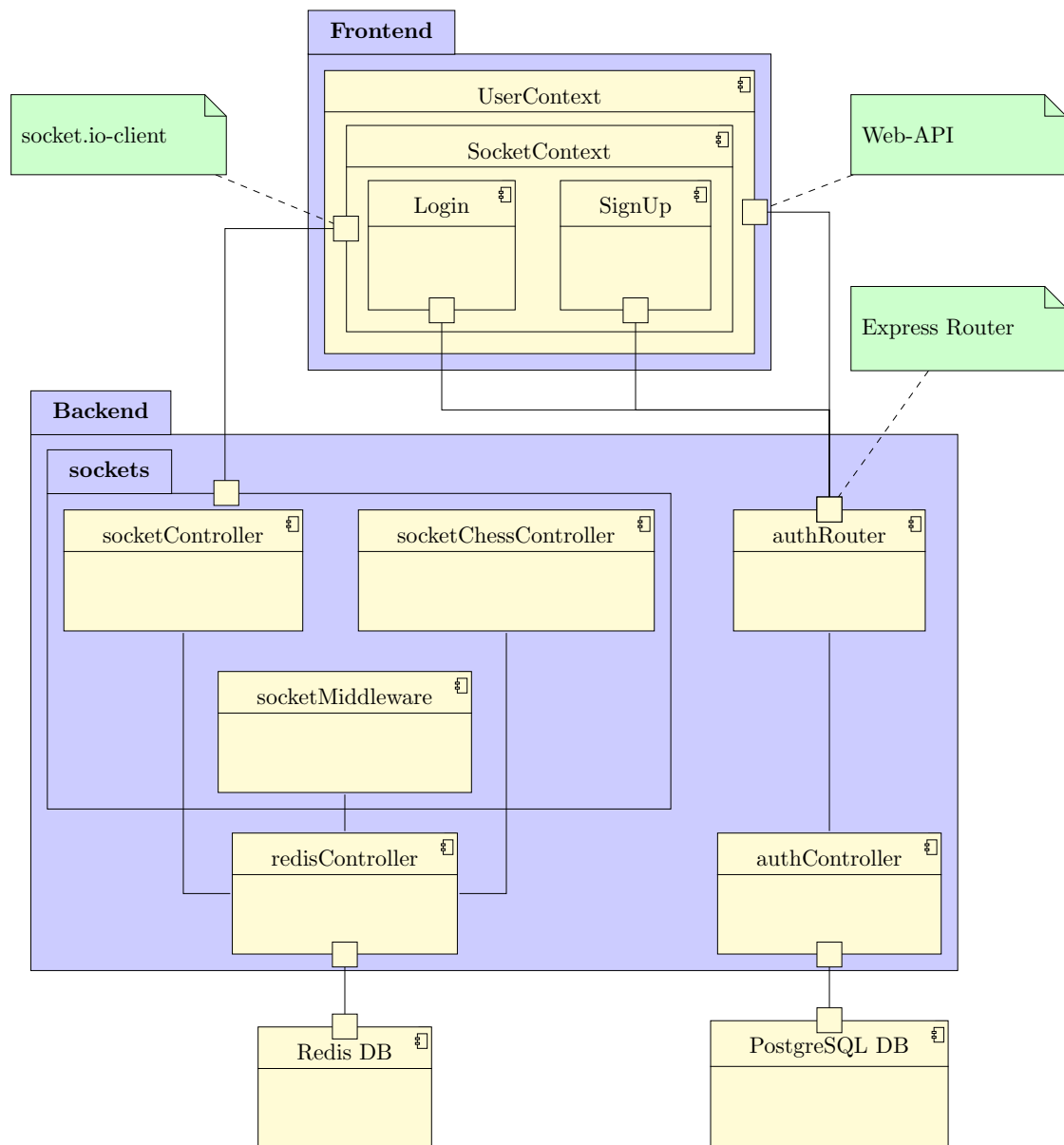


Abbildung 3.1: Komponentendiagramm der Anwendung

in den angezeigten Zeiten kommen. Es stellt sich die Frage, welche dieser Zeiten gültig ist.

Um dieses Problem zu umgehen liegt die einfachste Lösung darin, eine serverseitige 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 clientseitiger Code im Browser manipulierbar sein kann und dadurch die Schachuhren beeinflussbar wären.

Wie die Schachuhren konkret funktionieren wird in den nachfolgenden Kapiteln behandelt und eine Kritik und alternative Implementierungen werden in Kapitel 5.2.1 diskutiert.

## 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 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 Benutzende. Nicht angemeldete Benutzende 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.

```
client/
├── public/
│   ├── Gambit dark.png
│   ├── Gambit light.png
│   ├── Gambit springer.png
│   ├── index.html
│   ├── manifest.json
│   └── robots.txt
├── src/
│   ├── components/
│   │   ├── ActiveGames.js
│   │   ├── AddFriendModal.js
│   │   ├── Chat.js
│   │   ├── ChessClock.js
│   │   ├── Friend.js
│   │   ├── FriendList.js
│   │   ├── FriendRequest.js
│   │   ├── GameRequests.js
│   │   ├── Navbar.js
│   │   └── PromotionModal.js
│   ├── contexts/
│   │   ├── AccountContext.js
│   │   └── SocketContext.js
│   ├── themes/
│   │   └── Theme.js
│   ├── utils/
│   │   └── ChessLogic.js
│   ├── views/
│   │   ├── ChessGame.js
│   │   ├── Home.js
│   │   ├── Login.js
│   │   └── Signup.js
│   ├── App.js
│   ├── index.js
│   └── Views.js
└── package.json
```

Abbildung 3.2: Ordnerstruktur des Frontends

- **AccountContext:** Stellt Informationen über den Benutzerstatus allen Folgenden Komponenten mittels eines React-Contexts namens *UserContext* zur Verfügung. Diese Informationen beinhalten, ob eine benutzende Person angemeldet ist und falls sie das ist ihren Benutzernamen.

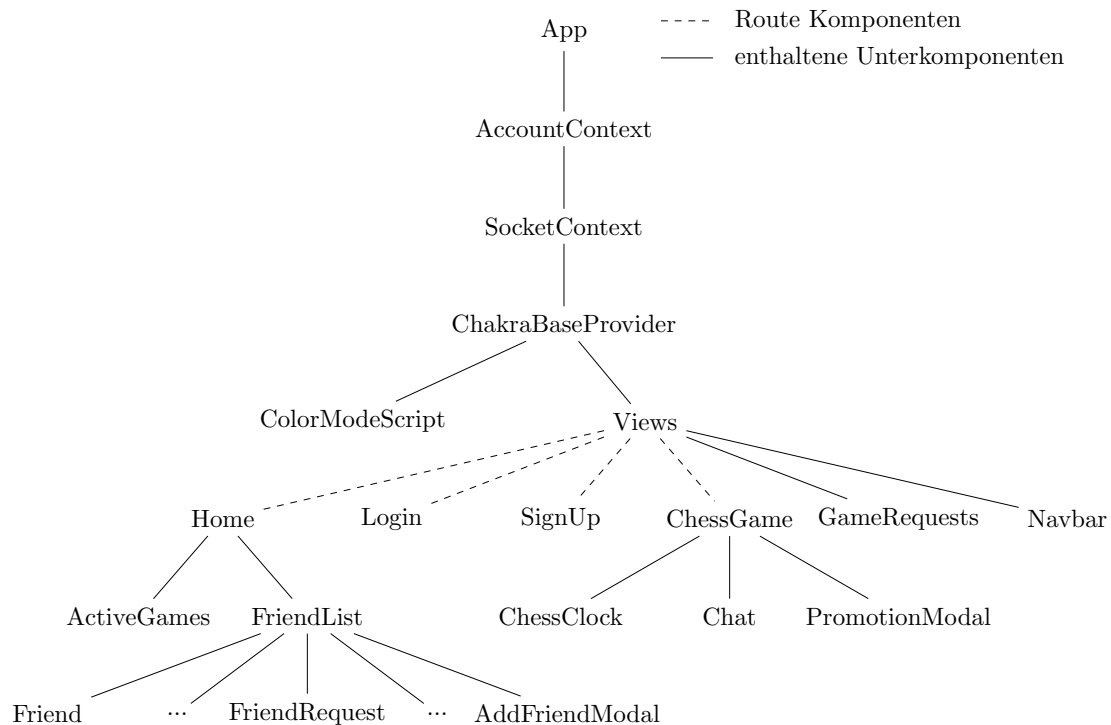


Abbildung 3.3: Aufbau der React-Komponenten für angemeldete Benutzer

- **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 Farbschemas (dunkel oder hell) oder das Zugreifen auf definierte Stile.
- **Views:** Mit Hilfe des React-Routers wird in dieser Komponente 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/-e Benutzer/-in 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](http://lichess.org) (siehe Abbildung 1.4)) bereits gängig und benötigen keine zusätzliche Erklärung. Diese Knöpfe sollen es möglichst schnell und einfach machen ein Spiel starten zu können.

Ist ein/-e Benutzer/-in 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 Benutzenden, während die Darstellung und Interaktion die Unterkomponenten *Friend* und *FriendRequest* übernehmen.
  - \* **Friend:** Übernimmt die Darstellung einer befreundeten Person. Mittels eines farbigen Punktes ist erkennbar, ob sie gerade online ist (grün) oder nicht (rot) (siehe Abbildung 3.5) (Mit online ist gemeint, dass die Person derzeit die Anwendung offen hat). Ist sie online erscheint noch mindestens ein weiterer Button. Es beinhaltet ein Icon in Form von gekreuzten Schwertern und einem Schild. Dies hat die Funktion eine befreundete Person zu einem Spiel herauszufordern. Falls sie gerade ein aktives Spiel hat erscheint noch ein zweiter Button mit einem Auge als Icon, welches den Benutzenden zu dem aktiven Spiel der Freundin, beziehungsweise 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 (siehe Abbildung 3.17).
- **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). Diese Komponenten befinden sich unter dem Pfad `/login`, beziehungsweise `/signup`.
- **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 in einem eigenen Kasten rechts 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).

- **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 Spielenden kommunizieren können. Zuschauer/-innen können ihn lesen, allerdings nichts schreiben, da sie Tipps geben könnten.
- **PromotionModal:** Über dieses Modal lässt sich bei einer Bauernumwandlung eine Figur auswählen, welche den Bauern ersetzen soll (siehe Abbildung 3.10).

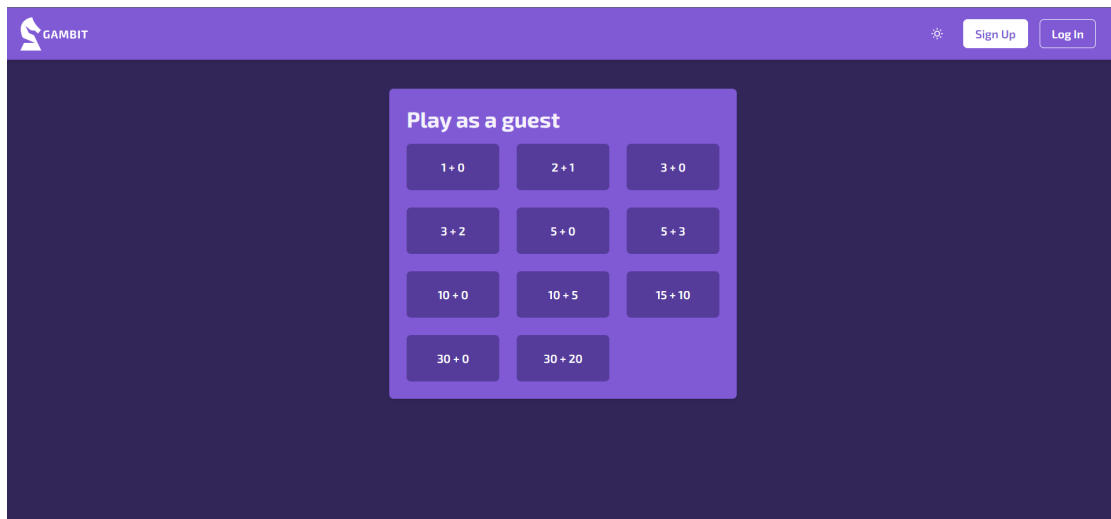


Abbildung 3.4: Home und Navbar Komponente eines nicht angemeldeten Benutzenden im dunklen Farbschema

### 3.4.2 Authentifizierung

Die Authentifizierung findet in drei Komponenten statt: Dem *UserContext* und den *Login*- und *SignUp*-Komponenten. Die Authentifizierung im Backend, wird in Abschnitt 3.5.1 erläutert. Konkrete Einzelheiten und Code-Beispiele werden in Abschnitt 4.1.1 vorgestellt.

Der *UserContext* beinhaltet den State `user`, der 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



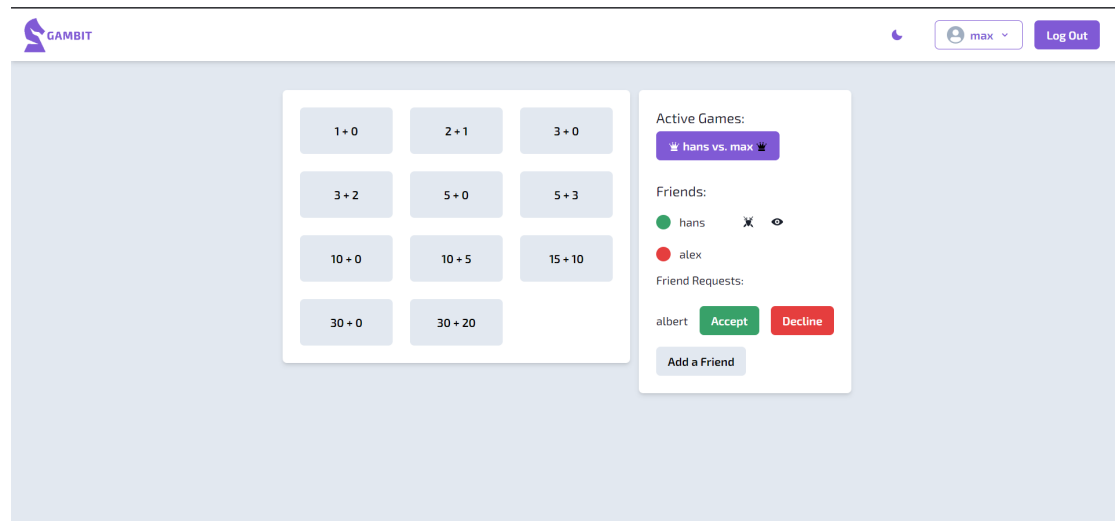


Abbildung 3.5: Home und Navbar Komponente eines angemeldeten Benutzenden im hellen Farbschema

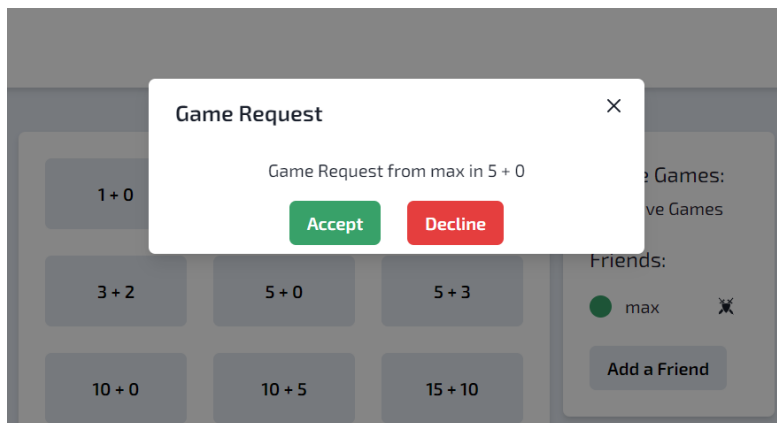
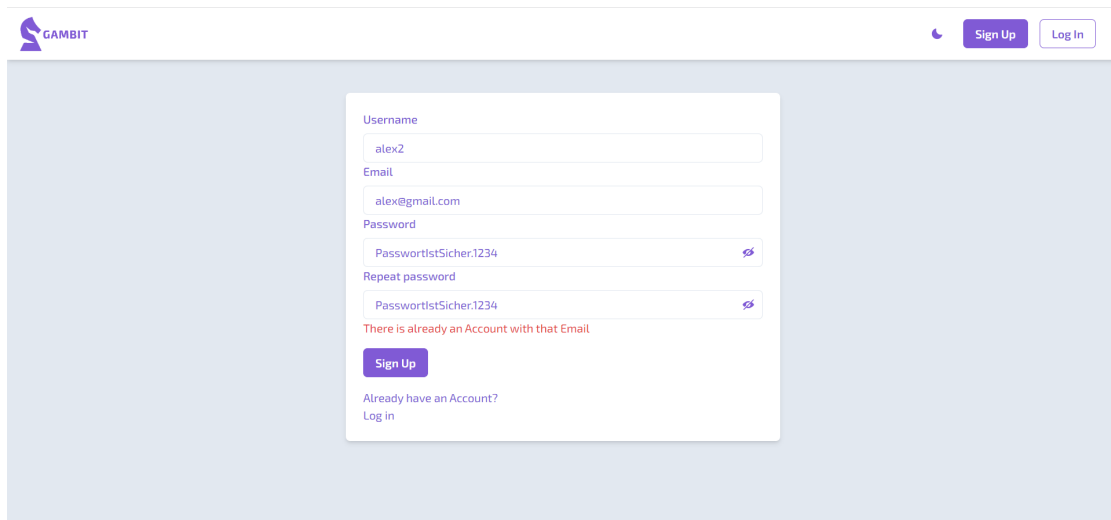
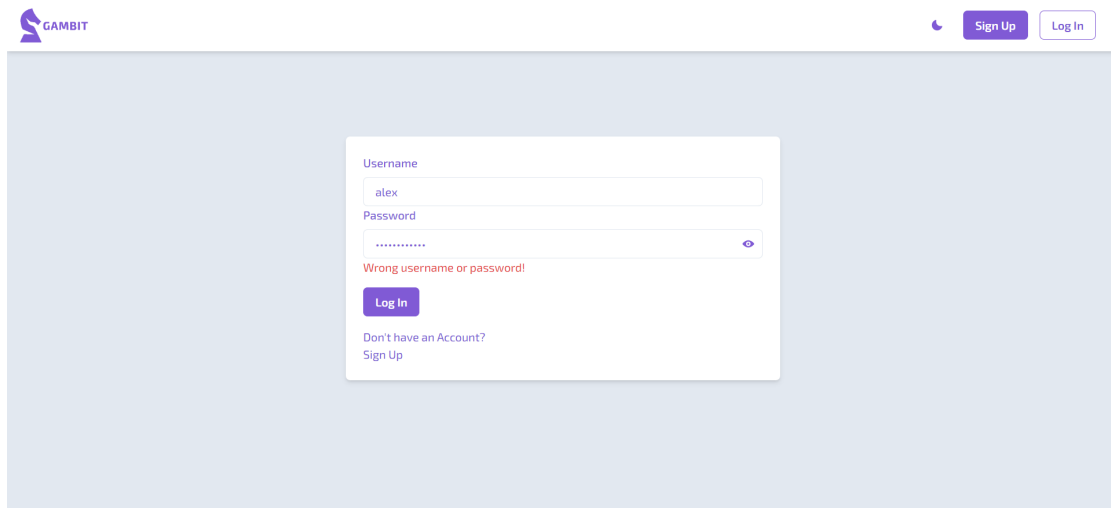


Abbildung 3.6: Das Modal der Komponente GameRequest in hellem Farbschema



The screenshot shows a web application interface for a sign-up process. At the top left is the 'GAMBIT' logo. At the top right are two buttons: 'Sign Up' (highlighted in purple) and 'Log In' (white with a purple border). The main content area features a white sign-up form with the following fields: 'Username' (containing 'alex2'), 'Email' (containing 'alex@gmail.com'), 'Password' (containing 'PasswortIstSicher:1234'), and 'Repeat password' (containing 'PasswortIstSicher:1234'). Each password field has an eye icon for toggling visibility. Below the 'Repeat password' field, a red error message states: 'There is already an Account with that Email'. At the bottom of the form is a purple 'Sign Up' button. Below the button, the text 'Already have an Account?' is followed by a link 'Log In'.

