Umsetzung MVP einer Multiplayer Schach-Webseite

Knight-Shift: Analyse, Design und Umsetzung einer Multiplayer Schach-Webseite

Timo J. Jokinen

Höhere Fachschule Teko Zürich
Diplomarbeit Fachrichtung Informatik, Applikationsentwicklung
Z-TIN-20-T-a, 2023
Klassifizierung: öffentlich

Vorwort

Schach, ein uraltes Strategiespiel für zwei Spieler, hat seine Wurzeln in Nordindien vor über tausend Jahren. Von dort aus hat es sich über die Seidenstrasse in den Nahen Osten und schliesslich nach Europa ausgebreitet, wo es seine heutige Form annahm. Das Spiel ist für seine taktische Tiefe und komplexe Strategie bekannt, und über die Jahrhunderte hinweg haben sich verschiedene Stile, Theorien und Öffnungen entwickelt, die es zu einem faszinierenden Studienobjekt für Enthusiasten auf der ganzen Welt gemacht haben. Einer der Gründe für seine anhaltende Beliebtheit ist die schiere Zahl der möglichen Spielverläufe. Trotz der begrenzten Anzahl von 64 Feldern und 32 Figuren, von denen jede ihre eigenen Bewegungsregeln hat, ist die Zahl der möglichen Kombinationen nahezu unendlich. Tatsächlich hat der berühmte Mathematiker Claude Shannon geschätzt, dass es mehr mögliche Stellungsvariationen im Schach gibt als Atome im beobachtbaren Universum [1]. Dies zeigt die immensen Möglichkeiten und die Tiefe des Spiels.

Der Autor dieser Arbeit ist Schachenthusiast und regelmässiger Nutzer von Online-Schachplattformen, wo er sich gegen andere Spieler misst. Während den ersten Schuljahren war er sogar Mitglied in einem Schachclub, allerdings nie besonders erfolgreich.

Nichtsdestotrotz ist die Faszination für das Spiel nie erloschen. In den letzten Monaten hat der Autor vermehrt mit dem Gedanken gespielt einen eigenen Schachcomputer zu programmieren, um sich selbst herauszufordern und in ein neues Gebiet einzutauchen. Deshalb hat er sich, ohne grosse Vorkenntnisse zu besitzen, dafür entschieden die Gelegenheit der Diplomarbeit zu nutzen, um in die Welt der Schachprogrammierung einzutauchen. Um den Themenbereich etwas zu erweitern, wurde aus der initialen Idee des Schachcomputers eine Schachwebseite. Den Weg, den er bei der Umsetzung dabei gegangen ist, wird in den nächsten Kapiteln präsentiert.

Inhaltsverzeichnis

1 Management Summary

Im Rahmen seiner Diplomarbeit an der höheren Fachschule «Teko Zürich» hat der Autor Timo Jokinen den MVP einer Mehrspieler-Schach-Website entwickelt. Dieser Bericht gibt detaillierte Einblicke in den gesamten Entwicklungsprozess.

Ein umfassendes Projekt zu entwickeln, das ein breites Spektrum an Software-Engineering-Facetten abdeckt, kann eine herausfordernde Aufgabe sein. Die Entwicklung einer Mehrspieler-Schach-Website bietet eine einzigartige Gelegenheit, verschiedene Bereiche der Softwaretechnik zu erforschen, darunter Architektur und Schnittstellendesign, Webentwicklung und UI, Backend Development, Datenbankdesign, Performance Optimierungen, Security, Networking und Testing.

In den nächsten Kapiteln wird der gesamte Prozess der Gestaltung einer Schach-Website, von Idee bis zum fertigen Produkt, dokumentiert. Dabei werden unter anderem Themen wie die Theorie der Schachprogrammierung, sowie die Implementation der erforschten Konzepte behandelt. Dabei wird ein besonderes Augenmerk auf die Effizienz gelegt. Weiterhin wird ein Abstecher in die Gameentwicklung gemacht, wobei der Autor einen Gameserver von Grund auf umsetzt und anschliessend alles mit einem Benutzerinterface verknüpft. Das Ziel und Resultat dieser Arbeit ist die Umsetzung einer funktionsfähigen Schach-Website mit den nötigen Funktionen.

