Philipps-Universität Marburg

Fachbereich 12 - Mathematik und Informatik

Bachelorarbeit

Webbasierte Multiplayer Schach-App

von
Jasper Paul Fülle

Mai 2023

Betreuer:
Prof. Dr. Thorsten Thormählen

Arbeitsgruppe Grafik und Multimedia Programmierung

Erklärung

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

Marburg, 24. Mai 2023

Jasper Fülle

Kurzzusammenfassung

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

Abstract

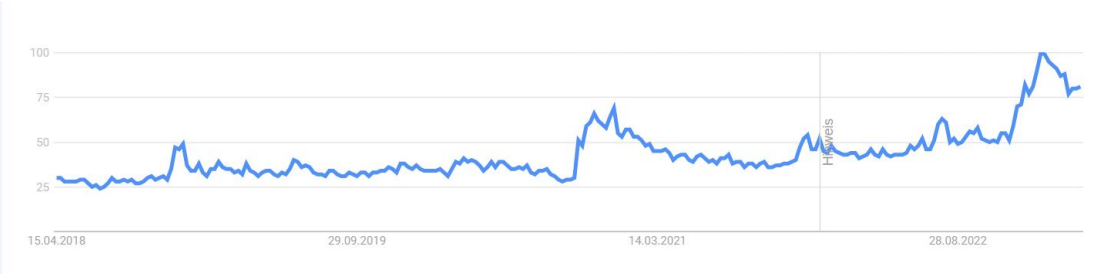
text text text text text text text text text text text text text text text text text text text text text text (exakte englische Übersetzung der deutschen Kurzzusammenfassung)

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Schach ist ein traditionsreiches und abwechslungsreiches Brettspiel, deren Ursprung nicht genau bestimmt werden kann. Es wird vermutet, dass das erste schachähnliche Spiel *Tschaturanga* seinen Ursprung in Nordindien um 600 n. Chr. hatte [vdL74]. Im Laufe der Jahrhunderte hat Schach eine bedeutende Rolle in der Kultur und Geschichte gespielt. So wurde beispielsweise die Schach-WM 1972 eine Art Machtkampf im kalten Krieg zwischen der UdSSR, welche den damaligen Schach dominierten, und der USA[1]. Schach bleibt bis heute ein beliebtes Spiel, welches 2020 durch die Netflix Serie *Damengambit* und 2022 durch den Betrugsvorwurf von Magnus Carlsen an seinen 19-jährigen Gegner Hans Niemann[2] eine breitere Aufmerksamkeit erhielt (siehe Abbildung 1.1).



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

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

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

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

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

1.2 Zielsetzung

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

1.3 Aufbau der Arbeit

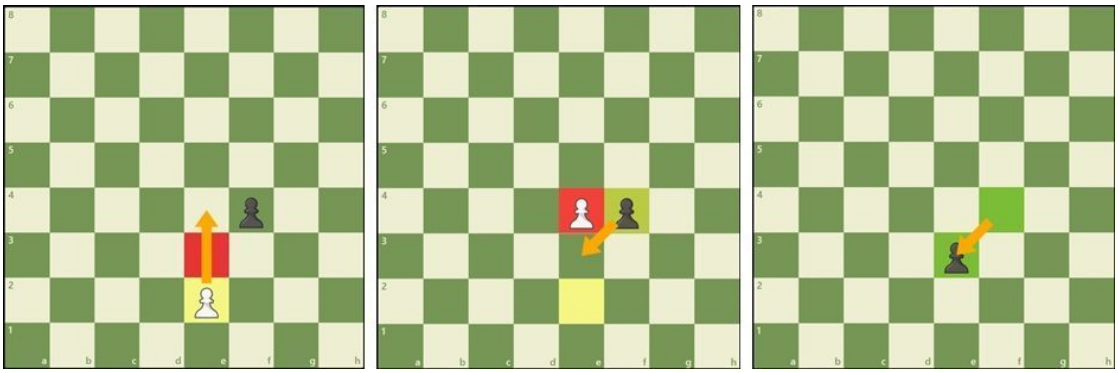
Diese Bachelorarbeit gliedert sich in sechs Hauptkapitel, die jeweils unterschiedliche Aspekte der Entwicklung und Implementierung der Schach-App behandeln.
Im ersten Kapitel, der *Einleitung*, werden die Motivation für die Entwicklung der Schach-App, die Zielsetzung der Arbeit und der Aufbau der Arbeit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

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

2 Theoretische Grundlagen

2.1 Schach

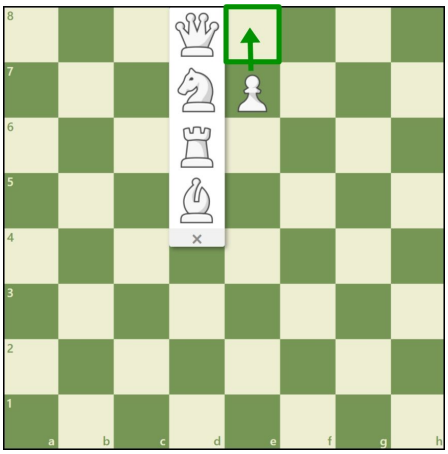
Schach ist ein strategisches Brettspiel für zwei Spieler, welches auf einem quadratischen Spielfeld mit 64 Feldern gespielt wird. Jeder Spieler beginnt mit 16 Figuren und das Ziel des Spiels ist es, den König des Gegners schachmatt zu setzen, indem man ihn bedroht, ohne dass der Gegner den Angriff verhindern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



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

Abbildung 2.1: Die Zusatzregel *en passant*

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



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

Abbildung 2.2: Die Zusatzregel *Bauernumwandlung*

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

2.2 Web-Technologien

2.2.1 Node.js und Express

Node.js und seine Vorteile

Node.js ist eine JavaScript-Laufzeitumgebung, welche erstmals 2009 angekündigt wurde[1] und speziell für die Entwicklung von skalierbaren Netzwerkanwendungen entworfen wurde [Fou23]. Skalierbarkeit bedeutet, dass mit steigender Benutzeranzahl der Ressourcenverbrauch idealerweise linear steigt. Zu den relevanten Ressourcen von Webanwendungen gehören Rechenleistung, Ein-/Ausgabeoperationen (kurz I/O) und Arbeitsspeicher, wobei Node.js vor allem die Skalierbarkeit von I/O intensiven Anwendungen verbessert [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 Code Snippet 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 jedoch erst bei einer hohen Anzahl von Zugriffen erkennbar.

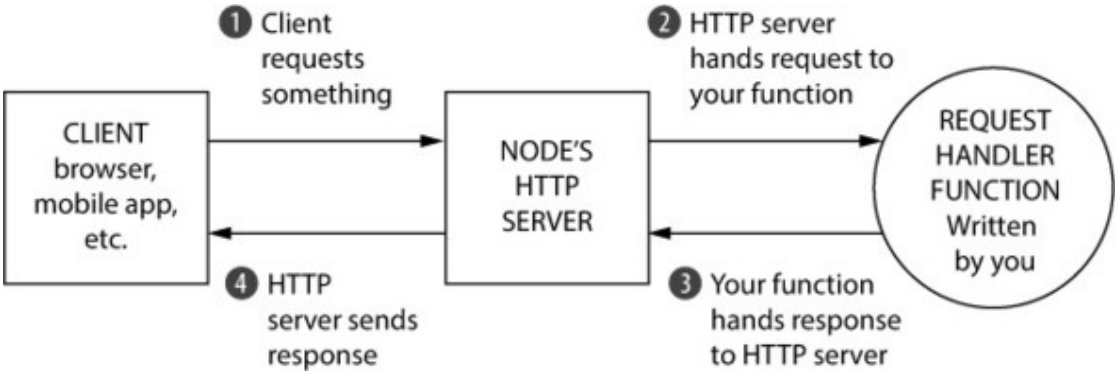
```
1 database.query(  "SELECT * FROM user",  function(result) {
```

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

```
result...
});
```

Code Snippet 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 natürlich besonders hilfreich sein, da Kommunikationsbarrieren durch unterschiedliche Programmiersprachen von Frontend und Backend niedriger sind. Natürlich versteht deshalb der Frontend-Entwickler nicht alles was der Backend-Entwickler macht und umgekehrt, jedoch gibt es eine gemeinsame Grundlage. Neben den Kommunikationsvorteilen ermöglicht die Verwendung von der gleichen Programmiersprache im Frontend und Backend das Teilen von Code. So ist es zum Beispiel möglich Callback Funktionen vom Frontend an das Backend zu senden und dort aufzurufen.
Node.js basiert auf der Verarbeitung von Requests vom Frontend und dem zurück senden von einem Result mittels dem HTTP-Protokoll. Das HTTP-Protokoll verwendet verschiedene Methoden wie GET, POST, PUT und DELETE, um unterschiedliche Aktionen durchzuführen [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.



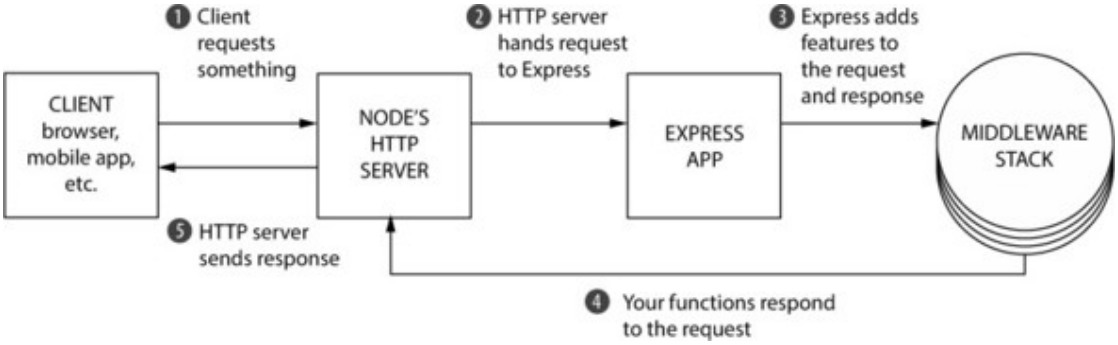
Quelle: [Hah16]

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

Express

Express ist ein leichtgewichtiges und sehr beliebtes Web-Frameworks, welches unter Node.js zur Verfügung steht. Es dient zur Vereinfachung der API von Node.js und stellt hilfreiche Funktionen bereit [Hah16]. Es ermöglicht beispielsweise die Verwendung von *Middleware* und *Routing*.