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



The screenshot shows the same web application interface as in the previous image, but with the login form displayed. The 'Username' field contains 'alex' and the 'Password' field contains a masked password '\*\*\*\*\*'. A red error message below the password field reads: 'Wrong username or password!'. A purple 'Log In' button is positioned below the error message. At the bottom of the form, the text 'Don't have an Account?' is followed by a link 'Sign Up'.

Abbildung 3.8: Beispiel einer Login-Komponente mit ungültigen Daten

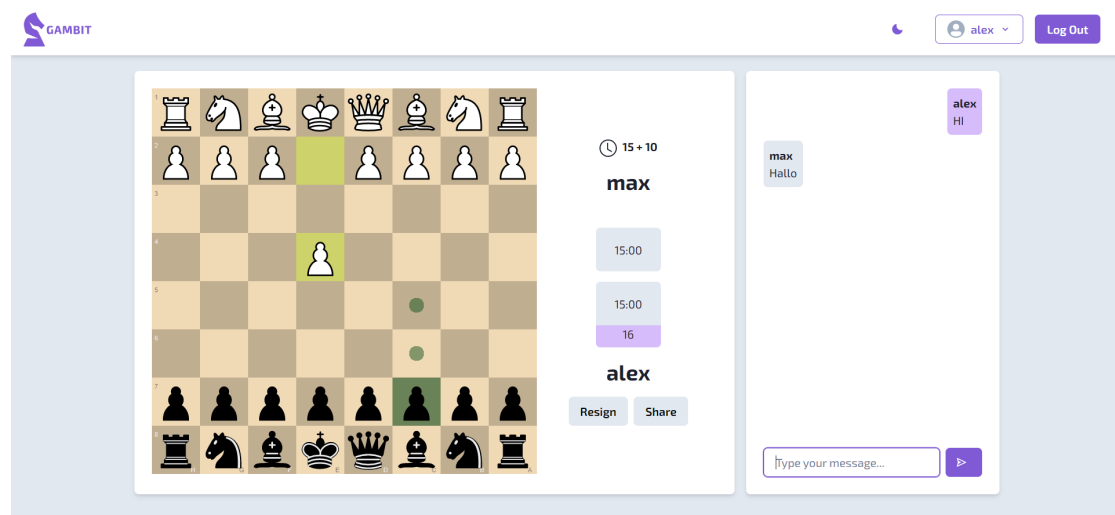


Abbildung 3.9: Beispiel einer ChessGame-Komponente

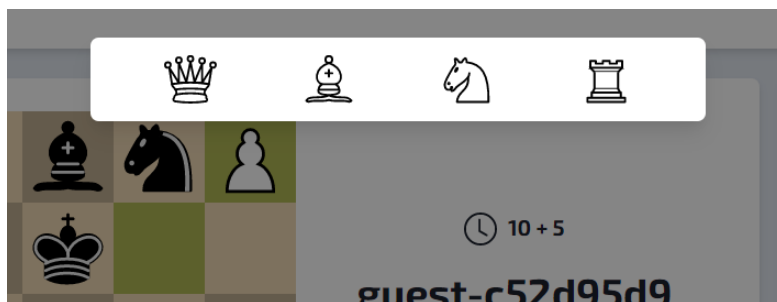


Abbildung 3.10: Das Modal der Komponente PromotionModal

sendet der Server die Antwort, ob die benutzende Person angemeldet werden konnte oder nicht und wenn ja, dann ihren Benutzernamen. Dies wird in den `user`-State gesetzt. 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 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).

### 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 das Zusammenspiel bei der Gegnersuche befindet sich in Abschnitt 3.5.2.

Das Schachspiel und die zugehörigen Schachuhren sind getrennt gehalten um die Modularität und Wartbarkeit zu erhöhen. Die Schachuhren und das Spiel haben jeweils eigene Events die sie empfangen und senden und kommunizieren nur bei der Initialisierung miteinander.

#### 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.

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 abubrechen (siehe Abbildung 3.11). Bei solch einem Abbruch wird das Event `leave_queue` mit der Zeitkonfiguration gesendet.

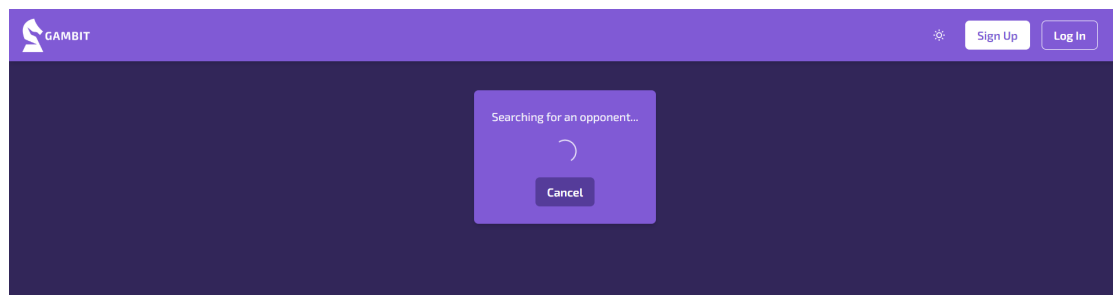


Abbildung 3.11: Lade-Bildschirm beim Starten eines Spiels mit den Buttons der *Home*-Komponente

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 *GameRequest*-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 der Herausforderer ist. 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 Spielenden, 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/die Benutzer/-in darauf hingewiesen (siehe Abbildung 3.12).

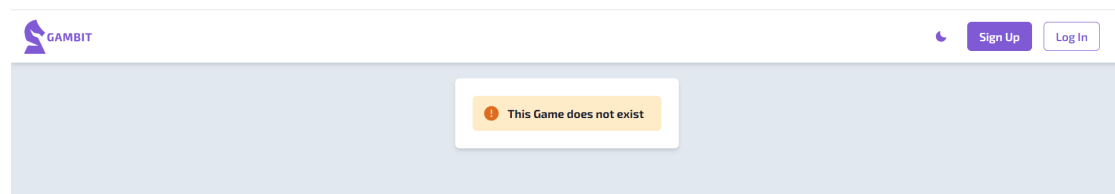


Abbildung 3.12: Benachrichtigung, falls kein Spiel unter der *roomId* gefunden werden konnte

Die Daten die gesendet werden umfassen folgendes:

- Die Namen des weißen und des schwarzen Spielenden.
- Welcher Zeitmodus gespielt wird (z.B.: 5 + 3, 10 + 5, ...)
- Die aktuelle Stellung der Partie
- Die bisherigen Nachrichten im Chat.

- Der aktuelle Status der Schachuhr: Hierbei gibt es vier mögliche Zustände – entweder läuft die Startzeit von Schwarz, die Startzeit von Weiß, die reguläre Spielzeit von Schwarz oder die reguläre Spielzeit von Weiß
- Die aktuellen Zeiten der Spielenden.

Dieses Einholen der Informationen wird für den Fall benötigt, dass die Seite geladen wird, nachdem das Spiel bereits in Gang ist. Außerdem kann dadurch jede Person, die den Link der Partie eingibt oder durch einen Freund dort landet der Partie zuschauen. Aufgrund der gesendeten Namen der Spielenden wird entschieden, ob man eine zuschauende oder eine spielende Person ist. Dafür werden die Benutzernamen mit dem eigenen Benutzernamen im *UserContext* verglichen. Doch was passiert wenn man eine Partie als unangemeldete/r Benutzer/-in spielt und deshalb keinen Benutzernamen im *UserContext* hat?

Bei dem Event `joined_game` (siehe Abschnitt 3.4.3) wird der Gast-Benutzername des Spielenden gesendet und in `location.state` gesetzt<sup>1</sup>. Dadurch kann in der *ChessGame*-Komponente darauf zugegriffen werden und es kann überprüft werden, ob es sich um einen spielende oder zuschauende Person handelt.

Dem entsprechend wird auch bestimmt wie das Schachbrett, die Namen und die Schachuhren ausgerichtet sind. Ist man Zuschauer/-in 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 Zuschauer/-innen Tipps geben könnten.

Zum Spielen der Partie werden folgende Listener definiert:

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

Die Events `checkmate`, `time_over`, `draw`, `resigned` und `cancel_game` beschreiben alle das Ende des Schachspiels. In ihren Listnern 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 3.13).

Der Ablauf beim Empfangen eines neuen Zugs ist im Aktivitätsdiagramm in Abbildung 3.14 dargestellt.

---

<sup>1</sup>Quelle: <https://github.com/remix-run/history/blob/main/docs/api-reference.md#locationstate> am 04. Mai 2023

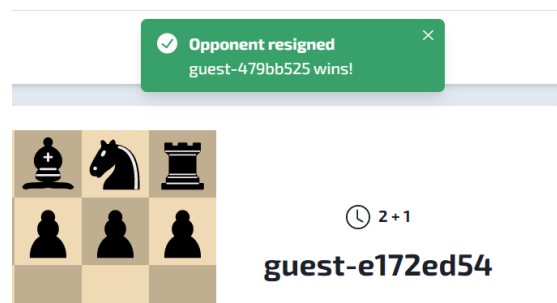


Abbildung 3.13: Beispiel eines Toasts, falls der Gegner aufgegeben hat

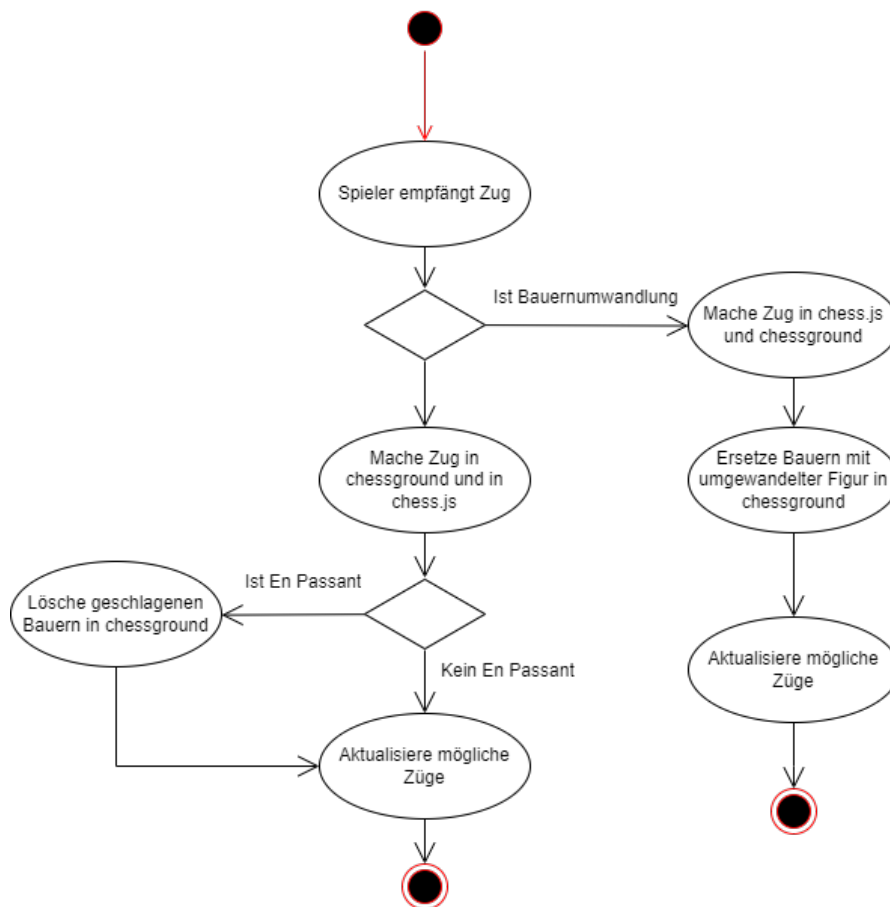


Abbildung 3.14: Aktivitätsdiagramm eines empfangenen Schachzugs

Die Bauernumwandlung und das en passant (siehe Kapitel 2.1) 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 und bewegt werden kann (siehe Abbildung 3.9).

Gesendet werden können die Events: **new\_move** zum senden eines Zugs, **resign** zum Aufgeben der Partie und **leave\_room**, wenn der/die Spieler/-in die *ChessGame* Komponente verlässt. Das **leave\_room**-Event ist insofern wichtig, als dass es mitverantwortlich ist, dass mehrere Spiele gleichzeitig gespielt werden können. Wenn der/die Spieler/-in eine Partie verlässt und eine zweite Partie startet und weiterhin die Events des ersten Spiels empfangen würde, kann es zu Problemen kommen, da die Listener der neuen Partie die Events der alten Partie empfangen. Durch das Verlassen des Raums wird gewährleistet, dass ein/-e Spieler/-in immer nur die Events zu dem Spiel empfängt, auf dem er/sie sich gerade befindet.

Ein Aktivitätsdiagramm des Sendens eines Zugs befindet sich in Abbildung 3.15. Genau wie bei dem Empfangen eines Zugs wird auch beim Senden zwischen Bauernumwandlung und en passant unterschieden. Ein 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.

## Die Uhr

Die *ChessClock*-Komponente bekommt von *ChessGame* als props die aktuelle Phase der Schachuhr, die jeweiligen aktuellen Zeiten und die Ausrichtung, welche Zeit oben beziehungsweise unten gezeigt werden soll. Neben der regulären Zeit gibt es noch eine Startzeit, welche während des ersten Zugs jedes Spielenden abläuft, um zu gewährleisten, dass das Spiel auch erst wirklich startet sobald beide Spielende bereit sind. Ist die Startzeit abgelaufen 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 Spielenden nach dem Zug und welche/-r Spieler/-in jetzt am Zug ist. Dementsprechend werden die Zeiten aktualisiert und die Uhr des Spielenden, welcher nun dran ist wird gestartet.
- **stop\_starting\_time\_white**: Stoppt die Startzeit des weißen Spielenden und startet die Startzeit des schwarzen Spielenden.
- **stop\_starting\_time\_black**: Stoppt die Startzeit des schwarzen Spielenden und lässt die reguläre Zeit des weißen Spielenden beginnen.
- **stop\_clocks**: Wird bei Beendung des Spiels empfangen und stoppt die aktive Uhr.



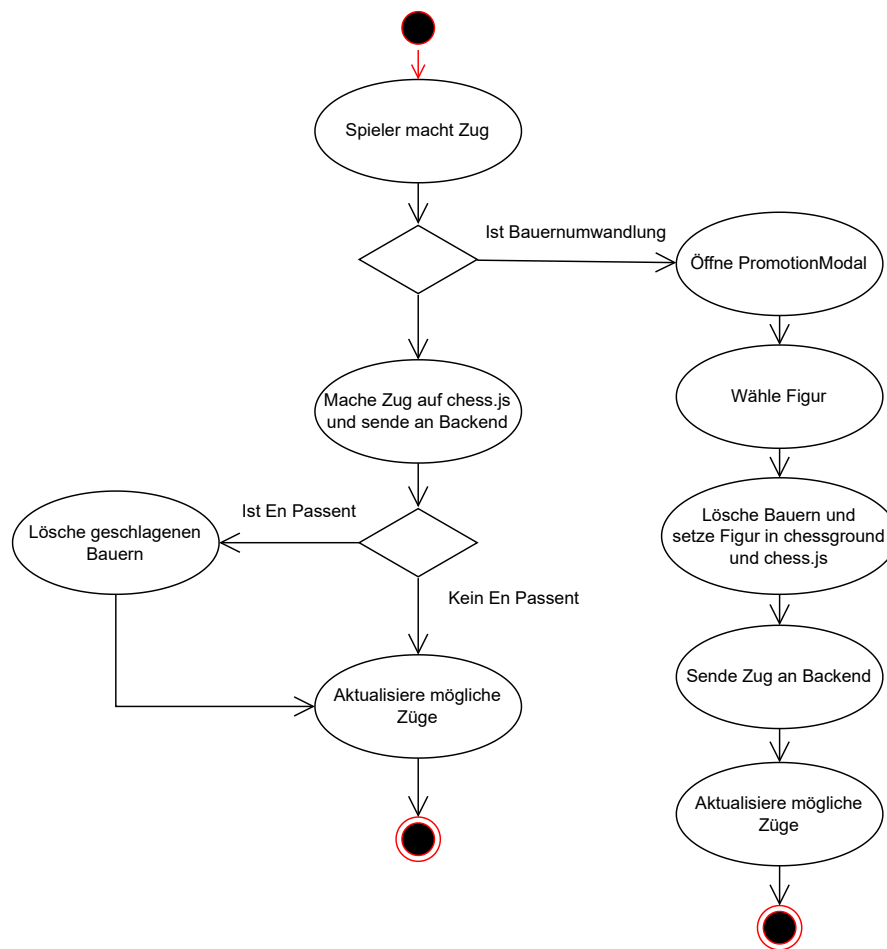


Abbildung 3.15: Aktivitätsdiagramm nach ziehen Schachzugs

## 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 eine/-n Zuschauer/-in 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 Versendenden, die Nachricht als String und die `roomId` des Spiels. Je nachdem ob der/die Versender/-in der Nachricht mit dem eigenen Namen übereinstimmt wird die Nachricht unterschiedlich ausgerichtet und eingefärbt (siehe Abbildung 3.9).

### 3.4.4 Verwaltung von befreundeten Personen

Die Verwaltung und Darstellung (siehe Abbildung 3.5) von befreundeten Personen und Freundschaftsanfragen obliegt der *FriendList*-Komponente. Diese beinhaltet die Unterkomponenten *Friend*, *FriendRequest* 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 befreundeten Personen je nachdem ob sie online sind oder nicht sortiert. *FriendList* hört dabei auf die folgenden Events:

- **friend\_request\_accepted:** Enthält Daten einer neuen befreundeten Person, welche deine Freundschaftsanfrage angenommen hat. Diese wird der Freundesliste hinzugefügt.
- **friend\_request:** Eingang einer neuen Freundschaftsanfrage. Wird der Liste der Freundschaftsanfragen hinzugefügt.
- **connected:** Dieses Event wird empfangen, falls ein/-e Freund/-in offline, beziehungsweise online geht. Der betreffende Freundes-Eintrag in der Freundesliste wird aktualisiert.

Zum Holen der Listen werden die zwei Events **get\_friends** und **get\_friend\_requests** jedes Mal gesendet, wenn auf die *Home*-Komponente navigiert wird. Diese beiden Events empfangen mittels Callback-Funktionen alle Daten über die befreundeten Personen und Freundschaftsanfragen. Dies ist nötig, damit, falls beispielsweise nach einer Schachpartie wieder auf die *Home*-Komponente navigiert wird, die Daten der befreundeten Personen aktualisiert werden.