Die nachfolgenden Kapitel beleuchten die technischen Aspekte des Projekts und bieten tiefe Einblicke in die Herausforderungen und Erkenntnisse, die während der Arbeit gewonnen wurden.

2 Projektinitialisierung

2.1 Systemidee

Der Autor widmet sich der Herausforderung, eine Schachwebseite zu entwickeln, die mit renommierten Plattformen wie lichess.org und chess.com vergleichbar ist. Aufgrund der Komplexität des Unterfangens wird die Entwicklung zunächst auf ein Minimum Viable Product (MVP) beschränkt. Der Schwerpunkt liegt daher auf den essenziellen Funktionen, die erforderlich sind, um eine minimale, aber vollwertige Schachplattform zu realisieren. Obwohl die Integration weiterer Funktionen möglich ist, werden diese im Rahmen der Arbeit nicht als entscheidend für den Erfolg betrachtet.

Die Benutzer können über die Webseite Schach gemäss den offiziellen Regeln spielen. Es wird sowohl die Möglichkeit geboten, gegen einen Computer als auch online gegen andere Benutzer anzutreten. Mithilfe eines integrierten Kontosystems soll den Benutzern ermöglicht werden, Spiele gegen menschliche Kontrahenten zu speichern. Dies eröffnet die Option, zu einem späteren Zeitpunkt auf vergangene Partien zuzugreifen und diese zu analysieren.

Zur Unterstützung dieser Analyse wird die Bewertung der jeweiligen Positionen durch einen leistungsstarken Schachcomputer bereitgestellt, eine Funktion, die auch auf etablierten Schachplattformen zu finden ist. Dieses Instrument erleichtert es den Spielern, Fehler und Unzulänglichkeiten in ihrer Spielweise zu identifizieren, zu verstehen und entsprechende Lernprozesse einzuleiten, um ihre Fähigkeiten kontinuierlich zu verbessern.

2.2 Erfolgskriterien

Auf dem Weg zur Entwicklung der Schachwebseite ist die Festlegung klarer und präziser Erfolgskriterien von entscheidender Bedeutung. Diese Kriterien dienen als Massstab, an dem die Funktionalität, Leistung und Gesamteffektivität der Plattform gemessen werden.

Das Wesen der Abgrenzung dieser Kriterien besteht darin, sicherzustellen, dass das Endprodukt mit den ursprünglichen Zielen übereinstimmt.

Die Erfolgskriterien sind in funktionale und technische Segmente unterteilt. Die funktionalen Kriterien konzentrieren sich auf die Benutzerinteraktion, die Spielmöglichkeiten und die Benutzererfahrung und gewährleisten, dass alle nötigen Funktionen für den Erfolg des Prozesses beinhaltet sind. Die technischen Kriterien hingegen konzentrieren sich auf die Leistung, Skalierbarkeit und Sicherheit der Plattform und gewährleisten, dass sie robust, reaktionsschnell und sicher ist.

Jedes Kriterium ist spezifisch formuliert, messbar, erreichbar und relevant, angelehnt an SMART [2]. Während sich das Projekt entfaltet, werden diese Kriterien als entscheidende Bezugspunkte dienen, die Strategien, Entscheidungen und Implementierungsansätze formen. In der Bewertungsphase erleichtern sie eine strukturierte und objektive Beurteilung der Funktionen, Leistung und der gesamten Benutzererfahrung der Plattform und legen damit den Grundstein für kontinuierliche Verbesserung und Weiterentwicklung.

2.2.1 Funktionale Erfolgskriterien

2.2.1.1 Registrierung

Kriterium	Benutzer können ein Benutzerkonto erstellen.
Messung	Funktionalität ist vorhanden
Relevanz	Erforderlich

2.2.1.2 Login

Kriterium	Benutzer können sich mit ihrem Konto bei der Webseite anmelden.
Messung	Funktionalität ist vorhanden
Relevanz	Erforderlich

2.2.1.3 Online-Spiel

Kriterium	Benutzer können mit oder ohne Anmeldung Schach gegen andere Benutzer spielen.
Messung	Funktionalität ist vorhanden
Relevanz	Erforderlich

2.2.1.4 Herausforderung

Kriterium	Benutzer können eine Herausforderung erstellen und andere Benutzer können diese annehmen.
Messung	Funktionalität ist vorhanden
Relevanz	Erforderlich