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



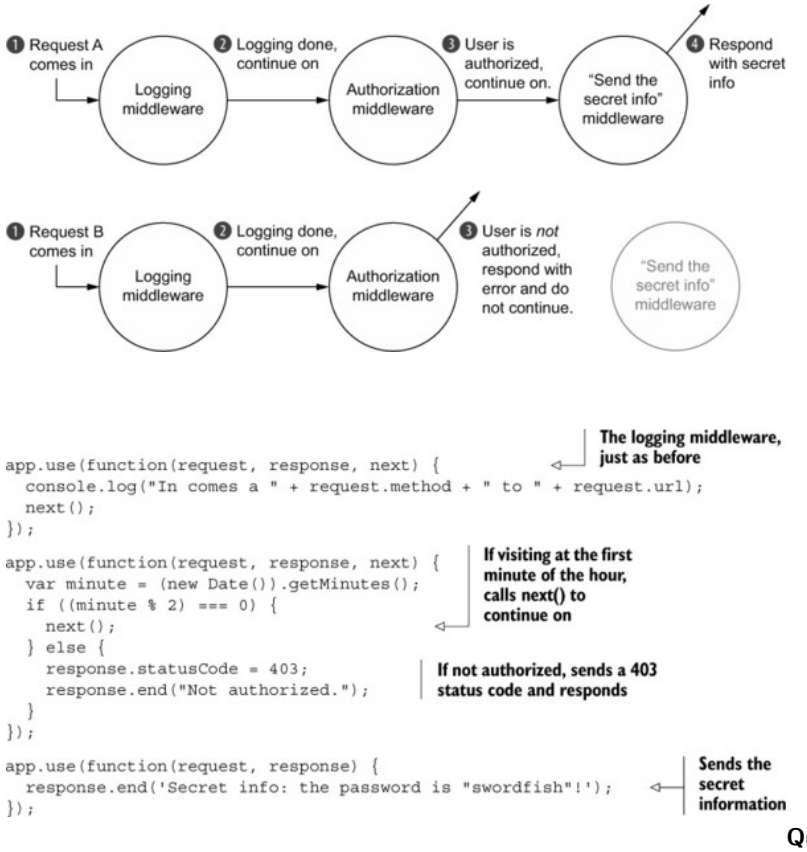
Quelle: [Hah16]

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 die URL `/auth` mit den angegeben Anmeldedaten des Benutzers gesendet werden. Unter diesem Pfad können wir dann bestimmte Middlewares verwenden, welche sich mit der Authentifizierung des Benutzers befassen.

2.2.2 Socket.io

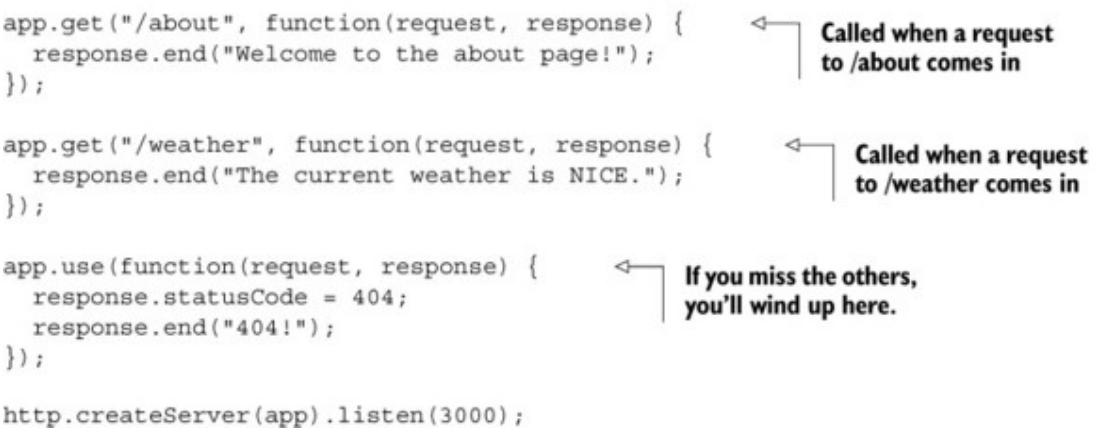
Die Kommunikation mit dem HTTP-Protokoll über Express hat den Nachteil, dass für jeden Datenaustausch eine neue Verbindung aufgebaut und wieder geschlossen wird, was zu einer Latenz führt, welche für Echtzeit-Anwendungen ungeeignet ist. Diese Problematik behebt das Framework *socket.io*.
Es ermöglicht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



Quelle: [Hah16]

Abbildung 2.5: Beispiel der Nutzung von Middlewares

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



Quelle: [Hah16]

Abbildung 2.6: Beispiel der Nutzung von Routing

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

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

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

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

const io = new Server(3000);

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

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

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

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

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

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

Quelle:
[Soc23]

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

```
14 ...
```

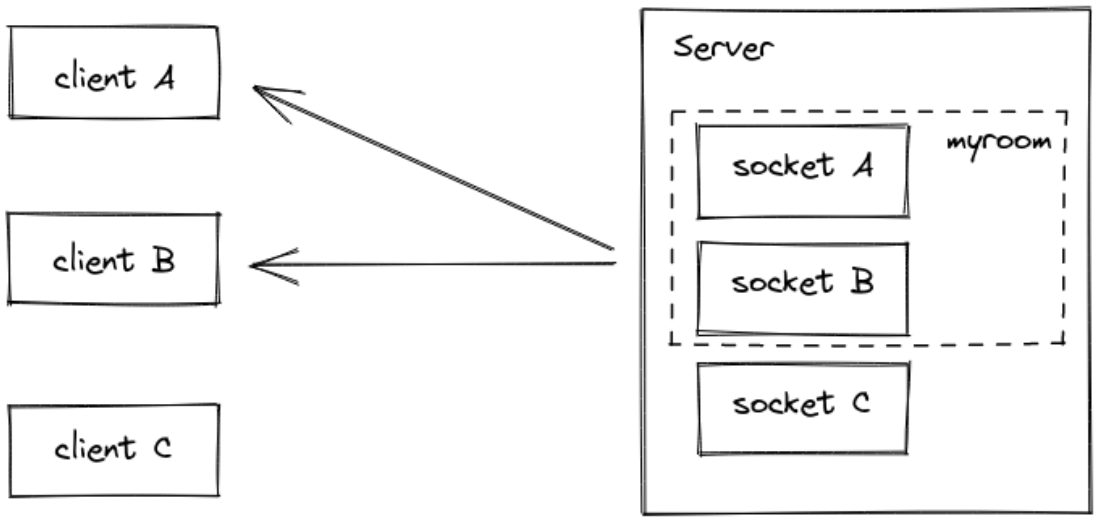
Code Snippet 2.2: Beispiel zum Beitreten Raums und das senden eines Events in diesen Raum

//QUELLE socket.io im nodejsbook

2.2.3 React

React

React ist eine der beliebtesten (nach einer Umfrage von 2022 von Stack Overflow sogar die beliebteste [Ove22]) Frontend Javascript Bibliotheken. Es basiert auf Komponenten, welche wiederverwendbar und kombinierbar sind und vereinfacht die Verwaltung von Interaktionen mit User Interfaces. Dabei benutzt React eine Syntax Erweiterung namens *JSX*. Mit dieser Erweiterung ist es erlaubt HTML Elemente in JavaScript Dateien zu verwenden, welches es ermöglicht die Logik hinter getrennten HTML und JavaScript Dateien in eine Datei zu kombinieren. Die Idee hinter React ist, dass wenn sich nur ein bestimmter Teil des User Interfaces im Vergleich zum aktuell sichtbaren User Interface ändert, auch nur dieser Teil neu gerendert wird und nicht das ganze User Interface. Diese reaktiven Änderungen veranlasst React mittels seiner *Hooks* [Sch22].
Ich werde die Art und Weise wie React und seine Hooks funktionieren an dem Beispiel 2.3 erklären. In diesem Beispiel implementieren wir die Komponente *ExampleComponent*, welche zum Beispiel mittels dem Tag `<ExampleComponent initialCount={10} loadingDelay={3000} />` verwendet werden kann.
Die zwei übergebenen Variablen *initialCount* und *loadingDelay* werden auch **props** genannt, welche in der Komponente verwendet werden können. Eine Komponente ist eine Funktion, welche als Rückgabe den HTML-Code hat, welcher angezeigt werden soll.



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

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

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

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

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

```
        "#333" }}>
36      {isLoading ? (
37        <p>Loading...</p>
38      ) : (
39        <>
40          <p>Count: {count}</p>
41          <button onClick={incrementCount}>Increment
                count</button>
42        </>
43      )}
44    </div>
45  );
46 }
47
48 export default ExampleComponent;
```

Code Snippet 2.3: Beispiel einer React Komponente

React Router

React Router ermöglicht die Erstellung von Anwendung mit mehreren Seiten unter verschiedenen Pfaden [Sch22]. Das ist sinnvoll um als Benutzer direkt einen Pfad angeben zu können um auf die gewünschte Seite zu kommen oder sie zu teilen. Ein Beispiel der Funktion von verschiedenen Komponenten auf verschiedenen Pfaden finden Sie in Abbildung 2.4.
In diesem Beispiel wird unter dem Pfad „/“ die Komponente `Dashboard` angezeigt, während unter dem Pfad „/orders“ die React Komponente `Orders` gerendert wird. Dafür werden die Komponenten `BrowserRouter`, `Routes` und `Route` des Pakets `react-router-dom` benötigt.