#### ***Friend*-Komponente**

Diese Komponente stellt eine befreundete Person dar. Es kriegt als props den Benutzernamen, die aktiven Spiele und ihren online Status übergeben. Die Komponente hat zwei Grundlegende Funktionen: Das Zuschauen einer Partie einer befreundeten Person und das Herausfordern zu einer Partie. Beim Herausfordern sind genau die gleichen Schachuhren möglich, wie bei der Suche einer zufälligen gegnerischen Person und das Zuschauen ist aufgebaut wie bei der *ActiveGames*-Komponente. (siehe Abbildung 3.16) Diese beiden Funktionen sind nur verfügbar, falls der/die Freund/-in gerade online ist, welches über einen grünen Punkt ersichtlich ist und zum Zuschauen benötigt der/die Freund/-in trivialerweise mindestens ein aktives Spiel.

Die Komponente hat zwei Eventlistener: **game\_request\_accepted** und **game\_request\_denied**, welche einen Toast darstellen und den Spielenden 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 die herausgeforderte Person die Anfrage angenommen, beziehungsweise abgelehnt hat. Entscheidet sich der/die Spieler/-in davor doch nicht mehr gegen die befreundete Person zu spielen, kann er/sie auf den „Cancel“ Button klicken und die Spielanfrage wird zurückgenommen.



Abbildung 3.16: 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 3.5.3 erläutert. War das Akzeptieren, beziehungsweise das Ablehnen der Anfrage erfolgreich wird die Freundschaftsanfrage aus der Liste gelöscht und gegebenenfalls wird die neue befreundete Person der Freundesliste hinzugefügt. Diese wird mittels Callback Funktion empfangen.

### ***AddFriendModal-Komponente***

Diese Komponente besteht aus einem Button und einem Modal, das sich öffnet, 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 (siehe Abbildung 3.17). Wie das Backend eine neue Freundschaftsanfrage behandelt wird in Abschnitt 3.5.3 erläutert.

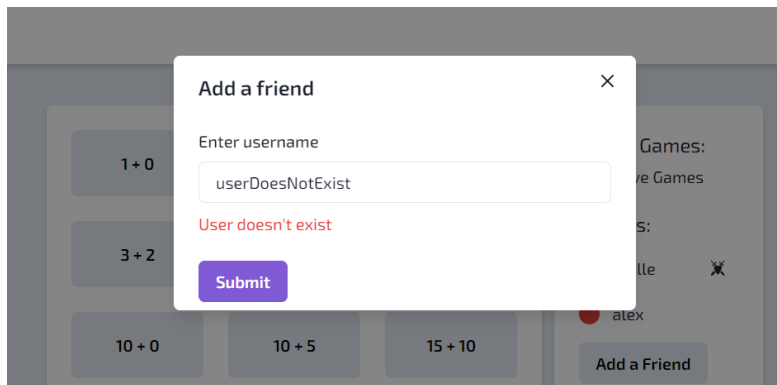


Abbildung 3.17: Das Modal der *AddFriendModal*-Komponente mit Fehlermeldung

### 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 sendet das Event `get_active_games`, welches mit einer Callback Funktion eine Liste aller aktiven Spiele empfängt. Dieses Event wird, wie bei dem Event `get_friends` von der *FriendList*-Komponente (siehe Kapitel 3.4.4), jedes mal gesendet, wenn auf die *Home*-Komponente navigiert wird, um die Liste der aktiven Spiele zu aktualisieren. Ist die Liste leer, wird nur „No active 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 3.18) 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 Benutzenden.

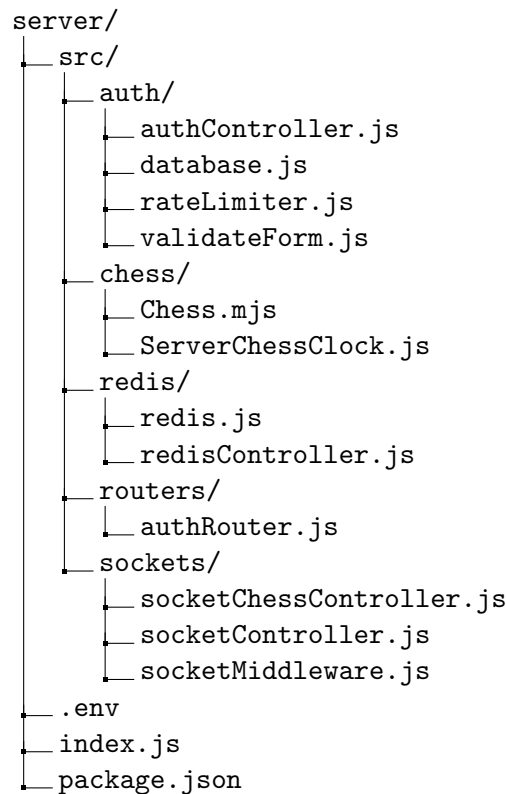


Abbildung 3.18: Ordnerstruktur des Backends

- **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.

### 3.5.1 Authentifizierung

Die Authentifizierung eines Benutzenden 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/e Benutzer/-in authentifiziert werden kann: Durch das Anmelden, durch das Registrieren oder durch das Lesen des Cookies.

Die Authentifizierung im Frontend 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 Routings verschiedene Middlewares verwendet.

Bei jeder Anfrage an die API stellt eine Middleware 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<sup>2</sup>, 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 diese/r Benutzer/-in in der Datenbank existiert und es wird mittels `bcrypt` kontrolliert, 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 Benutzenden geantwortet, dass die Anmeldung erfolgreich war mit der Übermittlung des Benutzernamens.

Beim Registrieren wird überprüft, ob bereits ein/-e Nutzer/-in mit dem Benutzernamen oder E-Mail existiert und anschließend wird ein neuer Tupel in der PostgreSQL Datenbank erstellt und dem Client geantwortet. 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 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/die Benutzer/-in automatisch angemeldet wird, ohne seine/ihre Anmeldeinformationen nochmals einzugeben.

Anschließend stellt der Client eine Socket.io Verbindung her.

### Socket.io Authentifizierung und Middleware

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

Die erste Middleware Authentifiziert die Socket des Benutzenden. Sie liest aus dem Cookie, der auch bei der Verbindungsherstellung 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.

---

<sup>2</sup>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](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

Wenn der/die Benutzer/-in keinen gültigen JWT-Token besitzt, 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.

In der zweiten Middleware tritt die Socket der `userid` des Benutzenden als Raum bei und wird sowohl in der Redis Datenbank unter `user:username` (siehe Kapitel 3.6.2), als auch bei befreundeten Personen mittels des Events `connected`, als online vermerkt. Der Beitritt der eigenen `userid` dient als Kommunikationsschnittstelle. So kann die `userid` aus Redis geholt werden und Sockets können sie verwenden, um an diesen Benutzenden zu senden.

Beim Schließen der Anwendung oder Abmelden des Benutzenden wird dementsprechend in Redis `connected` auf `false` gesetzt und die Freunde werden mit dem Event `connected` darüber in Kenntnis gesetzt, dass der/die Benutzer/-in nicht mehr online ist.

Als letzte Middleware werden alle nötigen Listener sowohl für das Schachspiel, als auch für sonstige Funktionen initialisiert.

### 3.5.2 Das Schachspiel

In diesem Abschnitt werden die Prozesse beim Suchen eines Gegners, der Initialisierung des Schachspiels, der Ausführung neuer Züge sowie dem Senden und Empfangen von Nachrichten im Chat detailliert beschrieben.

#### Finden eines Gegners

Es gibt 3 verschiedene Möglichkeiten ein Schachspiel zu starten: Unangemeldete Personen suchen nach zufälligen Mitspielenden, angemeldete Personen suchen nach zufälligen Mitspielenden oder angemeldete Personen fordern eine befreundete Person zu einer Partie heraus.

Dabei ist zu beachten, dass angemeldete Benutzende auch nur gegen andere angemeldete Benutzende spielen können und unangemeldete Spielende auch nur gegen unangemeldete Spielende. Dies hat Erweiterbarkeitsgründe (siehe Kapitel 5.3).

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 Initialisieren 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 die Person nicht angemeldet ist, wird ihr ein zufälliger Benutzername für dieses Schachspiel zugewiesen, der mit „guest-“ startet. Ihre `userid` wird als die Socket ID festgelegt.
- Daraufhin wird im Server ein/-e Spieler/-in aus der entsprechenden Warteschlange (siehe Abschnitt 3.6.2) in Redis entnommen.

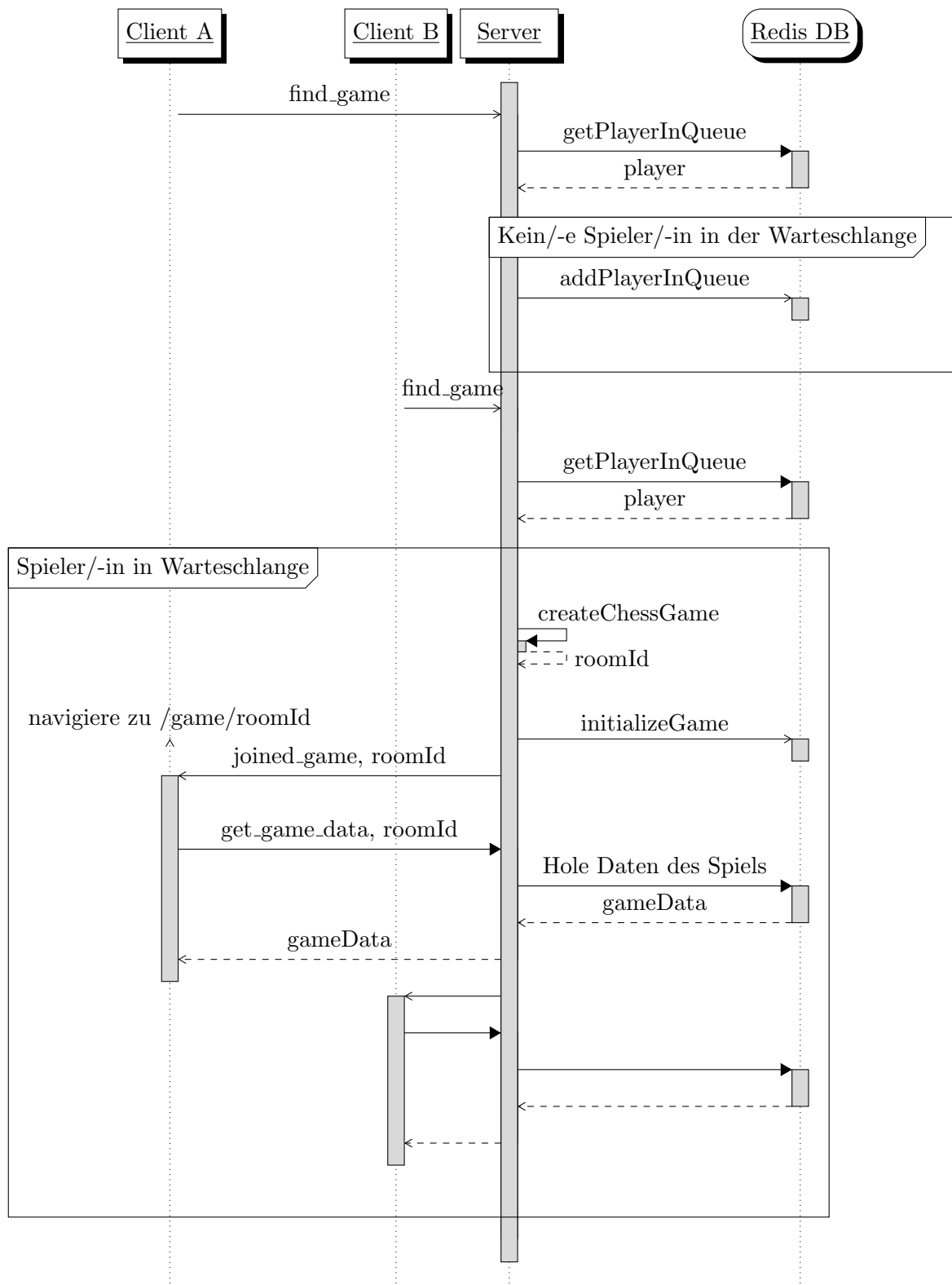


Abbildung 3.19: Sequenzdiagramm des Schachspielstartprozesses mit unbekanntem Gegner



- Falls dabei niemand entnommen werden konnte, wird der/die Benutzer/-in selbst in die Liste geschrieben und wartet bis er/sie von einem anderen Benutzenden aus der Liste genommen wird.
- Falls jemand aus der Liste entnommen werden konnte, wird das Spiel mit einer `roomId` als Identifikator initialisiert und in Redis gespeichert. Dabei wird auch ein `ServerChessClock` Objekt kreiert und in einem Array der `socketChessListeners` Datei gespeichert.
- An die beiden Spielenden 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.
- Die `ChessGame`-Komponente sendet das `get_game_data` Event. Daraufhin wird der aktuelle Zustand der Partie aus Redis und der `ServerChessClock` 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 einer befreundeten Person resultiert, wird natürlich in keine Warteschlange nach einer gegnerischen Person gesucht, anstatt dessen wird mit den Events `send_game_request`, `game_request` und `game_request_response` (siehe Abschnitte 3.4.3) die Anfrage versendet und beantwortet. Anschließend wird das Spiel initialisiert und an die beiden Spielenden das Event `game_request_accepted` mit der `roomId` gesendet (für Frontend-Informationen zu diesen Events siehe Kapitel 3.4.3 & 3.4.4).

Die Implementierungsdetails, wie ein Schachspiel auf dem Server mittels der Methoden `createChessGame` und `initializeGame` initialisiert und die Informationen des Schachspiels an das Frontend gesendet werden, sind in Kapitel 4.2.2 ausführlich beschrieben.

### ServerChessClock

Die Klasse `ServerChessClock` definiert einzelne Schachuhr Objekte, welche die Startzeiten und die regulären Zeiten einer Schachpartie 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. Die Funktionsweise und Codeausschnitte der serverseitigen Schachuhr werden in Kapitel 4.2.2 erläutert.

### Neue Züge

Ein Aktivitätsdiagramm zur Behandlung eines neuen Zugs im Backend ist in Abbildung 3.20 zu finden.

Sobald ein neuer Zug eines Spielenden ankommt wird zunächst die bisherige PGN-Notation des Spiels aus Redis und das `ServerChessClock` Objekt aus dem Array geholt.

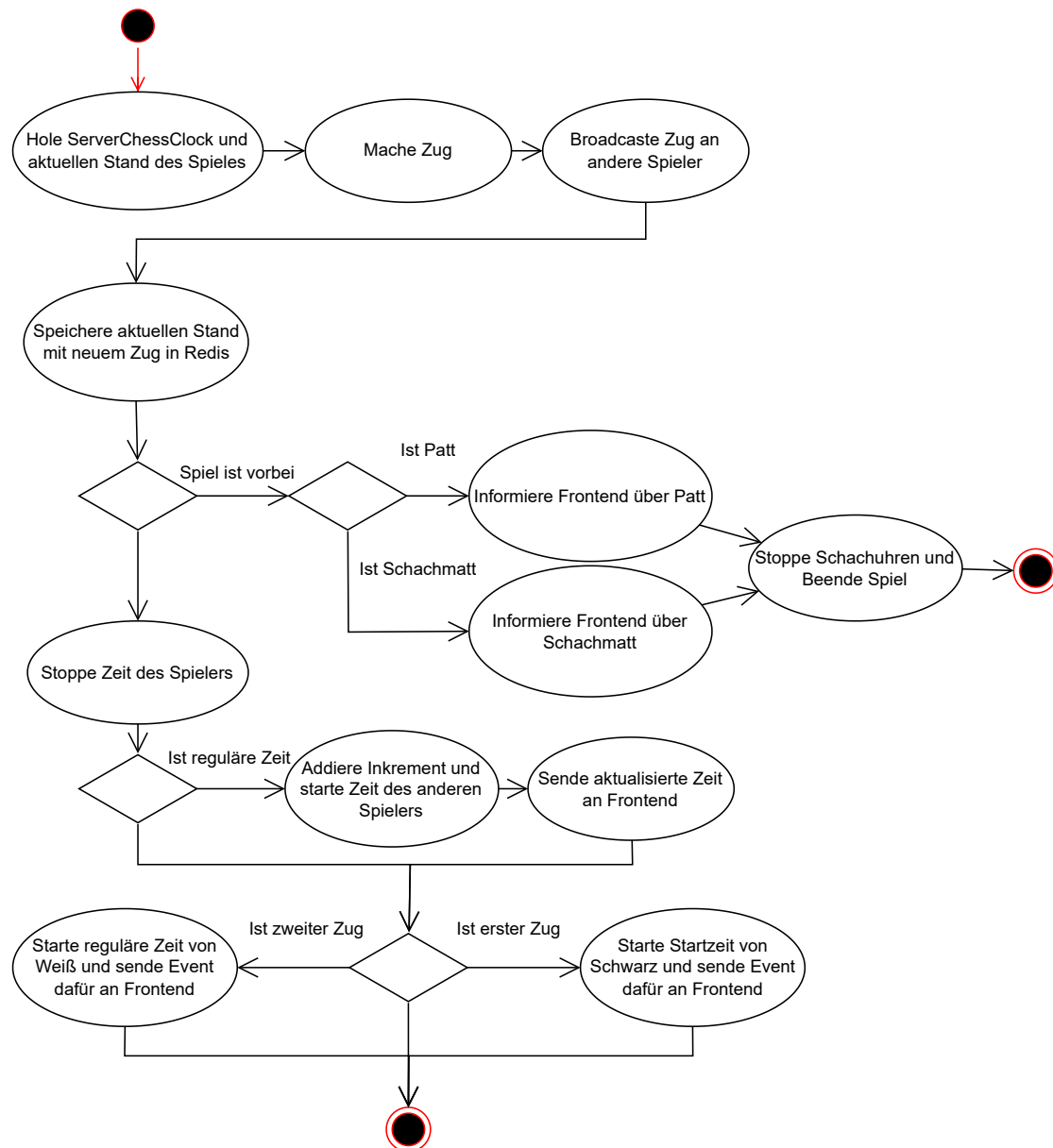


Abbildung 3.20: Aktivitätsdiagramm zur Behandlung eines neuen Zugs im Backend

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 der sendenden Person, da dieser ja bereits den Zug gemacht hat. Die veraltete PGN-Notation des Spiels wird nun mit der neuen Notation in Redis überschrieben.

Anschließend 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.

Falls dies nicht der Fall ist wird die Zeit des jetzigen Spielenden gestoppt und falls es eine reguläre Zeit ist, wird das Inkrement auf die Zeit gerechnet, die Zeit des anderen Spielenden 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 Spielenden 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.

### Ende einer Schachpartie

Das Ende einer Schachpartie kann durch folgende Situationen stattfinden: Schachmatt, Patt, Startzeit ist abgelaufen, eine reguläre Zeit ist abgelaufen oder ein/-e Spieler/-in 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 Benutzende handelt wird das entsprechende Spiel aus den **activeGames**-Listen von **user:username** gelöscht. Anschließend wird der Eintrag **game:roomId** gelöscht. Somit wird kein Speicher mehr von alten Spielen belegt.