2.2.1.5 Matchmaking

Kriterium	Benutzer können nach sich in die Matchmaking-Warteschlaufe begeben und werden automatisch mit einem passenden Kontrahenten gepaart.
Messung	Funktionalität ist vorhanden
Relevanz	Optional

2.2.1.6 Schachcomputer-Spiel

Kriterium	Benutzer können gegen einen starken Schachcomputer spielen. (Angedacht ist Stockfish 15)
Messung	Funktionalität ist vorhanden
Relevanz	Erforderlich

2.2.1.7 Spieloptionen

Kriterium	Es stehen verschiedene Zeitformate zur Verfügung in Form von Bedenkzeit pro Spieler und ein optionales Inkrement (Zeit, die nach jedem Zug zur Bedenkzeit dazugerechnet wird).
Messung	Mindestens folgende Zeitformate stehen zur Verfügung: - 3 Minuten - 3 Minuten + 2 Sekunden Inkrement - 5 Minuten - 5 Minuten + 3 Sekunden Inkrement - 10 Minuten
Relevanz	Erforderlich

2.2.1.8 Benutzerprofil

Kriterium	Benutzer sehen eine Liste der vergangenen Spiele in ihrem Profil und können diese Zug für Zug noch einmal durchspielen.
Messung	Funktionalität ist vorhanden
Relevanz	Erforderlich

2.2.1.9 Analyse

Kriterium	Für vergangene Spiele wird Benutzern eine Bewertungsleiste angezeigt, die von einem Schachcomputer angetrieben wird, um die Positionsstärke bzw. den Vorteil für Weiss oder Schwarz zu veranschaulichen.
Messung	Funktionalität ist vorhanden
Relevanz	Erforderlich

2.2.2 Technische Erfolgskriterien

2.2.2.1 Leistungsfähigkeit

Kriterium	Die Plattform kann mehrere Spiele gleichzeitig abwickeln.
Messung	Es wird ein Last-Test durchgeführt, wobei die Plattform 500 Spiele gleichzeitig stemmen soll.
Relevanz	Erforderlich

2.2.2.2 Skalierbarkeit

Kriterium	Die Plattform soll horizontal skalierbar sein (bedeutet mehrere Instanzen der Applikation können nebeneinander laufen).
Messung	Plattform ist so aufgebaut, dass weitere Instanzen dazugeschaltet werden könnten.
Relevanz	Erforderlich

2.2.2.3 Echtzeit

Kriterium	Spiele auf der Plattform zwischen zwei menschlichen Benutzern laufen in Echtzeit.
Messung	Züge von Kontrahenten werden sofort angezeigt, ohne dass die Seite aktualisiert werden muss.
Relevanz	Erforderlich

2.2.2.4 Legale Züge errechnen

Kriterium	Die Ausrechnung der legalen Züge wird programmiert, anstatt eine Bibliothek zu verwenden und integriert fortgeschrittene Techniken aus der Schachprogrammierung.
Messung	- Funktionalität ist vorhanden. - Code ist selbst geschrieben und verwendet Schachprogrammierung Theorie. - Performance wird evaluiert und die legalen Züge werden für eine Position in unter 10 Millisekunden generiert.
Relevanz	Erforderlich

2.2.2.5 Stockfish

Kriterium	Als Schachcomputer und für die Evaluation von Positionen wird Stockfish eingesetzt.
Messung	Stockfish wird integriert.
Relevanz	Erforderlich

2.2.2.6 Automatisierte Tests

Kriterium	Kritische Stellen der Applikation werden mittels automatisierten Tests getestet.
Messung	Automatisierte Tests sind vorhanden.
Relevanz	Erforderlich

2.2.2.7 Sicherheit

Kriterium	Die Methode der Authentifizierung entspricht heutigen Standards.
Messung	Die Authentifizierung folgt einer etablierten Methode.
Relevanz	Erforderlich

2.3 Einschränkungen und Abgrenzungen

Die zu entwickelnde Plattform ist im Rahmen eines Minimum Viable Products (MVP) mit bestimmten Einschränkungen und Abgrenzungen verbunden.

2.3.1 Desktop-Optimierung

In der ersten Entwicklungsphase wird die Plattform spezifisch für Desktop-Geräte optimiert. Anpassung und Optimierung für mobile Endgeräte ist in dieser Phase nicht vorgesehen.