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

React Router ermöglicht alledings auch noch ein paar weitere Funktionen, wie zum Beispiel die Hooks `useParams()` und `useNaviagte()` [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 (Beipiel 2.4).

```
import { BrowserRouter, Routes, Route, useNavigate } from
    'react-router-dom';
import { useCallback } from 'react';
import Dashboard from './routes/Dashboard';
import Orders from './routes/Orders';

function App() {
  const navigate = useNavigate();

  const navigateToOrders = useCallback(() => {
    navigate('/orders');
  }, [navigate]);

  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Dashboard />} />
        <Route path="/orders" element={<Orders />} />
        <Route path="/orders/:id" element={ <OrderDetail /> } />
      </Routes>
      <Button onClick={navigateToOrders}> To Orders </Button>
    </BrowserRouter>
  );


export default App;

function OrderDetail() {

  const params = useParams();

  const orderId = params.id;

  useEffect(() => {
  //fetch Data with orderId
  }, [])

  return (
  // Show Data
  );
}

export default OrderDetail;
```

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

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 sogenannte Datenbankmanagementsystemen (DBMS). Das beliebteste DBMS für PostgreSQL ist *pgAdmin*[3]. In relationalen Datenbanken sind Daten in Tabellen organisiert. Zur Bearbeitung und Auswertung von solchen Datenbanken wird die Structured Query Language (**SQL**) verwendet, die in drei Bereiche unterteilt ist [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 bestimme Restriktionen (auch *Constraints* genannt) zugewiesen werden, wie zum Beispiel die UNIQUE Restriktion, welche definiert, dass jeder Wert des Attributs nur einmal in der Tabelle vorkommen darf. Ein weiteres wichtiges Konzept sind Primär- und Fremdschlüssel. Mit Hilfe von ihnen können Tupel verschiedener Tabellen in Beziehung gebracht werden. Ein Primärschlüssel ist ein Attribut, welches jeden Tupel einer Tabelle eindeutig identifiziert und welches als Fremdschlüssel in anderen Tabellen referenziert werden kann. Als Beispiel könnte eine Artikelnummer in einer Tabelle der Artikel als Primärschlüssel genutzt werden, welche in einer Tabelle Rechnung mit verschiedenen abgeschlossenen Bestellungen als Fremdschlüssel referenziert werden kann.
Zudem ist PostgreSQL ACID-konform. **ACID** steht für folgende Fachbegriffe [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*: Während einer Transaktion wird die Datenbank isoliert, da während einer Transaktion ein inkonsistenter Zustand herrschen kann. Diese Isolation wird am Ende der Transaktion aufgehoben.

[3] Quelle: `https://www.pgadmin.org/` am 22. April 2023

- *Durability (Dauerhaftigkeit)*: Nach einer abgeschlossenen Transaktion sind die Änderungen an der Datenbank dauerhaft abgespeichert, sodass beispielsweise ein Systemabsturz die Daten nicht gefährden kann.

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

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, um Datenverluste zu vermeiden. Oft dient Redis als Cache-Speicher, um häufig verwendete Daten temporär zu speichern und dadurch den Zugriff auf die Daten zu beschleunigen[5]. Zu den möglichen Datentypen zählen Strings, Listen, Sets, Hashes, sortierte Sets, Streams und einige weitere[5].
Zudem ermöglicht Redis eine gute Skalierbarkeit indem mehrere Redis Instanzen verbunden werden, anstatt eine Instanz hoch zu skalieren.

2.2.5 Weitere verwendete Bibliotheken

JWT

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

Chakra UI

Chakra UI ist eine simple Komponenten Bibliothek, welche das designen von React Anwendungen vereinfacht[8]. Es stellt Komponenten zur Verfügung, welchen verschiedene

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

Attribute zugeordnet werden können um diese nach eigenem ermessen zu designen (siehe Code Beispiel 2.5 und Abbildung 2.10).

```
import * as React from "react";
import { Box, Center, Image, Flex, Badge, Text } from
    "@chakra-ui/react";
import { MdStar } from "react-icons/md";

export default function Example() {
  return (
    <Center h="100vh">
      <Box p="5" maxW="320px" borderWidth="1px">
        <Image borderRadius="md" src="https://bit.ly/2k1H1t6" />
        <Flex align="baseline" mt={2}>
          <Badge colorScheme="pink">Plus</Badge>
          <Text
            ml={2}
            textTransform="uppercase"
            fontSize="sm"
            fontWeight="bold"
            color="pink.800"
          >
            Verified &bull; Cape Town
          </Text>
        </Flex>
        <Text mt={2} fontSize="xl" fontWeight="semibold"
            lineHeight="short">
          Modern, Chic Penthouse with Mountain, City & Sea Views
        </Text>
        <Text mt={2}>$119/night</Text>
        <Flex mt={2} align="center">
          <Box as={MdStar} color="orange.400" />
          <Text ml={1} fontSize="sm">
            <b>4.84</b> (190)
          </Text>
        </Flex>
      </Box>
    </Center>
  );
}
```

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

Formik und Yup

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

chess.js

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

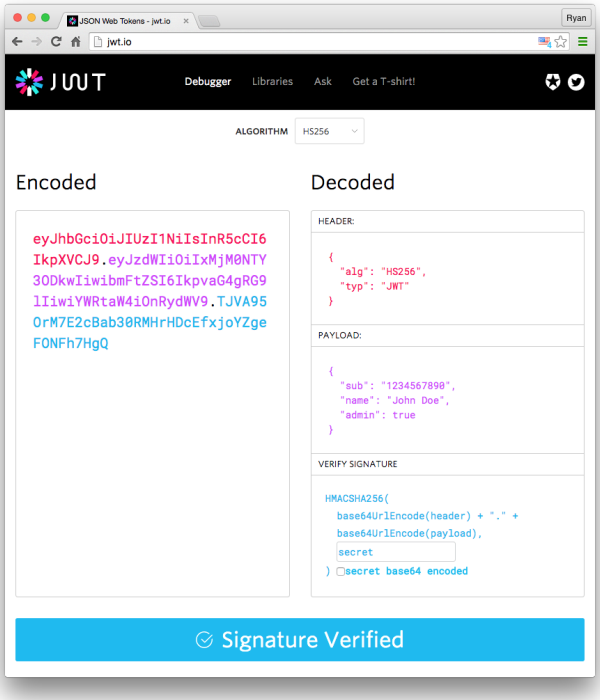
chessground

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

bcrypt

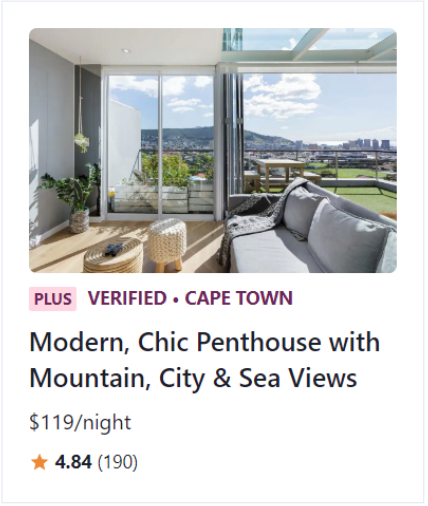
bcrypt ist eine Bibliothek welche entwickelt wurde um Passwörter zu verschlüsseln. Es basiert auf Blowfish, einem Verschlüsselungsalgorithmus, und löst das Problem, dass durch schnellere Hardware Passwörter immer schneller kodiert und dekodiert werden können und dadurch Sicherheitsrisiken entstehen.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

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



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

Abbildung 2.9: Beispiel eines verschlüsselten Tokens von JWT



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

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

3 Systemarchitektur

3.1 Einführung

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

[1] [MP23]
[2] [?]
[3] [Fou23]
[4] [Gro23]
[5] [?]