### Chat

Der Listener auf das Event **send\_message** empfängt und verwaltet die neue Nachricht, die ein/-e Spieler/-in 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 alten 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.

### 3.5.3 Verwaltung von Freunden

In den Aktivitätsdiagrammen in Abbildung 3.21 ist der Ablauf des Versendens und des Akzeptierens einer Freundschaftsanfrage abgebildet.

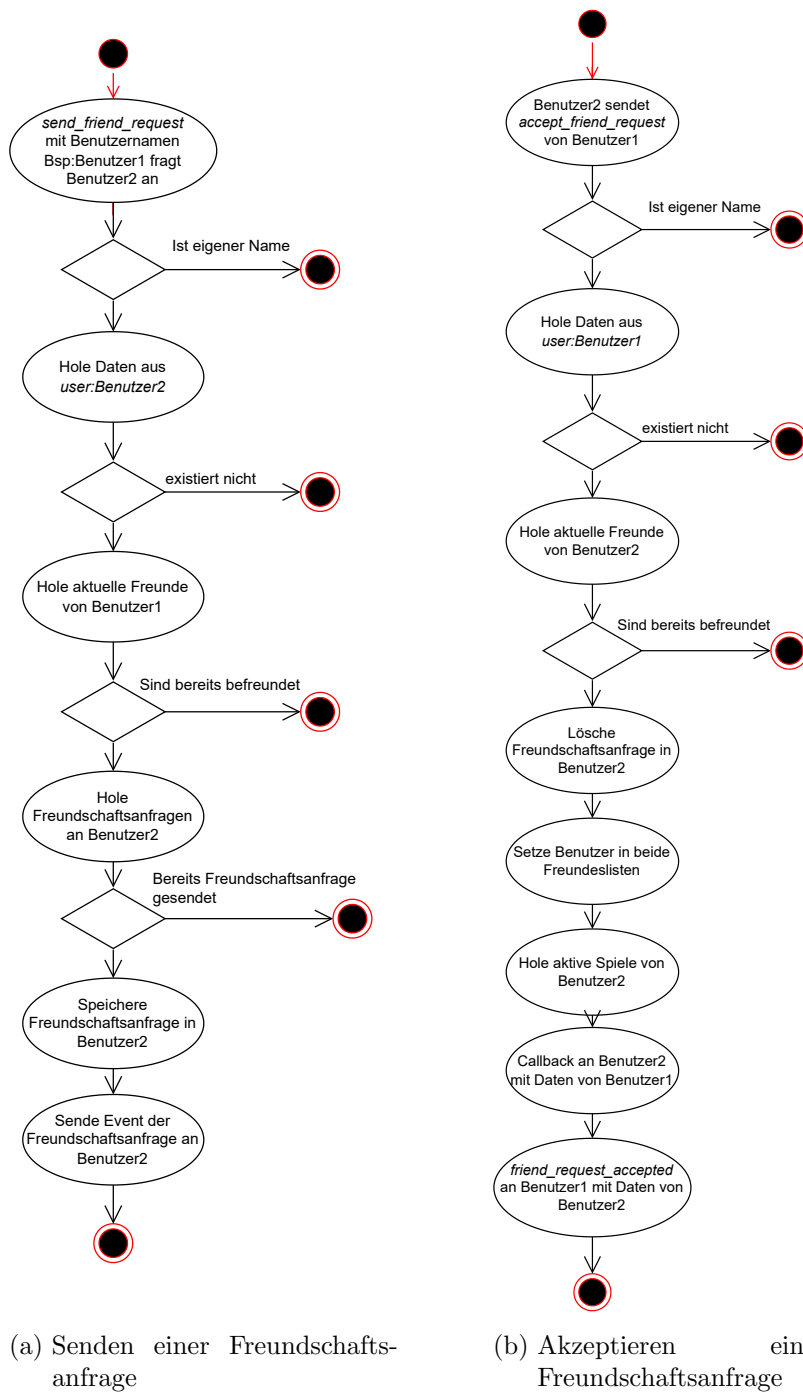


Abbildung 3.21: Aktivitätsdiagramme des Versenden und Akzeptieren von Freundschaftsanfragen

Die verschiedenen Datentypen der Speicherung in Redis befindet sich in Abschnitt 3.6.2. Beispiel Codeausschnitte und Implementierungsdetails über die Verwaltung von Freunden im Backend befindet sich in Kapitel 4.2.3.

### 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/die Nutzer/-in nicht existiert und ob die beiden Benutzenden schon befreundet sind. Ist eines davon der Fall, wird mit einer entsprechenden Fehlermeldung dem Frontend 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 ist, wird der/die Benutzer/-in ebenfalls darauf hingewiesen. Ansonsten wird die Freundschaftsanfrage in Redis gespeichert und ein Event mit der Freundschaftsanfrage wird an den betreffenden Spielenden gesendet und die sendende Person wird darüber informiert, dass die Freundschaftsanfrage erfolgreich 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 Benutzenden werden in die entsprechende Freundesliste gesetzt, die restlichen Daten der beiden Spielenden werden eingeholt und an den jeweils anderen Benutzenden werden alle Informationen, wie ob er/sie online ist oder ob er/sie aktive Spiele hat, gesendet. Dies stellt sicher, dass die befreundeten Personen im Frontend direkt richtig angezeigt werden können.

Beim Ablehnen einer Freundschaftsanfrage wird diese einfach nur aus Redis gelöscht.

## 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 Benutzenden.
- **userid:** Jede spielende Person erhält bei der Registrierung seine eigene userid. Diese dient der Kommunikation mit dem Benutzenden (siehe Kapitel 3.5.1).

- **password:** Das mit bcrypt verschlüsselte Passwort des Benutzenden.

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 beispielsweise `user:Max` oder `Max:18b06...`

#### `user:username`

Unter dem Key `user:username` befindet sich ein Redis Hash. Ein Redis Hash besitzt Key-Value Paare, auf welche man zugreifen kann<sup>3</sup>.

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/die Benutzer/in gerade online ist oder nicht.
- **activeGames:** Eine Liste in der alle `roomId`s der aktuellen Spiele des Benutzenden gespeichert sind

#### `game:roomId`

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

- **whitePlayer, blackPlayer:** Benutzernamen des weißen und schwarzen Spielenden.
- **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.

#### `friends:username`

Der Key `friends:username` verweist auf eine Liste von befreundeten Personen. Jeder Eintrag besteht aus einem String, der sich zusammensetzt aus `username:userid`. Dies verhindert, dass wenn ein Event an eine/-n Freund/-in gesendet werden soll, man einen extra Zugriff auf `user:username` machen muss um die `userid` zu bekommen.

<sup>3</sup>Quelle: <https://redis.io/docs/data-types/strings/> am 15 Mai 2023

**friend\_requests:username**

Dies ist eine Liste die genauso aufgebaut ist wie die **friends:username** Liste, außer, dass sie offene Freundschaftsanfragen dokumentiert.

**Warteschlangen**

Die Warteschlangen für Partien mit einer zufälligen gegnerischen Person bestehen aus **waitingPlayers:timeMode** und **waitingGuests:timeMode**, je nachdem, ob es sich um einen angemeldeten Benutzenden handelt oder nicht. **timeMode** repräsentiert hier alle möglichen Schachuhr-Konfigurationen, also beispielsweise „10 + 5“, „15 + 10“, ...

Diese Warteschlangen sind Listen, welche aus **username:userid** Einträgen bestehen, um zu verhindern, dass ein weiterer Zugriff auf **user:username** nötig ist, um die **userid** zu bekommen.

Durch die Redis Operationen **RPOP** und **LPUSH** lassen sich atomar Einträge hinzufügen oder herausnehmen, welches die Konsistenz der Liste gewährleistet<sup>4</sup>.

### 3.7 Testen der Anwendung

Während der Entwicklung der Schach-App, wurden die Funktionen ausgiebig getestet, um die Richtigkeit der Implementierung zu gewährleisten. Aufgrund der begrenzten Zeit wurde der Schwerpunkt auf das Testen des Zusammenspiels zwischen Frontend und Backend gelegt, statt auf ausgiebige Tests der einzelnen Codeausschnitte des Front- und Backends.

Um die Funktionsweise der verschiedenen Komponenten zu überwachen wurden viele Konsolenausgaben genutzt, um zu überprüfen, dass die Anwendung das gewünschte Verhalten aufzeigt und um Fehlerquellen zu identifizieren.

Es ist zu beachten, dass dieser Ansatz zwar eine schnelle und effektive Methode zum Testen der Hauptfunktionalitäten darstellt, jedoch möglicherweise nicht alle möglichen Fehlerfälle oder Randbedingungen abdeckt. Für die zukünftige Entwicklung wäre es sinnvoll, den Testprozess zu strukturieren und ausgiebige Tests für das Front- und Backend durchzuführen, um die Stabilität der Anwendung zu verbessern.

---

<sup>4</sup>Quelle: <https://redis.io/docs/data-types/lists/> am 15. Mai 2023





## 4 Implementierung

Grobe Funktionsweisen der Implementierung wurden bereits im Kapitel Systemarchitektur erläutert. In diesem Kapitel wird auf detaillierte Implementierungsaspekte anspruchsvoller Prozesse mit Hilfe von Codeausschnitten eingegangen. Funktionen, die weniger komplex sind und deren Behandlung im Kapitel Systemarchitektur als ausreichend angesehen wird, werden nicht erneut thematisiert.

### 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 Codeausschnitte vorstellen.

#### Erster Versuch der Authentifizierung mittels Cookie

Im Codeausschnitt 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   useEffect(() => {
9     fetch(`${process.env.REACT_APP_API_URL}/auth/login`, {
10       credentials: "include",
11     })
12       .catch(err => {
13         console.log(err);
14         return;
15       })
16       .then(r => {
17         if (!r || !r.ok || r.status >= 400) {
18           return;
19         }
20         return r.json();
21       })
22       .then(data => {
```

```

23         setUser({ ...data});
24     });
25 }, []);
26 return (
27     <AccountContext.Provider value={{user, setUser}}>
28         {children}
29     </AccountContext.Provider>
30 );
31 }
32 export default useContext;

```

Codeausschnitt 4.1: Die AccountContext.js-Datei

In ihr wird der State `user` mit `{loggedIn: null}` initialisiert. Weshalb wir dies tun wird durch den Codeausschnitt 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 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<sup>1</sup>. 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 Codeausschnitt 4.2.

```

1     const submitLogin = useCallback((values, setSubmitting) => {
2         fetch(`${process.env.REACT_APP_API_URL}/auth/login`, {
3             method: "POST",
4             credentials: "include",
5             headers: {
6                 "Content-Type": "application/json",
7             },

```

<sup>1</sup>Quelle: <https://javascript.info/fetch-crossorigin> am 11. Mai 2023

```

8      body: JSON.stringify(values)
9    })
10    .catch(err => {
11      setLoginError("Please try again later");
12      setSubmitting(false);
13      return;
14    })
15    .then(res => {
16      if (!res || !res.ok || res.status >= 400) {
17        setSubmitting(false);
18        setLoginError("Please try again later");
19        return;
20      }
21      return res.json();
22    })
23    .then(data => {
24      if (!data.loggedIn) {
25        setLoginError(data.message);
26        setSubmitting(false);
27        return;
28      }
29      setUser({...data});
30      setLoginError(null);
31      navigate('/');
32    });
33  }, [setLoginError, setUser, navigate]);

```

Codeausschnitt 4.2: Die Funktion zum Senden der Benutzerdaten an das Backend

Die Funktion zum Senden der Registrierungs-Daten ist ähnlich aufgebaut, 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 (siehe Kapitel 2.2.3). Der „Content-Type“ hilft dem Server zu erkennen um was für eine Art von Daten es sich handelt<sup>2</sup>, während im Body die angegebenen Anmeldedaten als String verpackt werden. Falls der Server mit einem Fehler antwortet, wird dieser durch den State `loginError` erfasst und dargestellt. Ansonsten wird der Benutzerstatus gesetzt und es wird auf den Pfad `/` navigiert, auf der sich die *Home*-Komponente befindet.

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}

```

<sup>2</sup>Quelle: <https://javascript.info/fetch> am 11. Mai 2023

```

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 isValid={form.errors.password &&
19              form.touched.password}>
20              <FormLabel htmlFor="password">Password</FormLabel>
21              <InputGroup>
22                <Input {...field}
23                  id="password"
24                  type={showPassword ? 'text' : 'password'}
25                  autoComplete="current-password" />
26                <InputRightElement>
27                  <IconButton
28                    icon={showPassword ? <ViewOffIcon />
29                      : <ViewIcon />}
30                    onClick={handlePasswordClick}
31                    variant="ghost"
32                    _hover={{ bg: hover }}
33                    aria-label="Toggle password
34                      visibility"
35                  />
36                </InputRightElement>
37              </InputGroup>
38              <FormErrorMessage>{form.errors.password}</FormErrorMessage>
39            </FormControl>
40          )}
41        </Field>
42        ...
43        <Button
44          type="submit"
45          disabled={!isValid || isSubmitting}
46          ...
47        >
48          Log In
49        </Button>
50      </Form>
51    )}
52  </Form>
53  ...

```

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

In diesem Codeausschnitt 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 Codeausschnitt 4.4) und es wird definiert, dass bei jeder neuen Änderung der Eingabefelder des Formulars, die Eingaben auf das Schema getestet werden sollen (siehe Zeilen 7 & 8 des Codeausschnitts 4.3).

`isValid` und `isSubmitting` (siehe Zeile 13 des Codeausschnitts 4.3) 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 (siehe Zeile 9 des Codeausschnitts 4.3). 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 (siehe Zeile 42 & 43 des Codeausschnitts 4.3).

Als Eingabefelder, gibt es den Benutzernamen und das Passwort. Aus Demonstrationszwecken habe ich in den Codeausschnitt 4.3 nur das Passwort eingefügt.

Die Props `field` und `form` (siehe Zeile 17 des Codeausschnitts 4.3) 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<sup>3</sup>.

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 (siehe Kapitel 4.1.4).

Die `Login`-Komponente besitzt den boolean State `showPassword`, welcher angibt, ob das Passwort zu sehen sein soll oder nicht (siehe Zeile 23 des Code Ausschnitts 4.3). Bei einem Klick auf das Icon mit dem Auge wird die Funktion `handlePasswordClick` ausgeführt, welche den `showPassword` State negiert. Je nach dem aktuellen Wert des States `showPassword` wird ein Auge oder ein durchgestrichenes Auge als Icon angezeigt. 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       => {
7         return !value.startsWith('guest');
8       })
9     .test('no_colon', 'Username cannot include ":"', (value) => {
10       return !value.includes(':');
11     })
12     .required('Username is a required field'),
13   password: Yup.string()
14     .min(8, 'Password has to be at least 8 characters long')
15     .minLowercase(1, 'At least one character hat to be in lower
16       case')
17     .minUppercase(1, 'At least one character has to be in upper
18       case')
19     .minNumbers(1, 'At least on character has to be a number')

```

<sup>3</sup>Quelle: <https://formik.org/docs/api/field> am 11. Mai 2023

```

17     .minSymbols(1, 'At least one character has to be a symbol')
18     .minRepeating(3, 'It is only allowed to have at most 3 repeating
19       characters')
20     .required('Password is a required field'),
  });

```

Codeausschnitt 4.4: Yup Schema für das Anmelden

In diesem Beispiel ist das `LoginSchema` zu sehen, welches für die Validierung der Eingabefelder im Anmeldeformular aus Codeausschnitt 4.3 verwendet wird. Yup ermöglicht es bestimmte Restriktionen an Eingabefelder zu stellen und dabei direkt die Fehlernachrichten zu definieren, falls eine Restriktion nicht erfüllt ist. Selbstverständlich sind die Restriktionen für das Registrieren die gleichen, wobei das Registrierungsformular noch ein paar weitere Felder hat.

Verwendet werden übliche Restriktionen an Benutzernamen und Passwörter, sodass das Passwort mindestens eine Nummer, ein Symbol, einen Großbuchstaben, ... beinhalten muss. Eine Besonderheit ist, dass der Benutzername nicht mit „guest“ beginnen darf (siehe Zeile 5 f. des Codeausschnitts 4.4), da unangemeldete Benutzer bei Schachpartien einen solchen Benutzernamen zugewiesen bekommen (siehe Kapitel 4.2.2). Eine zweite Besonderheit ist, dass der Benutzername keinen Doppelpunkt beinhalten darf. Dies liegt daran, dass wir in der Redis Datenbank öfters die `userid` mit dem Benutzernamen mit einem Doppelpunkt in einem String verbinden (siehe Kapitel 3.6.2).

### 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     useEffect(() => {
12         if(user.loggedIn !== null) {
13             if(socket) {
14                 socket.disconnect();
15             }
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;

```

Codeausschnitt 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 in der *UserContext*-Komponente ohnehin eine neue Verbindung aufgebaut werden würde. Die Verbindung der Socket zu trennen vor der neuen Verbindung hat den Sinn, im Backend ein `disconnect` Event zu triggern (siehe Kapitel 4.2.1).

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 Codeausschnitt 4.6).

```

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

```

Codeausschnitt 4.6: Ausschnitt der *Views*-Komponente

### 4.1.2 Das Schachspiel

In diesem Abschnitt werde ich darauf eingehen, was passiert nachdem die Daten erfolgreich vom Backend empfangen wurden, wie ein neuer Zug gehandhabt wird und wie die Bauernumwandlung implementiert ist. Der Ablauf wie ein Gegner gefunden wird, wie auf die *ChessGame*-Komponente navigiert wird und welche Daten von dem Backend empfangen werden befindet sich in Kapitel 3.4.3.

## Initialisierung nach dem Empfangen der Daten

```

1  useEffect(() => {
2      if(initialized) {
3          setGround(new Chessground(document.getElementById(roomId), {
4              fen: chess.fen(),
5              orientation: orientation,
6              viewOnly: isSpectator,
7              movable: {
8                  free: false,
9                  color: orientation,
10                 showdests: true
11             },
12             premovable: {
13                 enabled: false
14             },
15             animation: {
16                 enabled: true,
17                 duration: 400
18             }
19         }));
20     }
21 }, [initialized]);
22
23 useEffect(() => {
24     if(ground) {
25         ground.set({
26             movable: {
27                 events: {
28                     after: onMove(ground, chess)
29                 }
30             }
31         });
32         refreshBoard(ground, chess);
33         socket.on('opponent_move', (move) => {
34             if(move.flags.includes('p')) {
35                 onOpponentPromotion(move);
36                 return;
37             }
38             ground.move(move.from, move.to);
39             chess.move(move);
40             if(move.flags.includes('e')) {
41                 onEnPassent(ground, move);
42             }
43             refreshBoard(ground, chess);
44         });
45         socket.on('cancel_game', () => {...});
46         socket.on('time_over', (color) => {...});
47         socket.on('checkmate', (winner) => {...});
48         socket.on('draw', () => {...});
49         socket.on('resigned', (color) => {...});
50     }
51     return () => {

```



```

52     socket.off('opponent_move');
53     socket.off('Checkmate');
54     socket.off('draw');
55     socket.off('time_over');
56     socket.off('cancel_game');
57     socket.off('resigned');
58   }
59 }, [socket, ground, initialized, ground, chess, orientation,
    whitePlayer, blackPlayer]);

```

Codeausschnitt 4.7: Initialisierung des Schachspiels nach Empfangen der Daten

Sobald erfolgreich die Daten mit dem `get_game_data` Event empfangen und gesetzt wurden, wird `initialized` auf `true` gesetzt, um die nächsten Schritte in Gang zu leiten. Zunächst wird ein `chessground` Objekt mit Konfigurationen erstellt. In diesem wird übermittelt, wie die Figuren gerade stehen (mittels FEN-Notation), die Orientierung welche Farbe nach unten zeigen soll, ob man Figuren bewegen darf und wenn ja welche und weitere Optionen. Dieses `chessground` Interface wird dann in dem definierten `div` Bock angezeigt (siehe Zeile 3 des Codeausschnitts 4.7).