2.3.2 Benutzeroberfläche und Benutzererfahrung

Die Investition in die Benutzeroberfläche (UI), Benutzererfahrung (UX) und Barrierefreiheit (Accessibility) wird begrenzt sein. Diese Aspekte werden nach dem Best-Effort-Prinzip des Autors behandelt, wobei ein Fokus auf Funktionalität über die Ästhetik gestellt wird.

2.3.3 Speicherung von Benutzerdaten

Die Plattform wird Benutzerdaten nur in dem Umfang speichern, wie es für die Umsetzung spezifischer Funktionen unbedingt erforderlich ist. Eine umfangreiche Datensammlung und -speicherung ist nicht vorgesehen.

2.3.4 Datenschutz

Die strikte Einhaltung von nationalen oder internationalen Datenschutzrichtlinien ist in der MVP-Phase nicht prioritär. Dies ist auf den Umstand zurückzuführen, dass das Produkt in dieser Phase nicht für die Öffentlichkeit bestimmt ist. Datenschutzbelange werden womöglich in späteren Entwicklungsphasen (unabhängig von vorliegender Arbeit), insbesondere bei der Erweiterung der Nutzerbasis und der Veröffentlichung für ein breites Publikum, intensiver adressiert werden.

2.4 Allgemeines Projektziel und Auftraggeber

Neben den messbaren Erfolgskriterien fungiert diese Arbeit als Lernprozess für den Autor. Sie wird nicht als Produkt für einen Kunden oder Auftraggeber realisiert, sondern verfolgt das primäre Ziel, die Kompetenzen des Autors zu präsentieren und weiterzuentwickeln. Dabei umfasst das Projekt eine Vielzahl von Informatikthemen, in denen der Autor seine Expertise im Laufe der Arbeit ausbauen möchte. Die Arbeit soll zukünftig bei der Jobsuche im Resümee angehängt werden können. Da es hauptsächlich um den Lerneffekt geht, ist auch die Wirtschaftlichkeit dieser Arbeit nicht sehr relevant.

In der Zukunft besteht die Möglichkeit, die fertige Schachplattform für Schachbegeisterte- und Spieler, die sich online gegeneinander messen möchten, zu veröffentlichen. Zu diesem Zeitpunkt müssen Themen wie Wirtschaftlichkeit behandelt werden. Das ist allerdings nicht Teil dieser Arbeit.

2.5 Projektrisiken

Um Überraschungen zu vermeiden, werden im folgenden Abschnitt Risiken, die den Projekterfolg gefährden definiert und wie damit umgegangen wird:

Als Hauptrisiko erkennt der Autor, dass das Projekt sehr umfangreich ist und möglicherweise mehr Arbeit bedeutet, als initial erwartet. Dieses Risiko geht der Autor ein und handhabt es, indem früh genug gestartet wird, sodass Zeit verbleibt darauf zu reagieren. Zudem wird regelmässig mit dem Zeitplan validiert, wo der Autor steht. Bei starken Verzögerungen wird der Diplomcoach benachrichtigt.

Weiterhin besteht das Risiko, dass technische Limitationen auftreten oder Probleme bei Drittanbietern oder externen Schnittstellen, für welche der Autor keine Lösung finden kann. In diesen Fällen wird klar dokumentiert, was die Limitation ausgelöst hat und wie weiter vorgegangen wird. Ausserdem wird der Diplomcoach sofort benachrichtigt.

Hardwareprobleme können immer unvorhergesehen auftreten und deshalb wird für alle Artefakte (Dokumentation, Programmcode) bei jeder Änderung ein Backup erstellt und online gespeichert. Für Programmcode ist das die Versionskontrolle und für die Dokumentation Microsoft OneDrive.

2.6 Methodik / Vorgehen

Im folgenden Abschnitt wird auf das grobe Vorgehen innerhalb des Projekts eingegangen. Softwareprojekte sind oft nicht sehr gut geeignet für die Wasserfallmethode, nach welcher das Projekt zu Beginn bis ins kleinste Detail durchgeplant wird. Softwareprojekte haben oft die Eigenschaft zu Beginn einige Unklarheiten zu haben und während des Projektverlaufs Änderungen mit sich zu bringen, sei es aufgrund von technischen Schwierigkeiten oder Änderungen am Umfang oder an der Richtung.