3.2 Architekturübersicht

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

3.3 Konzeption der Schachuhren

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

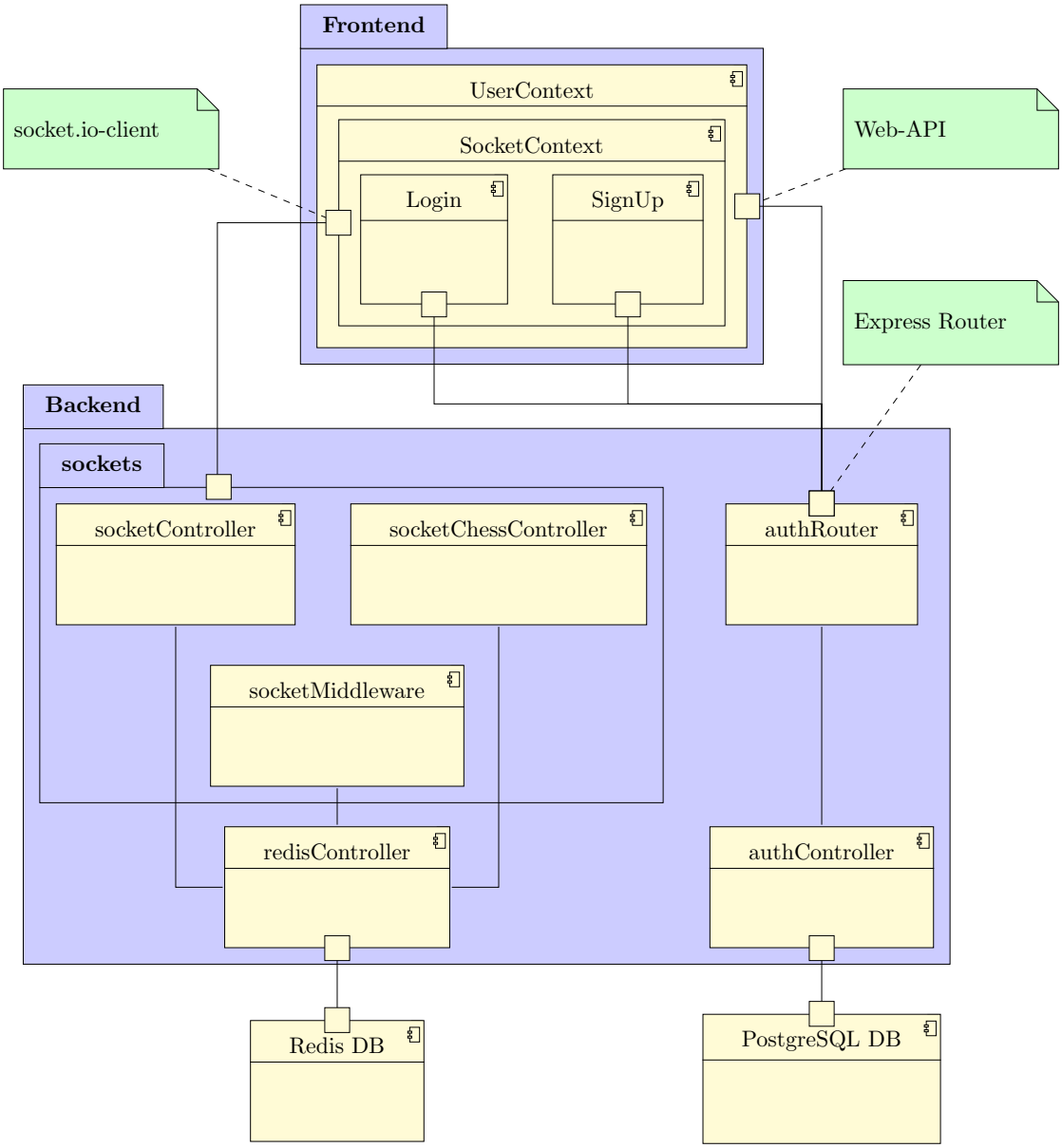


Abbildung 3.1: Komponentendiagramm der Anwenundung

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

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

3.4 Frontend-Architektur

Das Frontend der Anwendung wurde unter Verwendung von React (Abschnitt 2.2.3), Socket.io (Abschnitt 2.2.2) und weiteren Bibliotheken aus Abschnitt 2.2.5 entwickelt. Die Ordnerstruktur (Abbildung 3.2) ist wie folgt aufgebaut:

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

3.4.1 React-Komponenten

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

- **AccountContext:** Stellt Informationen über den Benutzerstatus allen Folgenden Komponenten mittels eines Contexts zur Verfügung. Diese Informationen beinhalten, ob ein Benutzer angemeldet ist und falls er das ist seinen Benutzernamen.
- **SocketContext:** In diesem React Context wird eine socket.io Verbindung mit dem Server hergestellt und allen darauf folgenden Komponenten bereitgestellt.

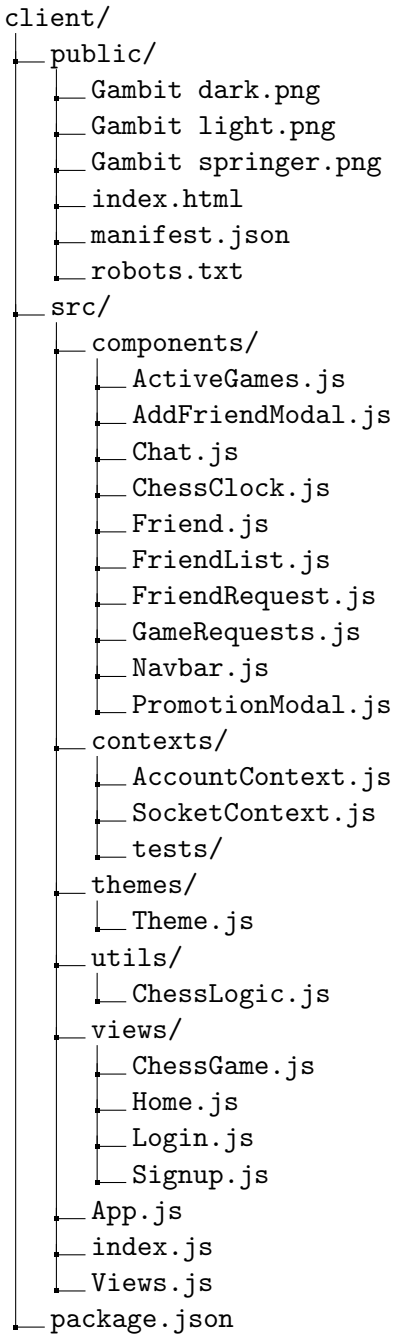


Abbildung 3.2: Ordnerstruktur des Frontends

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

hell) oder das zugreifen auf definierte Stile.

- **Views:** Diese Komponente beinhaltet das User Interface. Mit Hilfe des React-Routers werden hier die Komponenten des `view`-Ordners unter einem bestimmten Pfad definiert. Des weiteren beinhaltet es die Komponenten GameRequest und Navbar, welche durch die Definition in dieser Komponente auf jedem Pfad vorhanden sind.
  - **Navbar:** Die Navigationsleiste besteht aus dem Logo und einem Button zum wechseln des Farbschemas. Je nachdem, ob ein Benutzer angemeldet ist oder nicht beinhaltet es noch Buttons zum Anmelden, Registrieren oder Abmelden (siehe Abbildungen 3.4 & 3.5).
  - **GameRequests:** Diese Komponente ist dafür Verantwortlich beim Eingang einer Spielanfrage eines Freundes, dieses als Modal darzustellen und bietet die Möglichkeit diese Anfrage zu beantworten (siehe Abbildung 3.6).
- **Home:** Diese Komponente stellt die Startseite dar und enthält die Buttons zum Starten eines Spiels mit verschiedenen Schachuhr Konfigurationen (siehe Abbildung 3.4). Ist ein Benutzer angemeldet sind auch noch die Komponenten ActiveGames und FriendList auf der rechten Seite vorhanden (siehe Abbildung 3.5).
  - **ActiveGames:** ActiveGames ist eine Komponente die alle derzeit aktiven Spiele mit den Informationen der Benutzernamen und wer welche Farbe spielt als Buttons darstellt (siehe Abbildung 3.5). Beim klicken auf einen dieser Buttons wird zu der aktiven Partie navigiert.
  - **FriendList:** Diese Komponente verwaltet alle Freunde und Freundschaftsanfragen eines Benutzers, während die Darstellung und Interaktion die Unterkomponenten *Friend* und *FriendRequest* übernehmen. Die Funktionsweise der Komponente mit seinen Unterkomponenten wird in Abschnitt 3.4.3 erläutert.
    - **Friend:** Übernimmt die Darstellung eines Freundes. Mittels eines farbigen Punktes ist erkennbar, ob dieser Freund gerade online ist (grün) oder nicht (rot) (siehe Abbildung 3.5). Ist er online erscheint noch mindestens ein weiterer Button. Es beinhaltet ein Icon in Form von gekreuzten Schwertern und einem Schild. Dies hat die Funktion einen Freund zu einem Spiel herauszufordern. Falls dieser Freund gerade ein aktives Spiel hat erscheint noch ein zweiter Button mit einem Auge als Icon, welches den Benutzer zu dem aktiven Spiel des Freundes als Zuschauer navigiert.//Bilder
    - **FriendRequest:** Eine Freundschaftsanfrage wird mittels dieser Komponente dargestellt und beantwortet (siehe Abbildung 3.5).
    - **AddFriendModal:** Mit Hilfe dieser Komponente können Freundschaftsanfragen unter Angabe des Benutzernamens versendet werden. // BILD

- **Login & SignUp:** Komponenten, die das Anmelden und Registrieren mittels Formularen mit Formik und Yup (siehe Abschnitt 2.2.5) ermöglichen und mit dem Server zur Authentifizierung kommunizieren. //BILD
- **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 (siehe Abschnitt 3.7.1 für eine detaillierte Beschreibung des Ablaufs beim Starten / Suchen eines Spiels) . In der Abbildung 3.7 ist eine beispielhafte Komponente zu sehen. Auf der rechten Seite neben dem Brett befindet sich die *ChessClock* Komponete und daneben befindet sich die *Chat* Komponente. Das Spielfeld und die Figuren entstehen durch die Bibliothek chessground, während die Spiellogik hinter dem Schachspiel von chess.js verwaltet wird (siehe Abschnitt 2.2.5). Eine dataillierte Beschreibung, was in der Komponente beim spielen oder empfangen eines Zuges passiert befindet sich im Abschnitt 3.4.2.
  - **ChessClock:** ChessClock ist eine Komponente, die die Verwaltung und Darstellung der Schachuhren übernimmt.
  - **Chat:** Die Chat Komponente repräsentiert einen simplen gehaltenen Chat in dem die beiden Spieler kommunizieren können. Zuschauer können ihn lesen, allerdings nichts selber schreiben, da sie Tipps geben könnten.