Wurde dann `chessground` initialisiert, wird noch die Methode `onMove` zu `chessground` hinzugefügt, die ausgeführt werden soll, wenn man selbst einen Zug macht (siehe Zeile 28 Codeausschnitt 4.7) und Listener für verschiedene Events des Schachspiels werden initialisiert. Diese Listener werden wieder entfernt (siehe Zeile 52 ff. in Codeausschnitt 4.7), falls die `ChessGame`-Komponente verlassen wird, da sie nur für dieses Spiel auf die Events hören sollen und bei einem neuen Spiel neue Listener initialisiert werden.

Nach jedem Zug, egal ob eigener oder eines Gegners, wird die `refreshBoard` Methode aufgerufen.

```

1 export function refreshBoard(ground, chess) {
2   ground.set({
3     turnColor: toColor(chess),
4     movable: {
5       dests: getValidMoves(chess)
6     }
7   });
8 }

```

Codeausschnitt 4.8: Die refreshBoard Methode

Diese Methode sorgt dafür, dass eine Figur von `chessground` nur auf legale Züge ziehen darf und korrekt aktualisiert wird, welcher Spieler an der Reihe ist.

Ein `move` Objekt, das zurückgegeben wird nach dem ziehen eines Zuges auf einem `chess.js` Objekt (siehe `move` Parameter Zeile 35 in Codeausschnitt 4.7 oder Zeile 10 Codeausschnitt 4.9)) ist folgendermaßen aufgebaut<sup>4</sup>:

```
{ color: 'w', from: 'e2', to: 'e4', flags: 'b', piece: 'p', san: 'e4' }
```

Vor allem relevant sind die `flags` dieses Objekts, da sie ausdrücken um was für eine Art von Zug es sich handelt. Wird als `flag` beispielsweise „e“ angegeben, handelt es sich um

<sup>4</sup>Quelle: <https://github.com/jhlywa/chess.js/> am 08. Mai 2023

ein En passent. Bei einem „p“ liegt eine Bauernumwandlung vor. Je nachdem ob eines dieser flags vorhanden ist, wird der Zug anders verarbeitet. Wie ein gegnerischer Zug behandelt wird (siehe Zeile 33 ff. in Codeausschnitt 4.7), ist in Kapitel 3.4.3 erläutert.

### Neuer Zug

```

1  const onMove = useCallback(() => {
2      return (orig, dest) => {
3          //Promotion
4          if((dest.includes('8') || dest.includes('1')) &&
5              chess.get(orig).type === 'p') {
6              setSelectVisible(true);
7              setPromotionMove([orig, dest]);
8              return;
9          }
10         const player = (orientation === 'white') ? whitePlayer :
11             blackPlayer;
12         const move = chess.move({from: orig, to: dest});
13         socket.emit('new_move', roomId, player, move, ({done, errMsg}) =>
14             {
15                 if(!done) {
16                     chess.undo();
17                     ground.set({fen: chess.fen()});
18                     toast({
19                         title: "Invalid Move",
20                         description: errMsg,
21                         status: "error",
22                         position: "top",
23                         isClosable: true
24                     });
25                     refreshBoard(ground, chess);
26                     return;
27                 }
28             });
29         if(move.flags.includes('e')) {
30             onEnPassent(ground, move);
31         }
32         refreshBoard(ground, chess);
33     };
34 }, [socket, chess, roomId, ground, orientation, setSelectVisible,
35     setPromotionMove, whitePlayer, blackPlayer]);

```

Codeausschnitt 4.9: Die onMove Methode

Die `onMove` Methode, welche ausgeführt wird, sobald auf dem chessground Interface eine Figur von dem Spieler bewegt wurde, behandelt, wie schon in Abschnitt 3.4.3 erläutert, en passant- und Bauernumwandlungszüge gesondert. Wenn es sich um eine Bauernumwandlung handelt (ein Zug in Reihe 1 oder 8 mit einem Bauern), wird `selectVisible` auf `true` gesetzt, welches bewirkt, dass die Komponente *PromotionModal* sein Modal öffnet und man eine Figur auswählen kann, in welche der Bauer transformiert werden soll. Um die Felder dieses Zuges zu speichern und außerhalb dieser Methode Verfügbar

zu machen, werden sie in den State `promotionMove` gespeichert (siehe Zeile 6 des Codeausschnitts 4.9).

```

1  const promotion = useCallback(
2    (toPiece) => {
3      const move = chess.move({from: promotionMove[0], to:
4        promotionMove[1], promotion: toPiece});
5      ground.state.pieces.set(promotionMove[1], {
6        role: charPieceToString(toPiece),
7        color: parseInt(promotionMove[1].split('')[1]) === 8 ?
8          'white' : 'black',
9        promoted: true
10     });
11     const player = (orientation === 'white') ? whitePlayer :
12       blackPlayer;
13     socket.emit('new_move', roomId, player, move, ({done, errMsg})
14       => {
15       if(!done) {
16         chess.undo();
17         ground.set({fen: chess.fen()});
18         toast({
19           title: "Invalid Move",
20           description: errMsg,
21           status: "error",
22           position: "top",
23           isClosable: true
24         });
25         refreshBoard(ground, chess);
26         return;
27       }
28     });
29     setPromotionMove([]);
30     setSelectVisible(false);
31     refreshBoard(ground, chess);
32   },
33   [socket, roomId, chess, ground, promotionMove]
34 );

```

Codeausschnitt 4.10: Die promotion Methode

Wurde auf eine Figur des Modals geklickt, wird von dort die Methode `promotion` mit der entsprechenden Figur ausgeführt. Dann wird sowohl auf der `chess.js`, als auch auf der `chessGround` Instanz diese Bauernumwandlung durchgeführt, an das Backend gesendet und die States `promotionMove` und `selectVisible` werden zurückgesetzt. Da `chess.js` und `chessground` verschiedene Notationen der Figuren verwenden transformiert die Funktion `charPieceToString` die Notation der Figuren beispielsweise von „q“ in „queen“. Die Farbe der Figur wird durch die Reihe ermittelt in der die Bauernumwandlung stattfindet (siehe Zeile 6 des Codeausschnitts 4.10).

## Schachuhr

Die Komponente *ChessClock* übernimmt die Verwaltung und die Darstellung der Schachuhren. Die Listener dieser Komponente wurden in Kapitel 3.4.3 erläutert.

```

1  ...
2  const currentTimer = useRef(0);
3  ...
4  socket.on('updated_time', (timeWhite, timeBlack, turn) => {
5      stopClocks();
6      updateTime(timeWhite, timeBlack);
7      setCurrentTurn(turn);
8  });
9  ...
10 useEffect(() => {
11     if(currentTurn === 'off') {
12         return;
13     }
14     const functionMap = {
15         tw: setTimeWhite,
16         tb: setTimeBlack,
17         sw: setStartingTimeWhite,
18         sb: setStartingTimeBlack,
19     };
20     const setFunction = functionMap[currentTurn];
21     const id = setInterval(() => {
22         decrease(setFunction);
23     }, 1000);
24     currentTimer.current = id;
25     return () => {
26         clearInterval(id);
27     }
28 }, [currentTurn]);

```

Codeausschnitt 4.11: Ausschnitt der *ChessClock*-Komponente

Mit diesem Beispiel kann man gut demonstrieren, wie vielfältig die `useEffect`-Hook genutzt werden kann. Man nutzt in diesem Fall keine selbst definierte Funktion um eine Zeit zu starten, sondern nutzt die Eigenschaft der Hook, dass sie ausgeführt wird sobald sich ein Wert des Dependency Arrays (in diesem Fall `currentTurn`) ändert, um eine Zeit starten zu lassen. Den State der Zeit die ablaufen soll wird mittels `currentTurn` definiert. Diese kann die folgenden Werte annehmen:

- „sw“: Die Startzeit von Weiß läuft.
- „sb“: Die Startzeit von Schwarz läuft
- „tw“: Die reguläre Zeit von Weiß läuft.
- „tb“: Die reguläre Zeit von Schwarz läuft.
- „off“: Keine Zeit läuft.

Diese Werte werden auch in der serverseitigen Schachuhr verwendet (siehe Kapitel 4.2.2). Die `setInterval`-Methode (siehe Zeile 21 ff. des Codeausschnittes 4.11) akzeptiert als ersten Parameter eine Funktion und als zweiten Parameter eine Zahl, die das Zeitintervall in Millisekunden angibt, in dem die Funktion ausgeführt werden soll. In unserem Fall soll die verbleibende Zeit jede Sekunde um eine Sekunde reduziert werden. Die `setInterval`-Methode gibt eine ID zurück, die in der `useRef`-Variable `currentTimer` gespeichert wird. Im Vergleich zur `useState`-Hook hat die `useRef`-Hook den Vorteil, dass sie keine Neurenderung der Komponente auslöst, wenn sich der Wert ändert, und besser geeignet ist, um Werte über mehrere Rendervorgänge hinweg persistent zu speichern<sup>5</sup>. Über `currentTimer` kann in anderen Funktionen, beispielsweise beim Anhalten einer Uhr, auf die ID zugegriffen werden, und die `setInterval`-Funktion kann mit `clearInterval(currentTimer.current)` gestoppt werden.

### 4.1.3 Verwaltung von Freunden

Die Verwaltung von Freunden und Freundschaftsanfragen im Frontend wurde in Kapitel 3.4.4 behandelt. In diesem Kapitel möchte ich konkrete Code Beispiele der Komponenten erläutern.

#### Darstellung der Freundesliste

```

1 {friends.length === 0 ? (
2   <Text>No Friends</Text>
3 ) : (
4   <>
5     {friends
6       .slice()
7       .sort((a, b) => (a.connected === "true") - (b.connected ===
8         "true"))
9       .map((friend) => (
10        <Friend key={friend.username} friend={friend}
11          times={props.times} />
12      ))}
13   </>
14 )
15 }
```

Codeausschnitt 4.12: Darstellung der Freundesliste in *FriendList*

Die Darstellung von Daten in Arrays kann in React mittels der Array `map()` Funktion unkompliziert gehandhabt werden. Doch vor der Darstellung wird die Freundesliste noch nach dem `connected` Status sortiert, sodass Freunde die online sind zuerst angezeigt werden.

Dafür wird die Methode `sort()` verwendet. Diese nimmt eine Funktion mit zwei Parametern (hier `a` und `b`) entgegen, die jeweils ein Element des Arrays sind. Gibt uns die Funktion eine positive Zahl zurück, muss Element `a` vor Element `b` sortiert sein.

<sup>5</sup>Quelle: <https://react.dev/reference/react/useRef> am 09. Mai 2023

Bei negativen anders herum und bei 0 ist es gleichgültig. Indem die Booleschen Werte von einander subtrahiert werden, werden ihnen Zahlen zugeordnet: 1 für `true` und 0 für `false`. Nun haben wir die folgenden Szenarien: Wenn beide den gleichen Status haben ist der Wert 0. Ist `a.connected` true und `b.connected` ist false:  $1-0=1 \Rightarrow a$  wird vor `b` sortiert. Anders herum dem entsprechend  $0-1=-1$ .

Die Darstellung und Funktionen eines einzelnen Freundes übernimmt die *Friend*-Komponente, die ihre Daten als props übergeben bekommt.

Auch die Liste der Freundschaftsanfragen, die Liste der aktiven Spiele der *ActiveGames*- oder *Friend*-Komponente wird mittels der `map()` Funktion durch einzelne Komponenten dargestellt.

### connected-Event

Eine andere Interessante Methode der *FriendList*-Komponente ist der Listener auf das `connected` Event:

```

1 socket.on('connected', (status, username) => {
2     setFriends(prevFriends => {
3         return [...prevFriends].map(friend => {
4             if (friend.username === username) {
5                 friend.connected = status;
6             }
7             return friend;
8         });
9     });
10 });

```

Codeausschnitt 4.13: Der `connected` Eventlistener der *FriendList*-Komponente

Dieses Event wird ausgelöst, wenn ein Freund die Anwendung öffnet oder schließt beziehungsweise sich anmeldet oder abmeldet. Dabei wird die Freundesliste mit der Funktion `map()` durchlaufen und der Status des entsprechenden Freundes wird aktualisiert. Der Code zum Senden dieses Events vom Backend wird in Kapitel 4.2.1 dargestellt.

### Nutzung von `refreshKey`

```

1 //Home.js
2 ...
3 const [refreshKey, setRefreshKey] = useState(0);
4 const refreshData = useCallback(() => {
5     setRefreshKey((prevKey) => prevKey + 1);
6 }, [setRefreshKey, refreshKey]);
7
8 useEffect(() => {
9     refreshData();
10 }, [location.pathname]);
11 ...
12 //FriendList
13 useEffect(() => {
14     socket.emit('get_friends', ({friendList}) => {

```

```

15     setFriends(friendList);
16   });
17   socket.emit('get_friend_requests', ({requests}) => {
18     setFriendRequests(requests);
19   });
20 }, [socket, props.refreshKey]);

```

Codeausschnitt 4.14: Darstellung der Freundesliste in *FriendList*

`refreshKey` ist ein State, welcher den Komponenten *FriendList* und *ActiveGames* übergeben wird. Dieser State ist dient dazu die Freundesliste, Freundschaftsanfragen und aktive Partien aus dem Backend zu holen und zu aktualisieren. Wird beispielsweise von der *ChessGame*-Komponente auf die *Home*-Komponente navigiert, werden die Komponenten neu gerendert, doch die `useEffect`-Hooks werden nur neu ausgeführt, wenn eines der Elemente im Dependency Array geändert wurde. Um sicherzustellen, dass die Daten ordnungsgemäß aktualisiert werden wird der `refreshKey` dem Dependency Array hinzugefügt.

#### 4.1.4 Design

In diesem Abschnitt geht es um Code-Beispiele des Frontend Designs mittels Chakra UI. Wie schon in der Zielsetzung 1.2 besprochen ist das Design bisher nicht responsiv. Chakra UI stellt Komponenten zum Gestalten zur Verfügung, so ist zum Beispiel die Komponente `<Box>` vergleichbar mit einer `<div>` Komponente und `<Text>` mit `<p>`. Diesen Komponenten lassen sich Attribute geben, um die Anpassbarkeit der Elemente zu erhöhen.

```

1  ...
2  const backgroundBox = useColorModeValue("white", "purple.500");
3  ...
4  {user.loggedIn === true ?
5  <Box backgroundColor={backgroundBox} borderRadius="md" marginLeft={3}
6    p={6}
7    boxShadow="0 4px 6px -1px rgba(0, 0, 0, 0.1), 0 2px 4px -1px
8      rgba(0, 0, 0, 0.06)">
9    <Text marginBottom={1} fontSize="1.2rem"> Active Games: </Text>
10    <ActiveGames refreshKey={refreshKey}/>
11    <Text marginTop={5} fontSize="1.2rem"> Friends: </Text>
12    <FriendList refreshKey={refreshKey} times={times}/>
13  </Box>
14  :
15  <>
16  </>
17  }

```

Codeausschnitt 4.15: Darstellung der *Friendlist* und *ActiveGames* Komponenten in der *Home*-Komponente

In diesem Codeausschnitt der *Home*-Komponente wird die *FriendList* und *ActiveGames* gerendert siehe Abbildung 3.5).

Verwendet wird der Ternäre Operator, um die Komponenten nur anzuzeigen, falls der Benutzer angemeldet ist (siehe Zeile 4 des Codeausschnitts ??). Ist er nicht angemeldet wird das React Fragment `<> </>` gerendert. Ein React Fragment ermöglicht die Gruppierung von Komponenten ohne einen `<div>` Block verwenden zu müssen, da das React Fragment nicht in der DOM gerendert wird<sup>6</sup>. In diesem Fall dient das React Fragment dazu, nichts zu rendern, falls der Benutzer nicht angemeldet ist.

Die Chakra UI Syntax für die Attribute einer Komponente erinnert an die CSS Syntax. So steht das Attribut `p={6}` für das Padding<sup>7</sup>. Es handelt sich dabei allerdings nicht um sechs Pixel, sondern Chakra UI verwendet seine eigene Größenzuordnung. In diesem Fall entspricht das 1.5rem.

Um die Anwendung sowohl in einem dunklen Modus, als auch einem hellen Modus anzupassen, gibt es die Funktion `useColorModeValue`<sup>8</sup>. Diese Funktion lässt uns Werte je nach Modus definieren. In unserem Beispiel hat die `<Box>`-Komponente die Hintergrundfarbe Weiß im hellen Farbschema und Lila im dunklen Farbschema.

```

1 //App.js
2 ...
3 <ChakraBaseProvider theme={customTheme}>
4   ...
5 </ChakraBaseProvider>
6 ...
7
8 //themes.js
9 const customTheme = extendTheme({
10   overrides,
11   fonts: {
12     heading: 'Exo 2', sans-serif,
13     body: 'Exo 2', sans-serif,
14   },
15   components: {
16     Button: {
17       variants: {
18         'start-game-light': {
19           bg: "gray.200",
20           color: "black",
21           _hover: {
22             bg: "purple.200",
23           }
24         },
25         'start-game-dark': {
26           bg: "purple.700",
27           color: "white",
28           _hover: {
29             bg: "purple.600"
30           }
31         },
32       },
33     },
34   },
35   ...
36 })

```

<sup>6</sup>Quelle: <https://react.dev/reference/react/Fragment> am 11. Mai 2023

<sup>7</sup>Quelle: <https://chakra-ui.com/docs/styled-system/style-props> am 11. Mai 2023

<sup>8</sup>Quelle: <https://chakra-ui.com/docs/styled-system/color-mode> am 11. Mai 2023



```
33 //Home.js
34 ...
35 const startGameButton = useColorModeValue("start-game-light",
      "start-game-dark");
36 ...
37 <Button variant={startGameButton} ... > {time.string} </Button>
38 ...
```

Codeausschnitt 4.16: Verwendung von einem Thema mit Chakra UI

Chakra UI ermöglicht die Verwendung eines Themas, auf die man in allen Unterkomponenten der *ChakraBaseProvider*-Komponente (siehe Abschnitt 3.4.1) zugreifen kann<sup>9</sup>. In dem Beispielcode sieht man die Definition von zwei verschiedenen Buttons, die die Optik der Knöpfe in der *Home*-Komponente zum Starten eines Spiels definieren. Diese Optik wird in einigen anderen Komponenten wiederverwendet, weshalb sie in der *themes.js* Datei definiert werden, um eine bessere Wartbarkeit, Anpassbarkeit und Wiederverwendbarkeit zu gewährleisten.