Die Dynamik der Technologie und der Benutzeranforderungen bedingt, dass Veränderungen und Anpassungen an der Software unausweichlich sind. Aus diesem Grund lässt sich der Autor vom Grundsatz «Reagieren auf Veränderung über Befolgen eines Plans» aus dem Agile-Manifest [3] inspirieren und entscheidet sich bei der Entwicklung der Schachwebseite für eine Methode, die auf agilen Ansätzen basiert. Eine solche Art von Projektmethodik hat zum Vorteil, dass auf unvorhergesehene Umstände schnell und flexibel reagiert werden kann. Folgend wird erklärt, wie der Autor das Vorgehen zur Umsetzung des Projekts gestaltet und nach welchem Leitsatz gearbeitet wird. Da das Entwicklungsteam aus einer einzigen Person besteht, macht es keinen Sinn ein Framework, wie beispielsweise SCRUM anzuwenden. Stattdessen wird das Projekt in verschiedene Phasen unterteilt, die aufeinander aufbauen.

In der ersten Phase, der Analyse, widmet sich der Autor einer tiefgehenden Untersuchung des geplanten Projekts. Das grobe Gesamtkonzept der Applikation wird hierbei skizziert. Zunächst werden die ursprünglich definierten Ziele und Erfolgskriterien nicht nur verfeinert, sondern auch in detaillierte User Storys transformiert. Diese User Storys spielen eine entscheidende Rolle, da sie als Leitfaden für die Bestimmung der Anforderungen an die Architektur der Plattform dienen. Diese Anforderungen werden dann sorgfältig analysiert, geplant und sowohl grafisch als auch textuell dokumentiert. Ein weiteres Kernelement der Analysephase ist das Erkennen von Abhängigkeiten zwischen einzelnen Komponenten. Diese identifizierten Abhängigkeiten bilden das Fundament für die strategische Ausrichtung der nachfolgenden Projektschritte. Das primäre Ziel dieser Phase ist die Entwicklung eines strukturierten und umsetzbaren Plans für die Implementationsphase.

Die Implementationsphase bildet den praktischen Kern des Projekts. Hier werden die in der Analysephase erarbeiteten Konzepte in die Tat umgesetzt und zu einem funktionierenden Produkt zusammengefügt. Während dieser Umsetzung können unvorhergesehene Herausforderungen und Fragestellungen auftreten. Es ist daher essenziell, flexibel und adaptiv vorzugehen, um inkrementell

auf solche Überraschungen zu reagieren und das ursprünglich gesetzte Projektziel nicht aus den Augen zu verlieren. In der Implementationsphase werden Softwarekomponenten den Abhängigkeiten nach angegangen. Zu jedem Zeitpunkt während der Implementation kann allerdings ein Schritt zurück gemacht werden, wenn etwas nicht aufgeht.

Abschliessend wird eine Evaluationsphase durchgeführt, die dazu dient, die definierten User Storys anhand ihrer Akzeptanzkriterien und schlussendlich auch die Erfolgskriterien zu überprüfen. Falls dabei Fehler aufkommen, werden diese aufgenommen und behandelt.

2.7 Projektname

Als Name für das Projekt hat der Autor *KnightShift* gewählt, ein Wortspiel aus dem englischen Wort für Nachtschicht und der englischen Bezeichnung für die Springer Figur *Knight*. *Shift* bedeutet zudem eine Bewegung von einem Platz zu einem anderen, speziell über kurze Distanzen, was für Schachfiguren passend erschien.

2.8 Hilfsmittel

Der Autor verwendet ChatGPT oder andere LLMs, um Themen zu recherchieren und besser zu verstehen. Werkzeuge wie ChatGPT können komplexe Themen einfach und verständlich erklären und sind somit für diese Art von Hilfe gut geeignet. Weiterhin werden diese Werkzeuge für Mittel der Inspiration und Kreativität verwendet wie beispielsweise das Erarbeiten und Verfeinern von Ideen. Der aufgeführte Programmcode in dieser Arbeit ist jedoch ausschliesslich selbständig geschrieben und es werden keine autogenerierten Inhalte verwendet.