3.4.2 Das Schachspiel

Das Schachspiel und die zugehörigen Schach Uhren sind getrennt gehalten um die Modularität zu erhöhen. Die Schachuhren und das Spiel haben jeweils eigene Events und Listener auf die sie hören.
In diesem Kapitel werde ich näher darauf eingehen wie das Schachspiel im Frontend verwaltet und aktualisiert wird. Die Vorgehensweise im Backend und im Zusammenspiel finden Sie //REFERENZ

Das Spiel

Das Schachspiel findet in der Komponente *ChessGame* statt. Nach dem rendern der Komponente wird das `get_game_data` Event mit der roomId des Spiels und einer Callback Methode gesendet, die die Daten des Spiels beinhaltet, falls dieses existiert. Falls diese Partie nicht im Backend existiert, also keine Daten gesendet werden, wird der Benutzer darauf hingewiesen.
Die Daten die gesendet werden umfassen folgendes:

- Der Namen des weißen und des schwarzen Spielers.
- Welcher Zeitmodus gespielt wird (z.B.: 5 + 3, 10 + 5, ...)
- Die aktuelle Stellung der Partie

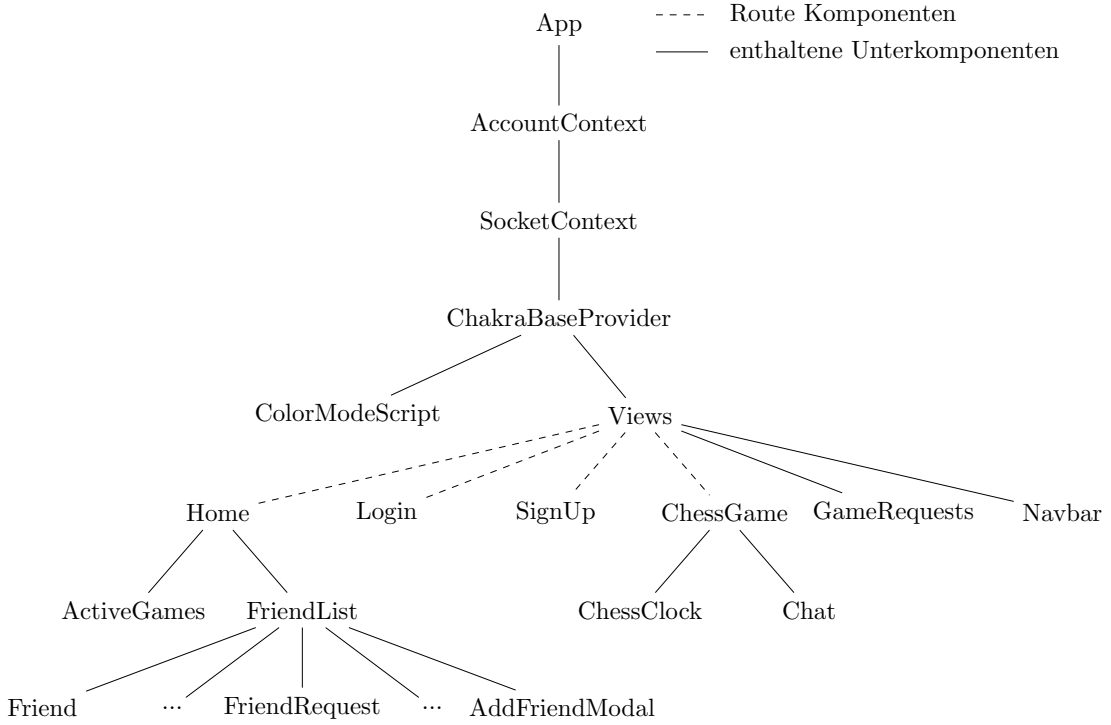


Abbildung 3.3: Aufbau der React-Komponenten für angemeldete Benutzer

- Die bisherigen Nachrichten im Chat.
- Die aktuelle Phase des Spiels: Dabei gibt es die vier Möglichkeiten, dass die Startzeit von Schwarz oder Weiß läuft oder dass die reguläre Zeit von Schwarz oder Weiß läuft.
- Die aktuellen Zeiten der Spieler.

Diese Informationen werden für den Fall benötigt, dass man die Seite neu lädt, über die `ActiveGames`-Komponente auf die Seite gelangt, über einen Link der Partie beitritt oder ähnliches. Das Einholen dieser Daten gewährleistet, dass man auch in diesen Fällen noch ohne Veränderungen weiterspielen kann.
Aufgrund der gesendeten Namen der Spieler wird entschieden, ob man ein Zuschauer oder ein Spieler ist. Dafür werden die Benutzernamen mit dem eigenen Benutzernamen im *UserContext* verglichen. Doch was passiert wenn man eine Partie als unangemeldeter Benutzer spielt und deshalb keinen Benutzernamen im *UserContext* hat?
Bei dem Event `joined_game` (siehe Abschnitt 3.7.1) wird der Gast-Benutzername des Spielers mitgesendet und in `location.state` gesetzt. Dadurch kann in der *ChessGame*-Komponente darauf zugegriffen werden und es kann Überprüft werden, ob es sich um einen Spieler oder Zuschauer handelt.

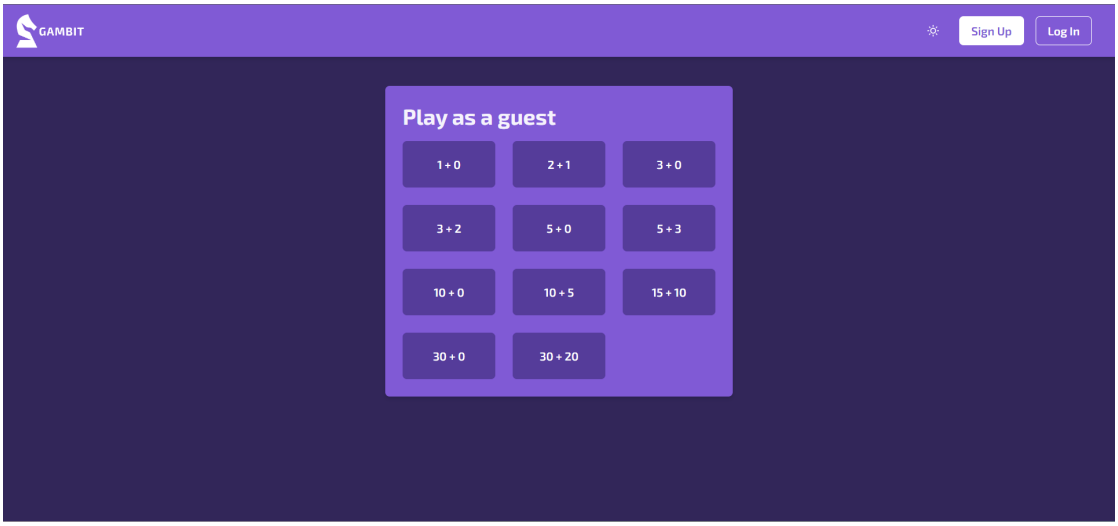


Abbildung 3.4: Home und Navbar Komponente eines nicht angemeldeten Benutzers im dunklen Farbschema

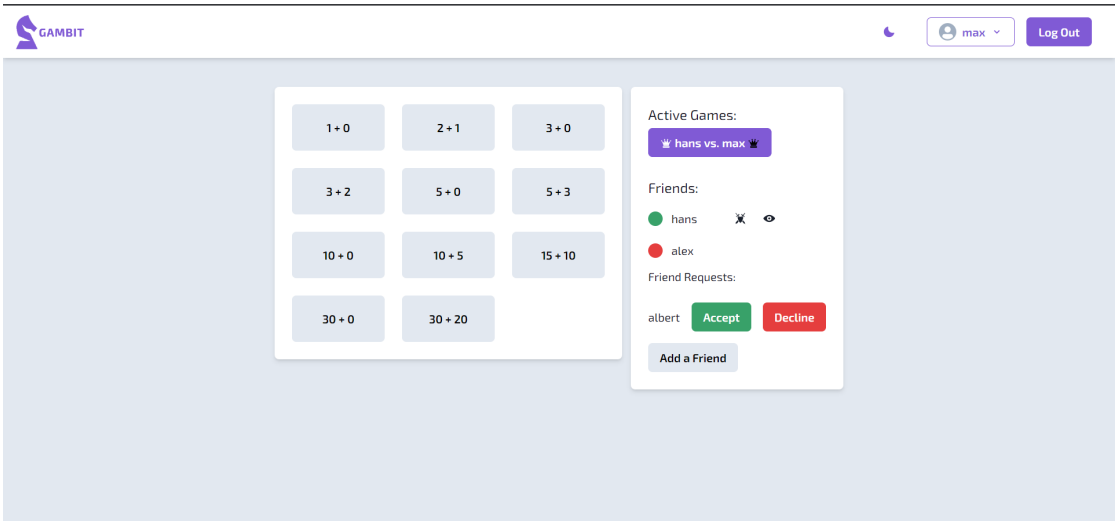


Abbildung 3.5: Home und Navbar Komponente eines angemeldeten Benutzers im hellen Farbschema