## 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 Codeausschnitt 4.17 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);
```

<sup>9</sup>Quelle: <https://chakra-ui.com/docs/styled-system/customize-theme> am 11. Mai 2023

```

18
19 module.exports = router;

```

Codeausschnitt 4.17: Ausschnitt aus index.js und die Datei authRouter.js

Für jede Anfrage auf den Pfad `/auth` wird die Middleware `rateLimiter` verwendet (siehe Zeile 11 des Codeausschnitts 4.17). Dieser achtet darauf, dass nicht zu viele Anfragen einer IP Adresse gesendet werden.

```

1  const redisClient = require("../redis/redis.js");
2
3  module.exports.rateLimiter = (secondsLimit, limitAmount) => async (req,
4    res, next) => {
5    const ip = req.connection.remoteAddress;
6    [response] = await redisClient
7      .multi()
8      .incr(ip)
9      .expire(ip, secondsLimit)
10     .exec();
11    if (response[1] > limitAmount) {
12      res.json({
13        loggedIn: false,
14        message: "Slow down!! Try again in a minute.",
15      });
16    } else next();
17  };

```

Codeausschnitt 4.18: 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<sup>10</sup>. Dieser Eintrag wird nach Ablauf des Zeitfensters gelöscht und falls das Anfragelimit innerhalb dieses Zeitfensters überschritten wurde, wird eine Fehlermeldung gesendet. 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 (siehe Zeile 13 des Codeausschnitts 4.17).

```

1  const getJwtFromCookie = req => {
2    return req.headers["cookie"] &&
3      cookie.parse(req.headers["cookie"])["jwt"];
4  };
5
6  module.exports.handleLogin = (req, res) => {
7    const token = getJwtFromCookie(req);
8    if (!token) {
9      res.json({loggedIn: false});
10   }
11 };

```

<sup>10</sup>Quelle: <https://redis.io/commands/incr/> am 11. Mai 2023

```

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   };

```

Codeausschnitt 4.19: Die handleLogin Middleware zum Authentifizieren mit Cookie

Diese Middleware läßt 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 module.exports.attemptLogin = async (req, res) => {
2   const potentialLogin = await query(
3     "SELECT id, username, password, userid FROM users u WHERE
u.username=$1",
4     [req.body.username]
5   );
6   if (potentialLogin.rowCount > 0) {
7     const isSamePassword = await bcrypt.compare(
8       req.body.password,
9       potentialLogin.rows[0].password
10    );
11    if (isSamePassword) {
12      const user = {
13        username: req.body.username,
14        userid: potentialLogin.rows[0].userid,
15      }
16      jwt.sign(
17        user,
18        process.env.JWT_SECRET,
19        {expiresIn: "24h"},
20        (err, token) => {
21          if(err) {
22            res.json({loggedIn: false, message: "Something went
wrong, try again later"});
23          } else {
24            const jwtCookie = cookie.serialize("jwt", token, {
25              httpOnly: true,
26              secure: process.env.NODE_ENV === "production",
27                // Secure flag only in production
28              maxAge: 24 * 60 * 60, // 24 hours

```

```

28         sameSite: "lax",
29         path: "/"
30     });
31     res.setHeader("Set-Cookie", jwtCookie);
32     res.json({loggedIn: true, username: user.username});
33 }
34 }
35 );
36 } else {
37     res.json({loggedIn: false, message: "Wrong username or
38         password!"});
39 }
40 } else {
41     res.json({loggedIn: false, message: "Wrong username or
42         password!"});
43 }
44 }
45 }
46 };

```

Codeausschnitt 4.20: Die attemptLogin Middleware zum Anmelden

Die Middleware `attemptLogin` überprüft zunächst, ob ein Benutzer mit diesem Benutzernamen in der PostgreSQL-Datenbank existiert und mittels `bcrypt` ob die Passwörter übereinstimmen. Dafür wird die `query`-Funktion (siehe Kapitel 4.3) genutzt, indem in der Tabelle `users` nach einem Eintrag mit dem Benutzernamen gesucht wird. Der SQL-String verwendet den Platzhalter `$1`, der durch den Wert `[req.body.username]` ersetzt wird<sup>11</sup>. Existiert kein Benutzer mit diesem Benutzernamen oder ist das Passwort falsch 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. Ansonsten wird eine zufällige `userid` erstellt, das Passwort mit `bcrypt` verschlüsselt, der Tupel in der Datenbank gespeichert und der Cookie mit dem JWT-Token gesetzt.

## Abmelden

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

```

1 module.exports.handleLogout = (req, res) => {

```

<sup>11</sup>Quelle: <https://node-postgres.com/features/queries> am 11. Mai 2023

```
2     res.setHeader(  
3         "Set-Cookie",  
4         cookie.serialize("jwt", "", {  
5             httpOnly: true,  
6             secure: process.env.NODE_ENV === "production",  
7             maxAge: -1,  
8             sameSite: "lax",  
9             path: "/"  
10        })  
11    );  
12    res.sendStatus(204);  
13 }
```

Codeausschnitt 4.21: Middleware zum Abmelden

Eine Anfrage zum Abmelden wird auf den Pfad `auth/logout` gestellt unter der die Middleware aus Codeausschnitt 4.21 durchlaufen wird. In dieser wird ein leerer (siehe Zeile 4), abgelaufener (siehe Zeile 7) „jwt“-Cookie gesetzt, der bewirkt, dass der bisherige „jwt“-Cookie gelöscht wird. Dann wird dem Frontend mit dem Code 204 geantwortet (siehe Zeile 12), welcher aussagt, dass die Anfrage erfolgreich bearbeitet wurde, jedoch keine Daten mit der Antwort gesendet werden<sup>12</sup>.

### Socket.io Authentifizierung und Middleware

Im Codeausschnitt 4.22 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.

```
1  ...  
2  const corsConfig = {  
3      origin: process.env.FRONTEND_URL,  
4      credentials: true,  
5  }  
6  const server = require('http').createServer(app);  
7  const io = new Server(server, {  
8      cors: corsConfig  
9  });  
10 io.use(authorizeUser);  
11 io.use(initializeUser);  
12 io.use((socket, next) => {  
13     initializeChessListeners(socket, io);  
14     initializeListeners(socket, io);  
15     next();  
16 });  
17 ...
```

Codeausschnitt 4.22: Ausschnitt der `index.js` Datei mit Socket.io Server Erstellung und Middleware Zuweisung

Die Initialisierung des Socket.io Servers benötigt die `corsConfig` Konfiguration, um Verbindungen vom Frontend zuzulassen und Header Informationen wie Cookies mitzusenden.

<sup>12</sup>Quelle <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204> am 05. Mai 2023

Es werden drei Middlewares bei einem Verbindungsaufbau durchlaufen, die ich in diesem Kapitel mit Codeausschnitten erläutern werde.

```
1 module.exports.authorizeUser = (socket, next) => {
2   let token = null;
3   if (socket.request.headers.cookie) {
4     token = cookie.parse(socket.request.headers.cookie).jwt;
5     if (token) {
6       jwt.verify(token, process.env.JWT_SECRET, (err,
7         decodedPayload) => {
8         if (err) {
9           next(new Error('Unable to Read Token'));
10          return;
11        }
12        socket.user = {...decodedPayload};
13      });
14    }
15    next();
16  }
```

Codeausschnitt 4.23: authorizeUser Middleware für Socket.io Verbindungen

Die Middleware `authorizeUser` (siehe Codeausschnitt 4.23) authentifiziert den Benutzer. Dabei wird auf den Cookie zugegriffen, der gegebenenfalls gesetzt wurde, um den Benutzernamen und die `userid` als Attribute der `socket` zu setzen (siehe Zeile 11 des Codeausschnitts 4.23). 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 module.exports.initializeUser = async (socket, next) => {
2   if (socket.user) {
3     socket.join(socket.user.userid);
4     await setUser(socket.user.username, socket.user.userid, true);
5     getFriends(socket.user.username).then(async friends => {
6       const friendRooms = friends.map(friend => friend.userid);
7       if (friendRooms.length > 0) {
8         socket.to(friendRooms).emit("connected", "true",
9           socket.user.username);
10      }
11    });
12  }
13  next();
14 }
```

Codeausschnitt 4.24: initializeUser Middleware für Socket.io Verbindungen

In der Middleware `initializeUser` wird der Nutzer als online vermerkt, seine Freunde werden darüber informiert, dass er nun online ist und er tritt dem Raum seiner `userid` bei. Dies passiert natürlich nur, wenn in der vorherigen Middleware der Benutzer authentifiziert werden konnte. Die Funktion `setUser` setzt den Eintrag `user:username` mit

der `userid` und `connected` auf `true`. Die Methode `getFriends` (siehe Kapitel ??) holt die Benutzernamen und die `userids` aller Freunde des Benutzers und mittels der `map` Funktion wird an alle `userids` der Freunde das Event `connected` gesendet. Dadurch wird im Frontend der Spieler als online angezeigt (siehe Abschnitt 4.1.3).

Die dritte Middleware setzt alle nötigen Listener für Events, die mit dem Schachspiel zusammenhängen, als auch sonstige Funktionen (siehe Zeilen 12 ff. des Codeausschnitts 4.22).

```
1 module.exports.onDisconnect = async (socket) => {  
2   if (!socket.user) {  
3     return;  
4   }  
5   await setUser(socket.user.username, socket.user.userid, false);  
6   getFriends(socket.user.username).then(friends => {  
7     const friendRooms = friends.map(friend => friend.userid);  
8     if (friendRooms.length > 0) {  
9       socket.to(friendRooms).emit("connected", "false",  
10        socket.user.username);  
11     }  
12   });  
13 }
```

Codeausschnitt 4.25: Die `onDisconnect` Methode für Sockets

Die `onDisconnect` Methode aus Codeausschnitt 4.25 wird aufgerufen, sobald eine socket Verbindung getrennt wird. Hierfür dient der `socket.disconnect()` Aufruf des Frontends (siehe Abschnitt 4.1.1). Ähnlich wie bei der `initializeUser` Middleware (siehe Codeausschnitt 4.24) wird dabei in Redis der Spieler als offline vermerkt und seine Freunde werden mit dem Event `connected` darüber in Kenntnis gesetzt.

### 4.2.2 Das Schachspiel

Das Schachspiel verwaltet die Datei `socketChessController.js`, welche eine der Funktionen zur Listener Initialisierung bereitstellt (siehe `inititalizeChessListeners` aus Codeausschnitt 4.22), wobei `ServerChessClock.js` die Schachuhren bereitstellt.

In diesem Abschnitt werde ich einige Aspekte der Implementierung des Schachspiels mit Code Beispielen erläutern. Funktionen wie Aufgeben, das Ende der Partie oder Chat Nachrichten werden im Abschnitt 3.5.2 erläutert und zeichnen sich durch geringere Komplexität aus, weshalb sie in diesem Abschnitt nicht nochmals behandelt werden. Auch das Finden eines Gegners wird in dem Abschnitt der Backend-Architektur detailliert beschrieben, weshalb dies hier nicht noch einmal erläutert wird.

#### Initialisierung eines Spiels

Die Suche und das Finden eines Gegners wurde in Abschnitt 3.5.2 erläutert. In diesem Abschnitt möchte ich genauer darauf eingehen, wie ein Spiel nach gefundenem Gegner initialisiert wird. Dabei handelt es sich um die `createChessGame` Methode (für Kontext siehe Sequenzdiagramm in Abbildung 3.19).

```

1  const createChessGame = async (io, username1, username2, time) => {
2      const roomId = UUIDv4();
3      const [whitePlayer, blackPlayer] = Math.random() < 0.5 ? [username1,
4          username2] : [username2, username1];
5
6      const chessInstance = await
7          import('../chess/Chess.mjs').then(ChessFile => {
8              return ChessFile.Chess();
9          });
10     const d = new Date();
11     chessInstance.header('White', whitePlayer, 'Black', blackPlayer,
12         'Date', d.toString());
13     //Store Game in Redis
14     await initializeGame(roomId, whitePlayer, blackPlayer, time,
15         chessInstance.pgn());
16     const chessClock = new ServerChessClock(time);
17     currentChessClocks[roomId] = {chessClock};
18
19     chessClock.startStartingTimer('white');
20     chessClock.ChessClockEvents.on('cancel_game', () => {
21         io.to(roomId).emit('cancel_game');
22         io.to(roomId).emit('stop_clocks');
23         endGame(roomId)
24     });
25     chessClock.ChessClockEvents.on('time_over', (color) => {
26         io.to(roomId).emit('time_over', color);
27         io.to(roomId).emit('stop_clocks');
28         endGame(roomId);
29     });
30     return roomId;
31 }

```

Codeausschnitt 4.26: 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<sup>13</sup> (siehe Codeausschnitt 4.27).

```

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

```

Codeausschnitt 4.27: Die Chess.mjs Datei

Mit diesem chess.js Objekt generieren wir eine PGN-Notation, in die die Benutzernamen der Spieler und das Datum eingetragen sind. Diese Informationen werden mittels der

<sup>13</sup>Quelle: <https://stackoverflow.com/questions/70396400/how-to-use-es6-modules-in-commonjs> am 06. Mai 2023



`initializeGame` Methode in Redis unter `game:roomId` gespeichert. Falls die Spieler angemeldet sind wird die `roomId` auch in die Listen `activeGames` der `user:username` Einträge der beiden Spieler hinzugefügt.

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 4.2.2 erläutert. Eventlistener für die Events `time_over` und `cancel_game` der `ServerChessClock` werden definiert. `cancel_game` ist ein Event, welches ausgelöst wird, falls eine Startzeit abgelaufen ist und `time_over`, falls eine reguläre Zeit abgelaufen ist. Da die Schachuhren und das Schachspiel möglichst Modular gehalten werden, wird an das Frontend in diesen Listenern immer zwei Events gesendet: `time_over`, beziehungsweise `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. Was bei diese Methode bewirkt wird in Kapitel 3.5.2 erläutert.

## Neue Züge

Beim Eingang eines neuen Zugs mittels des `new_move` Events wird die Methode `newMove` ausgeführt. Sie benötigt die socket des Spielers und die `Socket.io` Server Instanz. Die restlichen Daten die sie benötigt werden vom Frontend mit dem Event mitgesendet.

Ein Aktivitätsdiagramm und eine ausführliche Erklärung des Ablaufs wird in Abschnitt 3.5.2 behandelt. In diesem Abschnitt konzentrieren wir uns auf die Umsetzung dieser Aktivitäten im Code.

```

1 //Listener:
2 client.on('new_move', newMove(client, io));
3 //Methode
4 const newMove = (socket, io) => async (roomId, player, move, cb) => {
5   if(!currentChessClocks[roomId]) {
6     cb({done: false, errMsg: "Game does not exist"});
7     return;
8   }
9   const {chessClock} = currentChessClocks[roomId];
10  const chessInstance = await
11    import('../chess/Chess.mjs').then(ChessFile => {
12      return ChessFile.Chess();
13    });
14  try {
15    chessInstance.loadPgn(await getGame(roomId, "pgn").then(pgn => {
16      if(!pgn) {
17        cb({done: false, errMsg: "Game does not exist"});
18        return;
19      }
20      return pgn;
21    }));
22    chessInstance.move(move)
23  } catch(error) {
24    cb({done: false, errMsg: error});
25    return;
26  }

```

```

25     }
26
27     socket.to(roomId).emit('opponent_move', move);
28     //Store move in Redis
29     newChessMove(chessInstance.pgn(), roomId).then(r => cb({done:
        true}));
30
31     if(chessInstance.isGameOver()) {
32         if (chessInstance.isCheckmate()) {
33             io.to(roomId).emit('checkmate', socket.user.username);
34         } else {
35             io.to(roomId).emit('draw');
36         }
37         io.to(roomId).emit('stop_clocks');
38         chessClock.ChessClockEvents.emit('stop');
39         endGame(roomId);
40         cb({done: true});
41         return;
42     }
43
44     chessClock.ChessClockEvents.emit('toggle', ({remainingTimeWhite,
        remainingTimeBlack, turn}) => {
45         io.to(roomId).emit('updated_time', remainingTimeWhite,
            remainingTimeBlack, turn);
46     });
47
48     if (chessInstance.history().length === 1) {
49         chessClock.startStartingTimer('black');
50         io.to(roomId).emit('stop_starting_time_white');
51     } else if (chessInstance.history().length === 2) {
52         chessClock.startTimer('white');
53         io.to(roomId).emit('stop_starting_time_black');
54     }
55 }

```

Codeausschnitt 4.28: newMove Methode die beim Eingang eines neuen Zugs aufgerufen wird

Die Methode holt erst das ServerChessClock Objekt aus dem currentChessClocks Array und importiert die aktuelle PGN-Notation des Schachspiels aus der Redis Datenbank in ein chess.js Objekt und macht den Zug darauf (siehe Zeilen 5 - 25 des Codeausschnitts 4.28).

Die Kommunikation mit dem Frontend erfolgt über verschiedene Events wie `opponent_move`, `checkmate`, `draw`, `stop_clocks`, `updated_time`, `stop_starting_time_white` und `stop_starting_time_black`. Diese Events behandeln entweder ein Ereignis der Schachpartie oder verwalten die Schachuhren. Über die Callback Funktion wird mitgeteilt, ob der Zug erfolgreich behandelt werden konnte oder nicht.

Mit dem ServerChessClock Objekt wird mittels der Events `stop` und `toggle` (siehe Abschnitt 4.2.2) und den Funktionsaufrufen `startStartingTimer` und `startTimer` kommuniziert. `stop` wird bei einem Ende der Partie gesendet, `toggle` bei jedem neuen Zug,

es sei denn das Spiel ist vorbei und die Funktionsaufrufe dienen zum initialen starten der Startuhren oder der regulären Uhr.

Des weiteren wird die aktualisierte PGN-Notation des Spiels mit der Funktion `newChessMove` (siehe Zeile 28 im Codeausschnitt 4.9) in Redis gespeichert.

### Senden des aktuellen Zustands einer Partie

Beim Laden der *ChessGame*-Komponente im Frontend wird das Event `get_game_data` gesendet, um den aktuellen Zustand des Spiels zu erhalten und spielen zu können beziehungsweise zugucken zu können.