2.9 Zeitplan

2.9.1 Zeitplan Soll

Task	KW 28	KW 29	KW 30	KW 31	KW 32	KW 33	KW 34	KW 35	KW 36	KW 37	KW 38	KW 39	KW 40	KW 41	KW 42	KW 43
Projektinitialisierung																
Aufsetzen Dokumentation mit PanDoc																
Management Summary																
Einführung, Ausgangslage, Problemstellung																
Systemidee / Zielsetzung																
Abgrenzungen																
Vorgehensweise / Methodik definieren																
Research und Design																
Requirement Analyse																
Konkurrenz Analyse																
Konkretisierung Ziele in User Storys																
Informationsarchitektur																
Grobe Gesamtarchitektur																
Definition Technologiestack																
Wireframes																
Datenbank Design / Schema																
Security																
Vorbereitung Umsetzung																
Aufsetzen Code Basis / Repository																
Lokale Entwicklungsumgebung																
Legal Move Generator Umsetzung																
Initiale Research in Schachprogrammierung																
Konkretisierung Anforderungen / Design anhand Research																
Umsetzung und kontinuierliche Research																
Testing / Unit Tests																
Benchmarks																
Dokumentation																
Game Server																
Initiale Research & Design																
Konkretisierung Anforderungen anhand Research																
Umsetzung und kontinuierliche Research																
Testing																
Dokumentation																
User Interface																
Initiale Research & Design																
Planung anhand Research																
Umsetzung und kontinuierliche Research																
Testing																
Dokumentation																
Abschluss																
Review Ziele																
Anforderungsvalidation																
Fazit, Abschluss, Anhang																

2.9.2 Zeitplan Ist

Task	KW 28	KW 29	KW 30	KW 31	KW 32	KW 33	KW 34	KW 35	KW 36	KW 37	KW 38	KW 39	KW 40	KW 41	KW 42	KW 43
Projektinitialisierung																
Aufsetzen Dokumentation mit PanDoc																
Management Summary																
Einführung, Ausgangslage, Problemstellung																
Systemidee / Zielsetzung																
Abgrenzungen																
Vorgehensweise / Methodik definieren																
Research und Design																
Requirement Analyse																
Konkurrenz Analyse																
Konkretisierung Ziele in User Storys																
Informationsarchitektur																
Grobe Gesamtarchitektur																
Definition Technologiestack																
Wireframes																
Datenbank Design / Schema																
Security																
Vorbereitung Umsetzung																
Aufsetzen Code Basis / Repository																
Lokale Entwicklungsumgebung																
Legal Move Generator Umsetzung																
Initiale Research in Schachprogrammierung																
Konkretisierung Anforderungen / Design anhand Research																
Umsetzung und kontinuierliche Research																
Testing / Unit Tests																
Benchmarks																
Dokumentation																
Game Server																
Initiale Research & Design																
Konkretisierung Anforderungen anhand Research																
Umsetzung und kontinuierliche Research																
Testing																
Dokumentation																
User Interface																
Initiale Research & Design																
Planung anhand Research																
Umsetzung und kontinuierliche Research																
Testing																
Dokumentation																
Abschluss																
Review Ziele																
Anforderungsvalidation																
Fazit, Abschluss, Anhang																

3 Analyse

Im nachfolgenden wird eine detaillierte Sammlung von Informationen erarbeitet, die es ermöglicht, eine umfassende Liste von Anforderungen für die Schachwebseite zu erstellen. Diese Anforderungen bilden das Fundament für sämtliche weitere Überlegungen und Entscheidungen. Basierend darauf wird die Architektur der Plattform festgelegt, Wireframes entwickelt und alle weiteren notwendigen Planungen und Vorbereitungen vorgenommen.

3.1 Anforderungsanalyse

Die Anforderungsanalyse ist ein kritischer Teil in Softwareentwicklungsprojekten. Die bereits definierten Erfolgskriterien geben Anhaltspunkte für die spezifischen Anforderungen an die Applikation, aber definieren nicht unbedingt jede Funktion, die notwendig ist, um sie zu erreichen. Im Folgenden werden die Anforderungen erarbeitet, die notwendig sind für einen funktionierenden MVP.