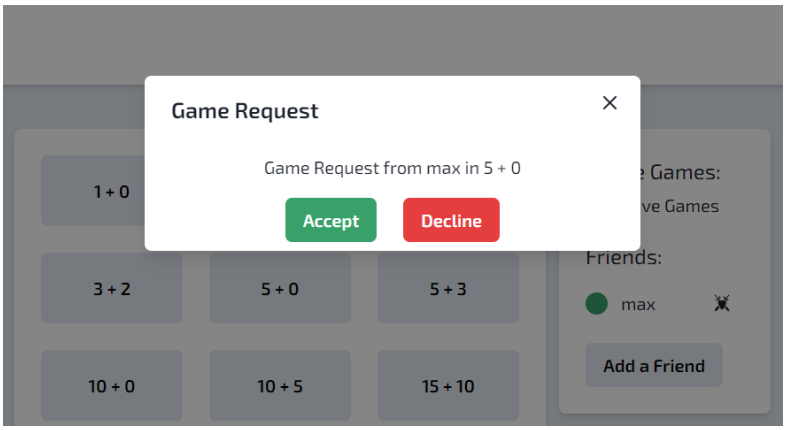


Abbildung 3.6: Das Modal der Komponente GameRequest in hellem Farbschema

Abbildung 3.7: Beispiel einer ChessGame-Komponente

Dem entsprechend wird auch bestimmt wie das Schachbrett, die Namen und die Schachuhren ausgerichtet sind. Ist man Zuschauer wird in chessground definiert, dass man keine Figuren bewegen kann und es gibt kein input Feld für den Chat, sodass man keine Nachrichten abschicken kann. Dies verhindert, dass ein Zuschauer Tipps geben könnte.
Zum Spielen der Partie werden folgende Listener definiert:

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

Die Events `checkmate`, `time_over`, `draw`, `resigned` und `cancel_game` beschreiben alle das Ende des Schachspiels. In ihren Listenern wird definiert, dass man keine Figur des Schach Interfaces von chessground mehr bewegen darf und man wird über den Ausgang des Spiels in Form von einem Toast benachrichtigt. //BILD
Der Ablauf beim Empfangen eines neuen Zugs ist im Aktivitätsdiagramm in Abbildung 3.9 dargestellt. Die Bauernumwandlung und das En Passant müssen separat behandelt werden, da chessground nur das Schach Interface zur Verfügung stellt und bei diesen beiden Zusatzregeln andere Figuren ersetzt oder entfernt werden, als bei regulären Zügen. Das aktualisieren möglicher Züge beinhaltet, dass chessground alle möglichen Züge von chess.js übertragen bekommt, welches zur Folge hat, dass bei einem Klick auf eine Figur korrekt angezeigt wird wohin diese Figur ziehen könnte und auch nur auf diese Felder kann eine Figur dann bewegt werden (siehe Abbildung 3.7).
Gesendet werden können diese Events: `new_move` zum senden eines Zugs, `resign` zum Aufgaben der Partie und `leave_room`, wenn der Spieler die *ChessGame* Komponente verlässt. Ein Aktivitätsdiagramm des Senden eines Zugs befindet sich in Abbildung 3.8. Genau wie bei dem Empfangen eines Zugs wird auch beim Senden zwischen Bauernumwandlung und En Passant unterschieden. Der einzige Unterschied ist, dass bei der Bauernumwandlung nach dem setzen des Zuges noch mittels des *PromotionModal* 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 Partie, die jeweiligen aktuellen Zeiten und die Ausrichtung, welche Zeit oben bzw. unten gezeigt werden soll.
Die Komponente sendet keine Events, sondern hört nur auf die folgenden:

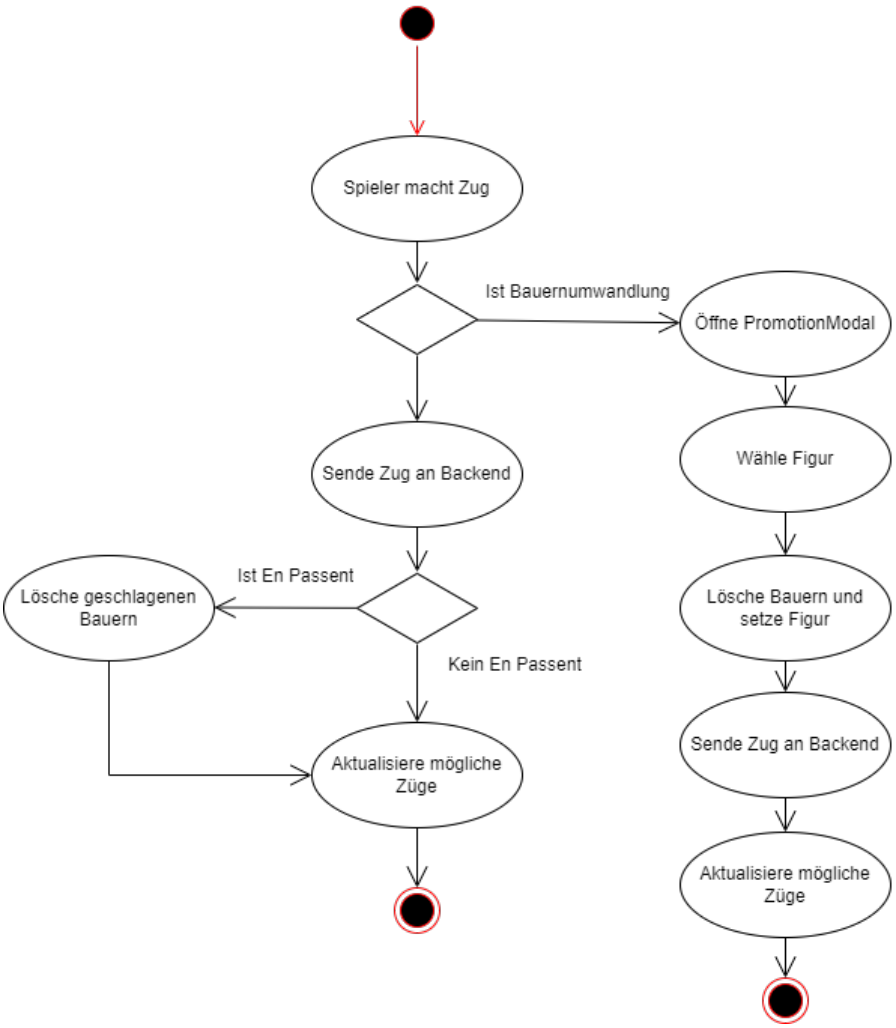


Abbildung 3.8: Aktivitätsdiagramm eines Schach Zugs

- `updated_time`: Dieses Event wird vom Backend gesendet, sobald ein Zug gemacht wurde und enthält die aktuellen Zeiten der Spieler nach dem Zug und welcher Spieler jetzt am Zug ist. Dementsprechend werden die Zeiten aktualisiert und die Uhr des Spielers, welcher jetzt dran ist wird gestartet.
- `stop_starting_time_white`: Stoppt die Start Zeit des weißen Spielers und startet die Start Zeit des schwarzen Spielers.
- `stop_starting_time_black`: Stoppt die Start Zeit des schwarzen Spielers und lässt die reguläre Zeit des weißen Spielers beginnen.
- `stop_clocks`: Wird bei Beendung des Spiels empfangen und stoppt die aktive Uhr.

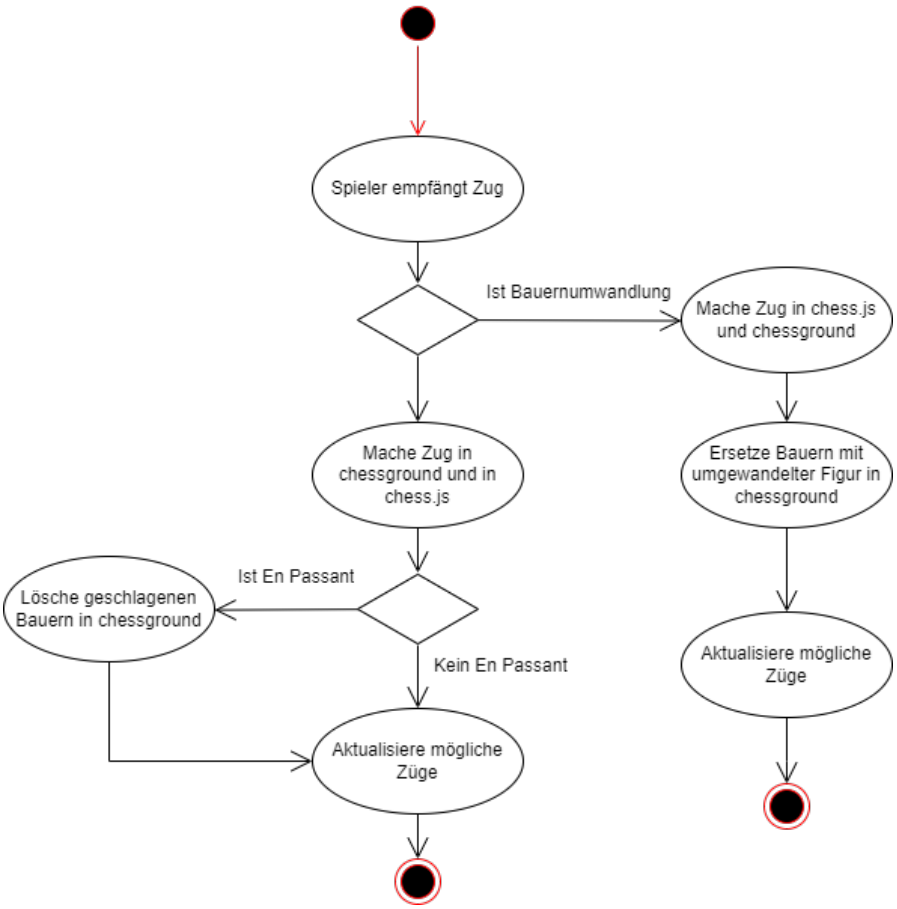


Abbildung 3.9: Aktivitätsdiagramm eines empfangenen Schach Zugs

3.4.3 Verwaltung von Freunden

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

FriendList-Komponente

FriendList verwendet zwei States in Form von Arrays: `friends` und `friendRequests`. Je ein Element dieser Listen wird durch eine *Friend*, beziehungsweise *FriendRequest*, Komponente dargestellt und verwaltet. *FriendList* hört dabei auf die folgenden Events:

- **friends:** Ein Event welches nach der Anmeldung des Nutzers empfangen wird, um die Liste der Freunde zu bekommen.
- **friend_requests:** Genau das gleiche Event, nur zum setzen der Freundschaftsanfragen.