```

1 client.on('get_game_data', (roomId, guestName, cb) => {
2   if (!currentChessClocks[roomId]) {
3     cb({done: false, errMsg: "This Game does not exist"});
4     return;
5   }
6   const {chessClock} = currentChessClocks[roomId];
7   if (!client.rooms.has(roomId)) {
8     client.join(roomId);
9   }
10  if(guestName) {
11    client.user = {username: guestName}
12  }
13  getGame(roomId)
14    .catch(() => {
15      cb({done: false, errMsg: "This Game does not exist"})
16    })
17    .then(game => {
18      if (!game) {
19        cb({done: false, errMsg: "This Game does not exist"});
20        return;
21      }
22      game.currentState = chessClock.getCurrentMode();
23      if (game.currentState.includes('s')) {
24        game.currentStartingTimer =
25          chessClock.getCurrentStartingTimer();
26      } else {
27        game.currentTimes = chessClock.getCurrentTimes();
28      }
29      cb({done: true, data: game});
30    })
31  });

```

Codeausschnitt 4.29: Der Listener des `get_game_data` Events zum übermitteln des Schachspiels

Es gibt zwei Szenarien, in denen kein Zustand des Schachspiels gesendet werden kann: Falls kein `ServerChessClock` Objekt in dem `currentChessClocks` Array vorhanden ist (siehe Zeile 2 des Codeausschnitts 4.29) oder falls kein Redis Eintrag zu diesem Spiel gefunden werden konnte (siehe Zeile 15 & 18 des Codeausschnitts 4.29).

Essenziell bei diesem Listener ist, dass die socket gegebenenfalls der roomId beitrifft, um zukünftige Events, wie neue Züge, erhalten zu können. Außerdem ist es wichtig den Benutzernamen zu setzen, falls es sich um einen Gast handelt. Denn wenn dieser die Seite neu lädt wird eine neue socket Verbindung hergestellt und der Gast-Benutzername wäre nicht mehr in der socket gespeichert. Allerdings wird auf den Benutzernamen der socket beispielsweise bei einem Schachmatt zugegriffen.

Die aktuellen Daten, wie welcher Spieler welche Farbe spielt und die aktuelle PGN-Notation des Spiels werden aus Redis mittels der Funktion `getGame` aus `game:roomId` geholt. Fehlen nur noch die aktuellen Zeiten. Dafür verwenden wir `getCurrentMode()`, um den aktuellen Zustand der Schachuhren zu bekommen (siehe Zeile 23). Dieser besteht wie im Frontend aus „tw“, „tb“, „sw“ und „sb“ (siehe 4.1.2).

Je nachdem welche Uhren gerade laufen werden entweder die Startzeiten oder die regulären Zeiten der Spieler den Redis Daten hinzugefügt und anschließend dem Frontend per Callback Funktion übermittelt.

Hier wird der Sinn hinter der Speicherung des `ServerChessClock` Objekts in einem Array ersichtlich. Um die aktuelle Zeit abzurufen muss auf das `ServerChessClock` Objekt zugegriffen werden. Eine Kritik dieser Speicherung der aktuellen Zeiten und Alternativen werden in Abschnitt 5.2.1 erläutert.

## Die Uhr

Die serverseitige Schachuhr wird von Objekten der Klasse `ServerChessClock` verwaltet. Diese Objekte besitzen die folgenden Attribute:

```

1 function ServerChessClock(time) {
2     this.timeMode = time;
3     this.currentMode = 'off';
4     this.remainingTimeWhite = {minutes: time.minutes, seconds: 0};
5     this.remainingTimeBlack = {minutes: time.minutes, seconds: 0};
6     this.ChessClockEvents = new EventEmitter();
7     this.startingTimeWhite = {seconds: 0};
8     this.startingTimeBlack = {seconds: 0};
9     switch (time.type) {
10        case 'Bullet': this.startingTimeWhite.seconds =
11                        this.startingTimeBlack.seconds = 15; break;
12        case 'Blitz': this.startingTimeWhite.seconds =
13                     this.startingTimeBlack.seconds = 20; break;
14        case 'Rapid': this.startingTimeWhite.seconds =
15                     this.startingTimeBlack.seconds = 30; break;
16        case 'Classical': this.startingTimeWhite.seconds =
17                          this.startingTimeBlack.seconds = 45; break;
18        default: this.startingTimeWhite.seconds =
19                 this.startingTimeBlack.seconds = "unknown"; break;
20    }
21 }
```

Codeausschnitt 4.30: Konstruktor der `ServerChessClock` Klasse

Vorab stelle ich vor, wie ein `time` (siehe Zeile 1 des Codeausschnitts 4.30) Objekt aufgebaut ist:

```
{type: 'Rapid', minutes: 15, increment: 10, string: '15 + 10'}
```

`type` klassifiziert die Schachuhr Konfigurationen zu verschiedenen Modi. Diese Klassifikationen der Schachuhren sind auf online Schach-Plattformen wie [lichess.org](https://lichess.org) bereits Standard. Die Gründe dafür liegen vor allem in der Erweiterbarkeit (siehe //REF). Bisher werden sie genutzt um wie in Zeilen 9 ff. des Codeausschnitts 4.30 verschieden lange Startzeiten zu initialisieren.

`string` dient der Darstellung und der einfachen Identifizierung dieser Objekte.

```

1  ServerChessClock.prototype.startTimer = function (color) {
2      this.currentMode = color === "white" ? "tw" : "tb";
3      const currentPlayerTime = color === "white" ?
4          this.remainingTimeWhite : this.remainingTimeBlack;
5
6      const isTimeOver = () => {
7          return currentPlayerTime.minutes === 0 &&
8              currentPlayerTime.seconds === 0;
9      };
10
11     const decreaseTime = () => {
12         if (currentPlayerTime.seconds === 0) {
13             currentPlayerTime.minutes -= 1;
14             currentPlayerTime.seconds = 59;
15         } else {
16             currentPlayerTime.seconds -= 1;
17         }
18     };
19
20     const timer = setInterval(() => {
21         decreaseTime();
22         if (isTimeOver()) {
23             clearInterval(timer);
24             this.stopCurrentGame();
25         }
26     }, 1000);
27
28     //If game ends, because player resigned, checkmate, ...
29     this.ChessClockEvents.once("stop", () => {
30         clearInterval(timer);
31     });
32
33     //If a player played a new move
34     this.ChessClockEvents.once("toggle", (cb) => {
35         clearInterval(timer);
36         if (color === "white") {
37             this.remainingTimeWhite = increment(
38                 currentPlayerTime,
39                 this.timeMode.increment

```

```

40         );
41         cb({
42             remainingTimeWhite: this.remainingTimeWhite,
43             remainingTimeBlack: this.remainingTimeBlack,
44             turn: "tb",
45         });
46         this.startTimer("black");
47     } else {
48         this.remainingTimeBlack = increment(
49             currentPlayerTime,
50             this.timeMode.increment
51         );
52         cb({
53             remainingTimeWhite: this.remainingTimeWhite,
54             remainingTimeBlack: this.remainingTimeBlack,
55             turn: "tw",
56         });
57         this.startTimer("white");
58     }
59 });
60 };
61
62 ServerChessClock.prototype.stopCurrentGame = function () {
63     if (this.getCurrentMode().includes('s')) {
64         this.ChessClockEvents.emit('cancel_game');
65     } else if (this.getCurrentMode().includes('w')) {
66         this.ChessClockEvents.emit('time_over', 'white');
67     } else {
68         this.ChessClockEvents.emit('time_over', 'black');
69     }
70 }

```

Codeausschnitt 4.31: Die `startTimer` und `stopCurrentGame` Methoden der `ServerChessClock` Klasse

Die Funktion `startTimer` wird ein mal von `SocketChessController` aufgerufen, falls der zweite Zug gespielt wurde (siehe Zeile 52 des Codeausschnitts 4.28). Anschließend wird die Methode selbst rekursiv aufgerufen, falls ein neuer Zug gespielt wurde und das `toggle` Event empfangen wurde (siehe Zeile 46 & 57 im Codeausschnitt 4.31).

Je nachdem welche Zeit läuft, wird `currentMode` aktualisiert und die entsprechende Zeit wird in `currentPlayerTime` referenziert (siehe Zeilen 2 & 3 des Codeausschnitts 4.31). Die Zeit läuft durch den `timer` ab, welcher `setInterval` nutzt, um jede Sekunde mittels `decreaseTime` die Zeit ablaufen zu lassen und `isTimeOver` um zu überprüfen ob die Zeit abgelaufen ist (siehe Zeile 19 ff. des Codeausschnitts 4.31).

Es wird ein Listener auf das Event `toggle` definiert, welcher bei einem neuen Zug das Inkrement auf die entsprechende Zeit rechnet und mittels Callback Funktion die aktuellen Zeiten durchgibt (siehe Zeilen 41 ff. und 52 ff. des Codeausschnitts 4.31), um diese in der `newMove` Methode an das Frontend zu senden (siehe Kapitel 4.2.2). Dieser Listener wird mit `ones(...)` definiert, wodurch er nur ein mal auf das Event hört. Dies ist wichtig, da anschließend die `startTimer` Methode rekursiv aufgerufen wird und der Listener neu

initialisiert wird.

Die `stopCurrentGame` Methode wird aufgerufen, falls eine Zeit (Startzeit oder reguläre Zeit) abgelaufen ist und sendet entsprechende Events an `SocketChessController`. Wie diese Events empfangen werden ist in Codeausschnitt 4.26 enthalten.

### 4.2.3 Verwaltung von Freunden

In diesem Kapitel zeige ich die Implementierung von dem Versenden von Freundschaftsanfragen und dem senden der Freunde an das Frontend. Die zugehörigen Redis Funktionen werden in Kapitel 4.3 erläutert.

Alle anderen Funktionen welche nicht mit dem Schachspiel und der Authentifizierung zusammenhängen werden durch die Listener in der Datei `socketController.js` zur Verfügung gestellt und verwenden den `redisController` für Operationen an der Redis Datenbank. Der Ablauf von Funktionen im Backend bezüglich Freunden wurde ausführlich in dem Kapitel 3.5.3 behandelt.

#### Versenden einer Freundschaftsanfrage

```

1 client.on('send_friend_request', (requestName, cb) => {
2   if(client.user) {
3     requestFriend(client.user.username, client.user.userid,
4       requestName)
5     .catch(err => {
6       cb({done: false, errorMsg: "Please try again later"});
7       return;
8     })
9     .then(result => {
10      if (result.errorMsg) {
11        cb({done: false, errorMsg: result.errorMsg});
12        return;
13      }
14      cb({done: true});
15      client.to(result.userid).emit('friend_request',
16        client.user.username);
17    })
18  } else {
19    cb({done: false, errorMsg: "Please try again later"});
20  }
21 });

```

Codeausschnitt 4.32: Der Listener des `send_friend_request` Events

Die Funktion `requestFriend` der `redisController` Datei überprüft, ob eine Freundschaft der beiden Spieler erlaubt wäre und gibt uns unter anderem die `userid` des angefragten Spielers zurück und speichert die Freundschaftsanfrage in Redis (siehe Kapitel 4.3.1). Ist die Freundschaft nicht erlaubt wird die entsprechende Fehlermeldung, warum sie nicht erlaubt ist an den Sender mittels Callback Funktion weitergeleitet. Ansonsten

wird die Freundschaftsanfrage mittels des Events `friend_request` an den Benutzer gesendet und der Sender wird über die Callback Funktion über den erfolgreichen Versand benachrichtigt.

### Das Senden der Freunde ans Frontend

Beim rendern der *Home*-Komponente werden Events vom Frontend gesendet, um Daten wie Freunde, Freundschaftsanfragen und aktive Spiele aus der Redis Datenbank mittels Callback Funktion zu holen. Eine davon ist das `get_friends` Event.

```
1 client.on('get_friends', async (cb) => {
2   if(client.user) {
3     const friends = await getFriends(client.user.username);
4     const parsedFriends = await parseFriendList(friends);
5     cb({friendList: parsedFriends});
6   }
7 });
```

Codeausschnitt 4.33: Der `get_friends` Listener

Die `get_friends` Methode liefert uns die Benutzernamen und `userids` aller Freunde (siehe Kapitel 4.3.1), allerdings benötigt das Frontend noch weitere Daten. Das sind die aktiven Partien des Benutzers und den Status ob er gerade online ist oder nicht. Diese Daten ergänzt die Methode `parseFriendList` (siehe Kapitel 4.3.1).

## 4.3 Datenbank Integration

Die Herstellung einer Verbindung zu der PostgreSQL und zu der Redis Datenbank wird jeweils in einer Datei hergestellt:

```
1 require('dotenv').config()
2 const Pool = require('pg').Pool
3
4 const pool = new Pool({
5   user: process.env.DATABASE_USER,
6   host: process.env.DATABASE_HOST,
7   database: process.env.DATABASE_NAME,
8   password: process.env.DATABASE_PASSWORD,
9   port: process.env.DATABASE_PORT,
10 })
11
12 const query = (text, params) => pool.query(text, params)
13 module.exports = { query }
```

Codeausschnitt 4.34: Initialisierung der PostgreSQL Datenbank und einer Methode für die Anfragen

```
1 const Redis = require('ioredis');
2
3 const redisClient = new Redis();
```



```

4
5 module.exports = redisClient;

```

Codeausschnitt 4.35: Initialisierung einer Redis-Datenbank Verbindung

Die Daten zum Herstellen einer Verbindung sind in einer `.env` Datei gespeichert, um die Sicherheit und Wartbarkeit zu erhöhen, da `.env` Dateien standardmäßig zu Dateien zählen, welche nicht in Git aufgenommen werden.

Wie die PostgreSQL Datenbank genutzt wird ist in Kapitel 4.2.1 erläutert.

### 4.3.1 Zugriff auf Redis mit dem `redisController`

Alle Operationen auf der Redis Datenbank werden von der Datei `redisController` zur Verfügung gestellt (mit Ausnahme der `rateLimiter` Middleware). Welche Datentypen unter welchen Keys in Redis gespeichert werden, wird in Abschnitt 3.6.2 erläutert. Die Datei enthält 19 Methoden, die fast alle auf die Datenbank zugreifen. In diesem Abschnitt werden exemplarisch einige wichtige Methoden vorgestellt.

#### `addPlayerInQueue`

```

1 module.exports.addPlayerInQueue = async (loggedIn, time, user) => {
2   return new Promise((resolve, reject) => {
3     const key = loggedIn === true ? 'waitingPlayers${time}' :
4       'waitingGuests${time}';
5     redisClient.lpush(key, `${user:username}:${user:userid}`,
6       (err, result) => {
7       if (err) {
8         reject(err);
9       } else {
10        resolve(result);
11      }
12    });
13  });
14 };

```

Codeausschnitt 4.36: Methode um einen Spieler in eine Warteschlange hinzuzufügen

Die `addPlayerInQueue` Methode hat die Funktion einen Spieler in eine Warteschlange für einen zufälligen Gegner hinzuzufügen. Je nachdem ob er angemeldet ist oder nicht wird die Warteschlange `waitingPlayers` oder `waitingGuests` mit der entsprechenden Zeit als Key genutzt, um `username:userid` in die Warteschlange zu schreiben.

Die Funktion `lpush` fügt den Spieler am Anfang der Liste ein. Herausgenommen werden Spieler mittels der Funktion `rpop`, die Spieler am Ende der Liste entnehmen. So wird gewährleistet, dass der Spieler der am längsten wartet zu erst entnommen wird.

Implementiert ist die Funktion durch eine `Promise`, welche es ermöglicht eine Fehlerbehandlung oder ein Ergebnis einer asynchronen Methode getrennt zu behandeln<sup>14</sup>, beziehungsweise in diesem Fall weiter zu geben.

<sup>14</sup>Quelle: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using\\_promises?retiredLocale=de](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises?retiredLocale=de) am 14. Mai 2023

### getFriends

```
1 const getFriends = async (username) => {
2   const friendList = await redisClient.lrange('friends:${username}',
3     0, -1);
4   return friendList.map(friend => {
5     return {username: friend.split(':')[0], userid:
6       friend.split(':')[1]};
  });
}
```

Codeausschnitt 4.37: Methode um Freunde eines Spielers zu erhalten

Die Methode `getFriends` holt alle Freunde eines Spielers aus dem key `friend:username`. Mittels der Funktion `lrange` können Elemente aus einer Liste geholt werden. Der Parameter 0 repräsentiert den Start, ab welchem Index die Freunde geholt werden sollen und der Parameter `-1` das Ende. In diesem Fall heißt `-1`, dass alle Elemente aus der Liste geholt werden sollen. Die Daten werden im letzten Schritt mittels der Array `map()` Funktion noch vernünftig aufbereitet um die Nutzung der Daten zu vereinfachen.

### parseFriendList

```
1 const parseFriendList = async (friendList) => {
2   const parsedFriendList = [];
3   for (let friend of friendList) {
4     const parsedFriend = await getUserData(friend.username);
5     const activeGames = {};
6     for (let activeGame of JSON.parse(parsedFriend.activeGames)) {
7       activeGames[activeGame] = await getGame(activeGame);
8     }
9     parsedFriendList.push({
10      username: friend.username,
11      userid: friend.userid,
12      connected: parsedFriend.connected,
13      activeGames: activeGames
14    });
15  }
16  return parsedFriendList;
17 }
```

Codeausschnitt 4.38: Methode um alle Informationen eines Arrays von Freunden zu bekommen

Die Methode `parseFriendList` hat die Funktion alle Daten aller Freunde einer Freundesliste zu holen. Sie dient als Erweiterung der `getFriends` Methode, falls mehr Daten nötig sind, als lediglich der Benutzername und die `userid`. Dies ist beispielsweise für die Anzeige im Frontend der Fall (siehe Kapitel 4.2.3).

Mit einer `for`-Schleife wird das Array der Freunde durchlaufen und für jeden Spieler werden folgende Daten gesammelt:

- Online-Status & aktive Partien: Aus dem Redis Eintrag `user:username` wird mittels der `getUserData` Methode alle Werte des `user:username` Eintrags des Freundes geholt. Dazu zählen der online-Status und die roomIds der aktiven Partien
- Daten der Aktiven Partien: Anschließend wird für jede aktive Partie dieses Freundes die Daten aus `game:roomId` mittels der Funktion `getGameData` abgerufen. Dies ist wichtig um im Frontend anzeigen lassen zu können, welche Farbe und gegen wen der Freund spielt.

Anschließend werden diese Daten in einem Objekt gesammelt und der neuen Freundesliste hinzugefügt, welche am Ende zurückgegeben wird.

### requestFriend und friendIsValid

```

1 module.exports.requestFriend = async (username, userid, requestName) => {
2   const requestedFriend = await friendIsValid(username, requestName);
3   if (requestedFriend.errorMsg) {
4     return {errorMsg: requestedFriend.errorMsg};
5   }
6   const currentFriendRequests = await
7     redisClient.lrange('friend_requests:${requestName}', 0, -1);
8   if (currentFriendRequests.includes([requestName,
9     requestedFriend.userid].join(':'))) {
10    return {errorMsg: "You've already sent a friend request to this
11      user"};
12  }
13  await redisClient.lpush('friend_requests:${requestName}', [username,
14    userid].join(":"));
15  return {userid: requestedFriend.userid};
16 }
```

Codeausschnitt 4.39: requestFriend Methode um eine Freundschaftsanfrage in redis zu speichern

Die `requestFriend` Methode wird beim Versenden einer Freundschaftsanfrage aufgerufen (siehe Kapitel 4.2.3), um zu überprüfen, ob eine Freundschaftsanfrage legal wäre und diese gegebenenfalls in Redis zu speichern.

Die Methode `friendIsValid` überprüft, ob eine Freundschaft zulässig ist (siehe Codeausschnitt 4.40). Ist sie es nicht wird die entsprechende Fehlnachricht zurückgegeben. Um sicherzustellen, dass auch die Freundschaftsanfrage wirklich gültig ist, muss zusätzlich gewährleistet werden, dass zwischen den beiden Spielern keine ausstehenden Freundschaftsanfragen existieren (siehe Zeile 7 des Codeausschnitts 4.39).

Ist dies nicht der Fall, kann die Freundschaftsanfrage der Liste unter dem Key `friend_requests:requestName` in der Form `username:userid` hinzugefügt werden und die `userid` des angefragten Freundes wird zurückgegeben, um an ihn die Freundschaftsanfrage zu senden (siehe Kapitel 4.2.3).