- **friend_request_accepted:** Enthält Daten eines neuen Freundes, welcher deine Freundschaftsanfrage angenommen hat. Dieser wird der Freundesliste hinzugefügt.
- **friend_request:** Eingang einer neuen Freundschaftsanfrage. Wird der Liste der Freundschaftsanfragen hinzugefügt.
- **connected:** Dieses Event wird empfangen, falls ein Freund von dir offline, beziehungsweise online geht. Der betreffende Freundes-Eintrag in der Freundesliste wird aktualisiert.

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

Friend-Komponente

Diese Komponente stellt einen Freund dar. Es kriegt als props alle wichtigen Daten des Freundes gestellt. Sie hat zwei Grundlegende Funktionen: Das Zuschauen einer Partie eines Freundes, das Herausfordern zu einer Partie.
Diese beiden Funktionen sind nur verfügbar, falls der Freund gerade online ist, welches über einen grünen Punkt ersichtlich ist.

FriendRequest-Komponente

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

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 3.10) ist auf dieser Weise aufgebaut:

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

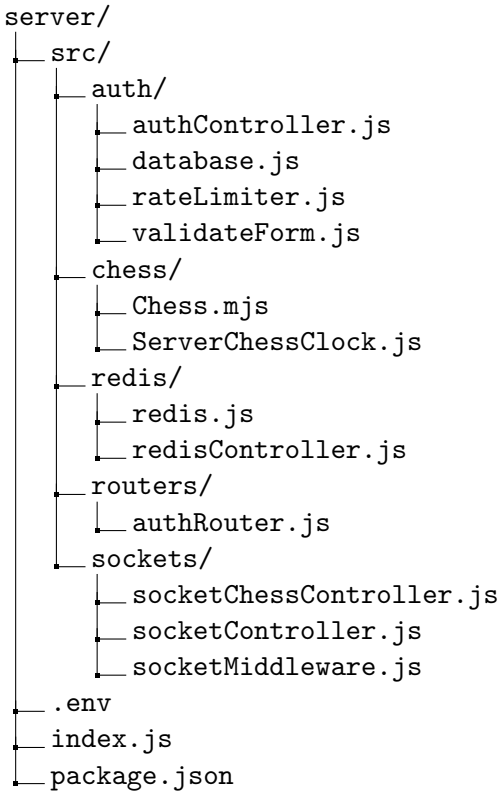


Abbildung 3.10: Ordnerstruktur des Backends

3.5.1 Authentifizierung

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

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

Authentifizierung mit der Web-API und PostgreSQL

Das Anmelden und Registrieren mittels Formular läuft über eine POST Anfrage des Clients an den Pfad `/auth/login`, beziehungsweise `/auth/signup`, die die angegebenen Formulardaten beinhaltet. Bei der Verarbeitung der Anfrage werden mittels des Express Routing verschiedene Middlewares verwendet.
Eine Middleware stellt sicher, dass die Anzahl der Anfragen über eine IP-Adresse in einer bestimmten Zeit begrenzt wird. Dies verhindert sogenannte Denial-of-Service (kurz: DoS) Attacken[6], bei denen probiert wird den Server mit so vielen Anfragen zu belasten, dass dieser außer Betrieb gesetzt wird.
Anschließend überprüft eine Middleware, ob die angegebenen Daten mit dem Schema übereinstimmen.
Treten bei diesen beiden Middlewares keine Fehler auf wird beim Anmelden überprüft, ob dieser Benutzer in der Datenbank existiert und es wird mittels bcrypt überprüft ob die Passwörter übereinstimmen. Ist dies der Fall, wird ein JWT-Token mit den Benutzerinformationen erstellt und als Cookie in den Browser des Clients gesetzt. Des weiteren wird dem Benutzer natürlich geantwortet, dass die Anmeldung erfolgreich war.
Beim Registrieren wird überprüft, ob bereits ein Nutzer mit dem Benutzernamen oder E-Mail existiert und anschließend wird ein neuer Tupel in der PostgreSQL Datenbank erstellt, der Cookie mit dem JWT-Token gesetzt und dem Benutzer geantwortet.
Bei dem ersten Aufruf der Seite vom Client wird eine GET Anfrage an `/auth/login` gestellt. Bei dieser wird ebenfalls die Middleware gegen DoS-Attacken verwendet und anschließend wird überprüft, ob er einen gültigen JWT-Token im Cookie hat und ihm wird dem entsprechend geantwortet. Das setzen des Tokens im Cookie hat den Vorteil, dass beim neuen Aufruf der Seite, solange der Cookie noch gültig ist, der Benutzer automatisch angemeldet ist, ohne seine Anmeldedaten nochmals einzugeben.
Anschließend stellt der Client eine Socket.io Verbindung her.

Authentifizierung und anschließende Middleware mit Socket.io

Bei der Verbindungsherstellung des Clients mit dem Socket.io Server durchläuft die socket verschiedene Middlewares.
Die erste Middleware Authentifiziert die Socket des Benutzer. Sie liest aus dem Cookie, der auch in der socket mitgesendet wird, den JWT-Token, falls dieser existiert. Die Daten die in dem Token kodiert sind werden dann in der Socket als Attribute gesetzt, sodass anschließend immer mittels `socket.user` darauf zugegriffen werden kann.

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

Wenn der Benutzer keinen gültigen JWT-Token hat, werden trotzdem alle Middlewares ohne Fehler durchlaufen. Dies liegt daran, dass bei einem fehlgeschlagenen Middleware-Prozess die Socket.io-Verbindung abgelehnt werden würde. Es soll allerdings auch das Spielen einer Schachpartie als Gast möglich sein.
Als zweite Middleware wird der Benutzer mit Daten versorgt, er tritt dem Raum seiner userid bei und wird als online vermerkt. Dabei wird in Redis `user:username` mit seinen Informationen und `connected true` gesetzt. Anschließend werden alle Freunde des Benutzers aus der Redis Datenbank geholt und an alle ein Event gesendet, das signalisiert, dass der Benutzer online ist. Des weiteren werden auch Informationen bezüglich aktiver Spiele und Freundschaftsanfragen aus Redis abgerufen und mittels entsprechenden Events werden die Informationen über Freunde, aktive Partien und Freundschaftsanfragen an den Benutzer gesendet.
Als letzte Middleware werden alle nötigen Listener sowohl für das Schachspiel, als auch für sonstige Funktionen initialisiert.

3.5.2 Hinzufügen von Freunden

In den Aktivitätsdiagrammen in Abbildung 3.11 ist der Ablauf des Versenden und Akzeptieren einer Freundschaftsanfrage abgebildet. Die verschiedenen Datentypen der Speicherung in Redis befindet sich in Abschnitt 3.6.2.

Versenden einer Freundschaftsanfrage

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

Akzeptieren und Ablehnen einer Freundschaftsanfrage

Beim Akzeptieren einer Freundschaftsanfrage wird wie bei dem Versenden nochmals überprüft, ob diese Freundschaft erlaubt ist. Anschließend wird die Freundschaftsanfrage gelöscht, die beiden Benutzer werden in die entsprechende Freundesliste gesetzt, die restlichen Daten der beiden Spieler werden eingeholt und an beide Benutzer werden alle Informationen, wie ob sie online sind oder ob sie aktive Spiele haben, an beide gesendet. Dies stellt sicher, dass sie im Frontend direkt richtig angezeigt werden können.

Beim Ablehnen einer Freundschaftsanfrage wird einfach nur die Freundschaftsanfrage 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 Benutzers.
- **userid:** Jeder Spieler erhält beim Registrieren seine eigene userid. Diese dient der Kommunikation mit dem Benutzer. Bei einer Verbindung mit Socket.io erhält die socket immer eine andere Id, also wie sendet man ein Event an einen bestimmten Spieler? Diese userid löst das Problem, indem man nach dem anmelden immer dieser id als Raum beitritt. Somit kann immer an diese Id gesendet werden und man stellt sicher, dass der Client das Event empfängt.
- **password:** Das mit bcrypt verschlüsselte Passwort des Benutzers.

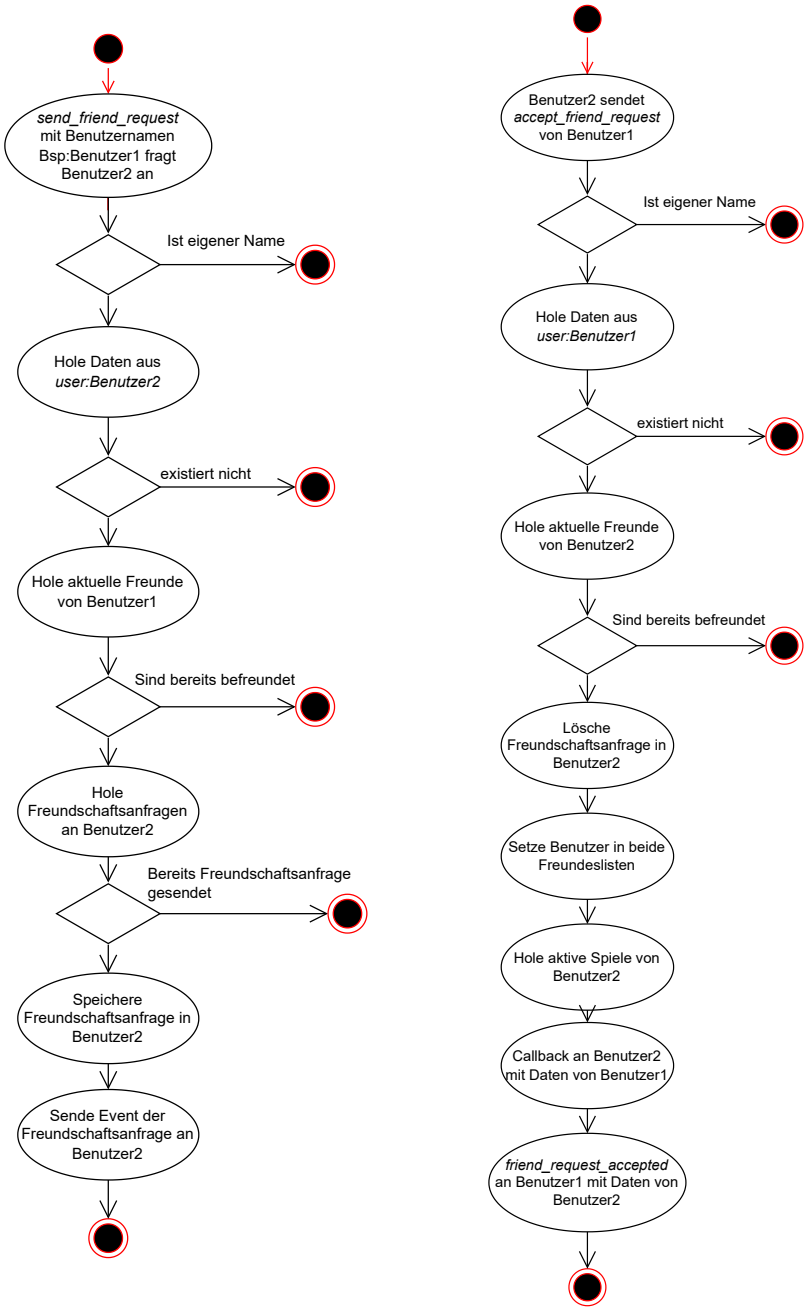
Jedes dieser Attribute, außer das Passwort, hat die Einschränkung, dass es einzigartig sein muss. Die E-Mail wird bisher nicht genutzt, kann aber in Zukunft zum Bestätigen der Registrierung oder Einrichtung eines Newsletters genutzt werden.

3.6.2 Redis

user:username

Unter dem Key `user:username` (wobei hier `username` mit dem entsprechenden Benutzernamen ausgetauscht wird) befindet sich ein Redis Hash. Ein Redis Hash besitzt Key-Value Paare, auf welche man zugreifen kann.
In unserem Fall werden folgende Values zu den Keys dort gespeichert:

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



(a) Senden einer Freundschaftsanfrage

(b) Akzeptieren einer Freundschaftsanfrage

Abbildung 3.11: Aktivitätsdiagramme des Versenden und Akzeptieren von Freundschaftsanfragen

game:roomId

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

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

activeGames:username

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

friends:username

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

friend_requests:username

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

Warteschlangen

Die letzten Elemente sind die Warteschlangen für Spiele. Diese bestehen aus `waitingPlayers:timeMode` und `waitingGuests:timeMode`, je nachdem, ob es sich um einen angemeldeten Benutzer handelt oder nicht. `timeMode` repräsentiert hier alle möglichen Schachuhren, also beispielsweise „10 + 5“, „15 + 10“, ...
Diese Warteschlagen sind Listen, welche aus `username:userid` Einträgen bestehen, um die Anzahl an Abfragen zu verringern.
Durch die Redis Operationen `RPOP` und `LPUSH` lassen sich atomar Einträge hinzufügen oder herausnehmen, welches die Konsistenz der Liste gewährleistet.

3.7 Interaktion zwischen Frontend und Backend

Es gibt einige Funktionen die nur durch komplexere Interaktionen zwischen Frontend, Backend und Datenbank möglich sind. Um diese Interaktionen besser zu verstehen werde ich sie in diesem Kapitel mittels Sequenzdiagrammen erläutern.

3.7.1 Suchen einer Partie mit unbekanntem Gegner

Um eine Schachpartie mit einem zufälligen Gegner zu starten klickt man auf einen der Buttons mit den Zeitkonfigurationen in der *Home*-Komponente (siehe Abbildung 3.4 oder 3.5). Der Ablauf, wie ein Spieler gefunden wird und ein Spiel gestartet wird werde ich hier erklären. In Abbildung 3.12 ist ein Sequenzdiagramm dieses Ablaufs dargestellt.

- Sobald ein Spieler auf einen der Buttons klickt, wird das Event `find_game` mit der Zeitkonfiguration gesendet.
- Falls der Benutzer nicht angemeldet ist, wird ihm ein zufälliger username zugewiesen, der mit „guest-“ startet. Seine userid wird als seine socket id festgelegt.
- Daraufhin wird im Server ein Spieler aus der entsprechenden Warteschlange aus Redis genommen (siehe Abschnitt 3.6.2).
- Falls dabei kein Spieler entnommen werden konnte, wird der Benutzer selbst in die Liste geschrieben und wartet bis er von einem anderen Benutzer aus der Liste genommen wird.
- Falls ein Spieler aus der Liste entnommen werden konnte, wird das Spiel mit einer *roomId* als Identifikator initialisiert und in Redis gespeichert.
- An die beiden Spieler wird das Event `joined_game` mit der roomId 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 Zustands der Partie aus Redis geholt und an das Frontend zurück gesendet. Ein Ablauf was im Frontend bei einer Schachpartie passiert befindet sich im Abschnitt 3.4.2.

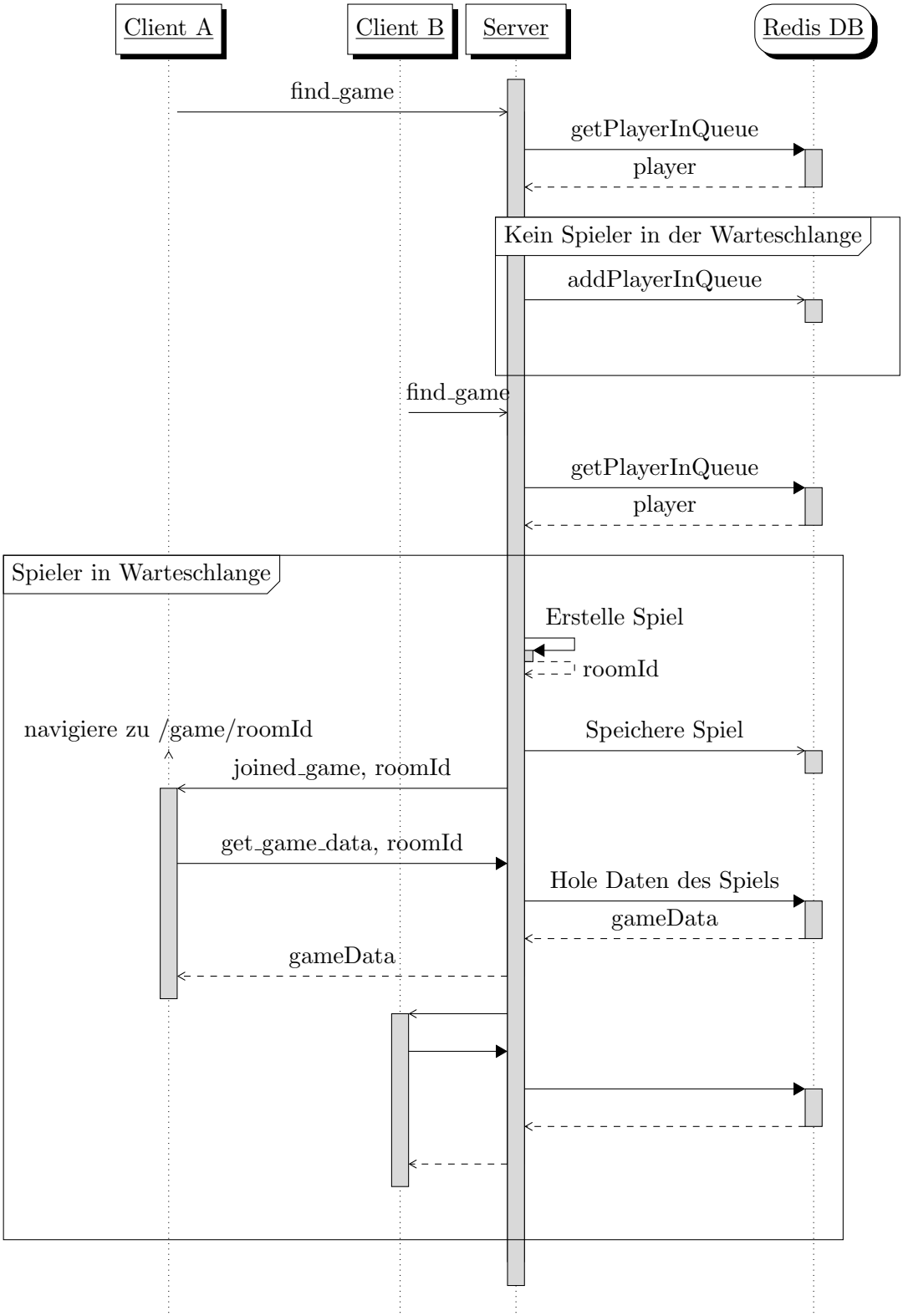


Abbildung 3.12: Sequenzdiagramm des Schachspielstartprozesses mit unbekanntem Gegner

4 Implementierung

4.1 Frontend-Entwicklung

4.2 Backend-Entwicklung

4.3 Datenbank-Integration

5 Fazit und Ausblick

5.1 Zusammenfassung der Ergebnisse

5.2 Limitationen

5.3 Potenzielle Erweiterungen und Weiterentwicklung

Abbildungsverzeichnis

Literaturverzeichnis

[Add21] Addams R.: *SQL - Der Grundkurs für Ausbildung und Praxis.* 2021.

[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.

[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.