```

1 const friendIsValid = async (username, requestName) => {
2   if (requestName === username) {
```

```
3     return {errorMsg: "You can not add yourself as a Friend"};
4   }
5   const friend = await getUserData(requestName);
6   if (!Object.keys(friend).length) {
7     return {errorMsg: "User doesn't exist"};
8   }
9   const currentFriendList = await
    redisClient.lrange(`friends:${username}`, 0, -1);
10  if (currentFriendList.includes([requestName,
    friend.userid].join(':'))) {
11    return {errorMsg: "You are already friends with that user"};
12  }
13  return friend;
14 }
```

Codeausschnitt 4.40: friendIsValid Methode um zu überprüfen, ob eine Freundschaft legal wäre

Bei einer legalen Freundschaft muss gewährleistet sein, dass man nicht mit sich selbst befreundet sein kann (siehe Zeile 2 des Codeausschnitts 4.40), dass der angefragte Benutzer existiert (siehe Ziele 5 & 6) und, dass die beiden Benutzer nicht bereits befreundet sind.

Die Überprüfung ob der Nutzer existiert, basiert darauf, dass falls er nicht existiert `friend` aus einem leeren Objekt `{}` besteht, was bedeutet, dass `Object.keys(friend).length` gleich 0 ist (siehe Zeile 6). Der `!`-Ausdruck konvertiert diesen Wert zunächst in den booleschen Wert `false` und negiert ihn anschließend, sodass der Ausdruck `true` zurückgibt.



# 5 Fazit und Ausblick

## 5.1 Zusammenfassung der Ergebnisse

Diese Bachelorarbeit hatte das Ziel eine Schachanwendung als Basis für Erweiterungen zu entwerfen und zu implementieren. Das Ergebnis ist eine App mit folgenden Funktionen:

- Schachpartien mit unterschiedlichen Zeitkonfigurationen können sowohl gegen zufällige als auch befreundete Gegner gespielt werden.
- Das gleichzeitige Spielen mehrerer Partien ist möglich.
- Während einer Schachpartie ermöglicht ein integrierter Chat die Kommunikation zwischen den Gegnern.
- Zuschauer können Partien über die URL verfolgen.
- Die Erstellung eines Accounts und die Anmeldung sind benutzerfreundlich und sicher gestaltet.
- Das Hinzufügen von Freunden und die Anzeige ihres Online-Status sind implementiert.
- Eine Funktion zum einfachen Wiedereinstieg in aktive Partien erleichtert das gleichzeitige Spielen mehrerer Spiele.

All diese Funktionen bilden die Grundlage für eine Schachanwendung, die besonderen Wert auf soziale Interaktionen legt. Bei der Implementierung lag der Schwerpunkt auf Modularität, Wartbarkeit, Skalierbarkeit, Erweiterbarkeit und Echtzeitkommunikation. Dafür wurden moderne Web-Technologien für eine schnelle und benutzerfreundliche Anwendung verwendet.

## 5.2 Herausforderungen

In diesem Kapitel bespreche ich Herausforderungen auf die ich gestoßen bin während der Konzeption oder Implementierung der Schachanwendung.

### 5.2.1 Schachuhren

Die Konzeption der Schachuhren war eine Herausforderung, die vielen verschiedene Ansätze beinhaltete. Im Endeffekt wurde sich für eine getrennte serverseitige und clientseitige Lösung entschieden (siehe Kapitel 3.3). Doch auch bei dieser Lösung gab es

Gestaltungsspielraum, beispielsweise wie die Zeiten konsistent gehalten werden sollten, wie das Inkrement addiert wird oder wie die verschiedenen Zeiten (Startzeit, reguläre Zeit, von weiß, von schwarz) und Events realisiert werden sollten.

Ein Problem, welches bei der Konzeption und Realisierung nur bedingt gelöst werden konnte ist die Abfrage der Zeit im Backend, wenn die Daten des Schachspiels geholt werden.

Die Frage ist, wie kann man die Zeiten konsistent im Backend speichern kann, um zu jedem Möglichen Zeitpunkt dem Frontend mitteilen zu können welche genauen Zeiten die Spieler noch haben.

Aktuell ist so realisiert, dass das `ServerChessClock` Objekt bei der Initialisierung eines Schachspiels im Backend in einem Array der `socketChessController` Datei gespeichert wird (siehe Kapitel 4.2.2) und bei einer Abfrage des Frontends, das Objekt herausgenommen wird und die Zeiten übermittelt werden.

Das Problem dieser Implementierung besteht darin, dass bei vielen aktiven Partien zahlreiche `ServerChessClock`-Objekte den Arbeitsspeicher des Backends belegen und bei einem Neustart oder Problem des Backends das Array neu initialisiert werden könnte, wodurch alle diese Objekte verloren gehen würden.

Somit verringert diese Implementierung sowohl die Skalierbarkeit, als auch die Wartbarkeit des Backends.

Der Grund, warum dennoch vorerst diese Implementierung gewählt wurde, liegt in den zu diesem Zeitpunkt fehlenden Alternativen. Es wäre denkbar, die aktuellen Zeiten wie die anderen Daten des Schachspiels in Redis zu speichern. Jedoch hätte eine Speicherung der aktuellen Zeit jede Sekunde viel zu viele I/O Operationen zur Folge und beispielsweise nur alle 10 Sekunden die aktuellen Zeiten zu speichern würde zu einer zu großen Diskrepanz führen, wenn das Frontend in einer ungünstigen Zeit die Daten abfragen würde und es sich um ein schnelles Spiel wie „1+0“ handeln würde. Die Speicherung der Zeit nach jedem Zug könnte auch zu sehr großen Diskrepanzen führen, wenn lange kein Zug gemacht wurde.

Eine mögliche Alternative um die Skalierbarkeit und Wartbarkeit hinsichtlich dieser Implementierung zu erhöhen und dennoch keine großen Zeitdiskrepanzen zu haben, wäre eine Lösung, bei der man bei jedem Zug die aktuellen Zeiten der Spieler und den Zeitpunkt des Zugs in Redis speichert. Sobald eine Abfrage der Daten von dem Frontend kommt, können die aktuellen Zeiten der Spieler berechnet werden, indem die Differenz zwischen dem jetzigen Zeitpunkt und dem Zeitpunkt des alten Zugs berechnet wird und diese von der Zeit des Spielers abgezogen wird, der gerade am Zug ist. Eine serverseitige Überprüfung, ob eine Zeit abgelaufen ist, müsste in einem Intervall von jeder Sekunde ausgeführt werden. Allerdings könnte man vermeiden, dass die `ServerChessClock`-Objekte im Arbeitsspeicher des Backends gespeichert werden müssen.

Diese Alternative scheint eine vielversprechende Möglichkeit zur Verbesserung zu sein.

### 5.2.2 Züge des Schachspiels

Bei der Implementierung der Behandlung eines neuen Zugs im Front- und Backend sind kleinere Herausforderungen aufgekommen.

Während des Testens der Anwendung wurde festgestellt, dass chessground die Szenarien von Bauernumwandlungen und En-passant-Schlägen nicht ausreichend unterstützt und die betreffenden Figuren manuell entfernt oder ersetzt werden müssen (siehe Kapitel 3.4.3). Nach etwas Feinarbeit konnten diese Szenarien problemlos behandelt werden. Im Backend trat das Problem auf, dass die Bibliothek chess.js ein ECMAScript-Module ist und Node.js das CommonJS Modulsystem verwendet. Mithilfe eines Umwegs konnte dieses Problem allerdings gelöst werden (siehe Kapitel 4.2.2).

### 5.3 Zukünftige Erweiterungen und Verbesserungen

Neben der in 5.2.1 beschriebenen Verbesserung der Backend Schachuhr und das ausgiebigere strukturierte Testen ist die Anzahl der möglichen Erweiterungen dieser Schachanwendung nahezu unbegrenzt. Einige der wichtigsten Erweiterungen, welche als nächstes angegangen werden könnten wären:

- **Responsive Design:** Die Anwendung für verschiedene Geräte zu optimieren wäre ein wichtiger Schritt Richtung Veröffentlichung.
- **Hosting:** Durch das Bereitstellen der Anwendung auf einer Hostingplattform könnte man die Anwendung der Öffentlichkeit zugänglich machen.
- **Analysen:** Schachpartien analysieren zu können ist wichtig um aus den Fehlern der Partie zu lernen. Es gibt einige Analyseprogramme, wobei die beste derzeit Stockfish ist<sup>1</sup>.
- **Wertungen:** In Schach gibt es eine sogenannte Elo-Wertung, welche die relative Stärke eines Spielers misst<sup>2</sup>. Typischerweise besitzt jeder Spieler jeweils eine Elo für einen Zeitmodi (Bullet, Blitz, Rapid, ...) und Spielende werden aufgrund ihrer Wertung für eine Partie zusammengeführt. Diese Zeitmodi sind schon implementiert, sie werden nur bisher kaum genutzt (siehe Kapitel 4.2.2). Aus dieser Erweiterbarkeit hat sich auch die Funktion entwickelt, dass angemeldete Benutzende nur gegen angemeldete Benutzende spielen können, denn Gast-Spielende besitzen keine Elo-Wertung.
- **Historie vergangener Partien:** Eine Schachpartie wird bereits mit der Historie aller Züge in PGN-Notation gespeichert, allerdings wird diese nach dem Ende des Spiels gelöscht. Man könnte diese Datensätze wiederverwenden, um sich vergangene Spiele nochmal anschauen oder teilen zu können.
- **Zurücknehmen von Zügen:** Das Zurücknehmen eines bereits gemachten Zugs ist eine wichtige Funktion, falls man sich verklickt haben sollte. Meist schickt der Spieler eine Anfrage der Zugzurücknahme, welcher der Gegner annehmen oder ablehnen kann.

<sup>1</sup>Quelle: <https://stockfishchess.org/> am 13. Mai 2023

<sup>2</sup>Quelle: <https://www.chess.com/de/terms/elo> am 13. Mai 2023



Da vor allem die sozialen Interaktionen dieser Schach-App ausgeprägt sein sollen, sind weitere mögliche Erweiterungen die Gründung von Clubs, welche gegeneinander in Ligen oder Turnieren antreten können, einen Clubchat haben, in dem unter anderem Partien geteilt werden können und Rollen wie Präsident, Vize-Präsident,... verteilt werden können.

Um das Spielen zu Belohnen könnte man Figuren-, Bretter- oder Emotodesigns als Belohnungen einführen, welche nicht nur der Spieler selber, sondern auch der Gegner sieht. Zusätzliche Belohnungen könnte man einführen durch täglich wechselnde und statische Herausforderungen.

Dies bietet auch eine Möglichkeit der Kommerzialisierung: Nicht durch Werbung oder freischaltbare Funktionen Einnahmen generieren, sondern durch freischaltbare Designs. Die Benutzer können sich somit keinen Vorteil durch Funktionen wie Analysen oder Tutorials erkaufen, sondern ausschließlich optische Anpassungen.

Die Vision dieser Anwendung ist eine Schach-App, die stärker auf soziale Interaktionen und Anreize zum Spielen setzt als `lichess.org` und weniger kommerzialisiert ist, indem wichtige freischaltbare Funktionen nicht ausschließlich gegen Geld angeboten werden, wie es bei `chess.com` der Fall ist (siehe Kapitel 1.4).

Insgesamt bietet diese Bachelorarbeit eine solide Grundlage für eine Schach-App, die durch soziale Interaktionen und eine Vielzahl von Funktionen den Schachspielern ein umfassendes und ansprechendes Erlebnis bietet.

# Literaturverzeichnis

- [Add21] ADDAMS R.: *SQL - Der Grundkurs für Ausbildung und Praxis*. 2021.
- [che] Chessground.
- [Fou23] FOUNDATION O.: Node.js - an open-source, cross-platform javascript runtime environment., 2023.
- [Gro23] GROUP T. P. G. D.: Postgresql - the world's most advanced open source relational database., 2023.
- [Hah16] HAHN E. M.: *Express in Action - Writing, Builing and Testing Node.js applications*. Manning Publications, 2016.
- [Han] HANSEN A.: Wie ein schachspiel zum wettstreit der systeme wurde.
- [Le21] LE D. Q. H.: *Developing Modern Database Applications with PostgreSQL*. 2021.
- [MP23] META PLATFORMS I.: React – a javascript library for building user interfaces, 2023.
- [Nel16] NELSON J.: *Mastering Redis - take your knowledge of Redis to the next level to build enthralling applications with ease*. 2016.
- [Ove22] OVERFLOW S.: Stack overflow survey 2022, 2022.
- [Pre15] PREDIGER R.: *NODE.JS - Professionell hochperformante Software entwickeln*. Carl Hanser Verlag, 2015.
- [Rob12] ROBBINS J. N.: *Learning Web Design*. O'REILLY, 2012.
- [Sch22] SCHWARZMÜLLER M.: *React Key Concepts*. 2022.
- [Soc23] SOCKET.IO: Socket.io - bidirectional and low-latency communication for every platform., 2023.
- [vdL74] VAN DER LINDE A.: *Geschichte und Litteratur des Schachspiels, Erster Band*. 1874.



# Abbildungsverzeichnis

1.1	Relatives Suchinteresse des Wortes <i>Chess</i> auf Google in den letzten 5 Jahren.	1
1.2	Home-Bildschirm von Chess.com . . . . .	4
1.3	Abonnement Möglichkeiten von Chess.com . . . . .	4
1.4	Der Lichess Desktop Home-Bildschirm . . . . .	5
2.1	Die Zusatzregel <i>en passant</i> . . . . .	7
2.2	Die Zusatzregel <i>Bauernumwandlung</i> . . . . .	8
2.3	Ablauf einer Anfrage an einen Node.js Server . . . . .	9
2.4	Ablauf einer Anfrage an einen Node.js Server mit Express . . . . .	10
2.5	Beispiel der Nutzung von Middlewares . . . . .	11
2.6	Beispiel der Nutzung von Routing . . . . .	12
2.7	Simple Beispiel der Initialisierung einer socket.io Verbindung und das Senden und Empfangen von Events . . . . .	12
2.8	Darstellung eines Raumes <i>myroom</i> mit zwei sockets . . . . .	13
2.9	Beispiel eines verschlüsselten Tokens von JWT . . . . .	20
2.10	Beispiel einer mit Chakra-UI designten React-Komponente . . . . .	20
3.1	Komponentendiagramm der Anwenundung . . . . .	25
3.2	Ordnerstruktur des Frontends . . . . .	27
3.3	Aufbau der React-Komponenten für angemeldete Benutzende . . . . .	28
3.4	Home und Navbar Komponente eines nicht angemeldeten Benutzenden im dunklen Farbschema . . . . .	30
3.5	Home und Navbar Komponente eines angemeldeten Benutzenden im hellen Farbschema . . . . .	31
3.6	Das Modal der Komponente GameRequest in hellem Farbschema . . . . .	31
3.7	Beispiel einer SignUp-Komponente mit bereits verwendet E-Mail Adresse . . . . .	32
3.8	Beispiel einer Login-Komponente mit ungültigen Daten . . . . .	32
3.9	Beispiel einer ChessGame-Komponente . . . . .	33
3.10	Das Modal der Komponente PromotionModal . . . . .	33
3.11	Lade-Bildschirm beim Starten eines Spiels mit den Buttons der <i>Home</i> -Komponente . . . . .	34
3.12	Benachrichtigung, falls kein Spiel unter der roomId gefunden werden konnte . . . . .	35
3.13	Beispiel eines Toasts, falls der Gegner aufgegeben hat . . . . .	37
3.14	Aktivitätsdiagramm eines empfangenen Schachzugs . . . . .	37
3.15	Aktivitätsdiagramm nach ziehen Schachzugs . . . . .	39
3.16	Das Herausfordern und Zuschauen bei Freunden . . . . .	41
3.17	Das Modal der <i>AddFriendModal</i> -Komponente mit Fehlermeldung . . . . .	42

---

3.18 Ordnerstruktur des Backends . . . . .	43
3.19 Sequenzdiagramm des Schachspielstartprozesses mit unbekanntem Gegner	46
3.20 Aktivitätsdiagramm zur Behandlung eines neuen Zugs im Backend . . . .	48
3.21 Aktivitätsdiagramme des Versenden und Akzeptieren von Freundschafts- anfragen . . . . .	50



# Listings

2.1	Beispiel einer Callback Funktion <b>Quelle:</b> [Pre15]	8
2.2	Beispiel zum Beitreten Raums und das senden eines Events in diesen Raum	13
2.3	Beispiel einer React Komponente	14
2.4	Beispiel von verschiedenen Komponenten auf verschiedenen Pfaden Quelle: [Sch22] (abgewandelt)	16
4.1	Die AccountContext.js-Datei	55
4.2	Die Funktion zum Senden der Benutzerdaten an das Backend	56
4.3	Ein Ausschnitt der <i>Login</i> -Komponente mit Formik	57
4.4	Yup Schema für das Anmelden	59
4.5	Die Datei <i>SocketContext.js</i>	60
4.6	Ausschnitt der <i>Views</i> -Komponente	61
4.7	Initialisierung des Schachspiels nach Empfangen der Daten	61
4.8	Die refreshBoard Methode	63
4.9	Die onMove Methode	64
4.10	Die promotion Methode	64
4.11	Ausschnitt der <i>ChessClock</i> -Komponente	65
4.12	Darstellung der Freundesliste in <i>FriendList</i>	67
4.13	Der connected Eventlistener der <i>FriendList</i> -Komponente	68
4.14	Darstellung der Freundesliste in <i>FriendList</i>	68
4.15	Darstellung der <i>Friendlist</i> und <i>ActiveGames</i> Komponenten in der <i>Home</i> -Komponente	69
4.16	Verwendung von einem Thema mit Chakra UI	70
4.17	Ausschnitt aus index.js und die Datei authRouter.js	71
4.18	Die rateLimiter Middleware	72
4.19	Die handleLogin Middleware zum Authentifizieren mit Cookie	72
4.20	Die attemptLogin Middleware zum Anmelden	73
4.21	Middleware zum Abmelden	74
4.22	Ausschnitt der index.js Datei mit Socket.io Server Erstellung und Middleware Zuweisung	75
4.23	authorizeUser Middleware für Socket.io Verbindungen	76
4.24	initializeUser Middleware für Socket.io Verbindungen	76
4.25	Die onDisconnect Methode für Sockets	77
4.26	Die createChessGame Methode zum Initialisieren einer Schachpartie	77
4.27	Die Chess.mjs Datei	78
4.28	newMove Methode die beim Eingang eines neuen Zugs aufgerufen wird	79

4.29	Der Listener des get_game_data Events zum übermitteln des Schachspiels	81
4.30	Konstruktor der ServerChessClock Klasse . . . . .	82
4.31	Die startTimer und stopCurrentGame Methoden der ServerChessClock Klasse . . . . .	83
4.32	Der Listener des send_friend_request Events . . . . .	85
4.33	Der get_friends Listener . . . . .	86
4.34	Initialisierung der PostgreSQL Datenbank und einer Methode für die Anfragen . . . . .	86
4.35	Initialisierung einer Redis-Datenbank Verbindung . . . . .	86
4.36	Methode um einen Spieler in eine Warteschlange hinzuzufügen . . . . .	87
4.37	Methode um Freunde eines Spielers zu erhalten . . . . .	88
4.38	Methode um alle Informationen eines Arrays von Freunden zu bekommen	88
4.39	requestFriend Methode um eine Freundschaftsanfrage in redis zu speichern	89
4.40	friendIsValid Methode um zu überprüfen, ob eine Freundschaft legal wäre	89