3.1.1 Analyse bestehende Plattformen

Um Anforderungen zu bestimmen, macht es Sinn die Funktionen von bestehenden, etablierten Plattformen wie chess.com oder lichess.org zu analysieren. Dann kann entschieden werden, welche Funktionen relevant für den MVP sind. Folgend werden die Funktionen dieser beider Plattformen aufgelistet, inklusive einer Priorität, die bestimmt, wie wichtig die Funktion für den MVP ist (1 = höchste Priorität, 3 = niedrigste Priorität). Aufgeführte Funktionen sind in mindestens einer der beiden Plattformen eingebaut (in keiner bestimmten Reihenfolge).

Funktion	**Kommentar**	**Prio**
Online-Spiele mit verschiedenen Zeitformaten	Standardfunktionalität für eine Schachplattform. Es kann eine Bedenkzeit pro Spieler und ein Zeitinkrement pro Zug (Zeit, die zur Bedenkzeit des Spielers dazugerechnet wird, nachdem er seinen Zug abgeschlossen hat) definiert werden.	1
Schach-Puzzles	Schwierige Positionen, in denen der Benutzer den besten Zug finden muss.	3
Spiele gegen einen Computer	Es kann gegen verschiedene Bots gespielt werden.	1
Zuschauer-Modus	Die Möglichkeit Spielen eines anderen Spielers live zuzusehen.	3
Eröffnungsdatenbank	Ein Werkzeug, um Eröffnungen zu erforschen.	3
Endspieldatenbank	Ein Werkzeug, um Endspiele zu erforschen.	3
Anonyme Spiele	Spiele ohne einen Benutzeraccount	1
Benutzerkonto System	Login / Registrieren	1
Elo-System	Spieler mit einem Benutzerkonto werden nach ihren Fähigkeiten bewertet nach dem Elo-System. Nach ein paar Spielen, die zur Auswertung dienen, wird dem Spieler eine Zahl zugewiesen, die seinen Fähigkeiten entspricht.	2
Matchmaking	Automatisch mit einem Gegner gepaart werden. Spieler mit Benutzerkonto werden mit einem Gegner gepaart, der eine ähnliche Elo Bewertung hat.	2
Spielen beitreten	Anders, als das automatische Paaren (Matchmaking) kann auch aus Herausfordern in einer Liste ausgewählt und deren Spiel beigetreten werden.	1
Analyse des Spiels	Detaillierte Analyse mit dem Schachcomputer. Es werden die besten Züge angezeigt, Fehler markiert und eine	1

	Bewertungsleiste wird angezeigt, die den Vorteil einer bestimmten Farbe darstellt.	
Zeichnen auf dem Brett	Auf dem Brett können Pfeile gezeichnet, sowie Felder markiert werden.	3
Lernmaterialien	Materialien und Werkzeuge, um das Spiel zu lernen, oder besser zu werden.	3
Chat-Funktion	Bei Online-Spielen kann mit dem Gegner gechattet werden.	2
Benutzerprofil	Im Benutzerprofil können detaillierte Informationen zu den eigenen Spielen, sowie dem eigenen Elo-Rating angesehen werden.	2
Turniere	Es kann an Online-Turnieren teilgenommen werden.	3
Resignieren	Spiele können zu jedem Zeitpunkt aufgegeben werden.	2
Zug zurücknehmen	In Online-Spielen kann ein Benutzer anfragen, ob er seinen letzten Zug zurücknehmen darf. Der Gegner kann dies bestätigen oder ablehnen. In Spielen gegen den Schachcomputer können Züge ebenfalls rückgängig gemacht werden.	2
Remis-Angebot	Es kann ein Remis angeboten werden. Der Gegner kann annehmen oder ablehnen.	2
Zugnavigation	Bei Spielen wird eine Liste der gemachten Züge angezeigt. Der Benutzer kann durch diese hindurchnavigieren, um vergangene Positionen anzusehen.	1
Community	Benutzer können ausserhalb von Spielen andere Benutzer befreunden, ihre Profile ansehen und ihnen Nachrichten senden.	3
Alternative Spielmodi	Es können alternative Spielmodi mit unterschiedlichen Regeln, abweichend von den Standardschachregeln gespielt werden.	3
Massgeschneiderte Trainings	Anhand von Spielen der Benutzer werden ihnen massgeschneiderte Lektionen und Trainings vorgeschlagen.	3
Schachmeister Datenbank	Spiele von Schachmeistern können angesehen und analysiert werden.	3
Notizen	Für Spiele können Notizen gemacht werden, die später wieder einsehbar sind.	2

3.1.2 Auswertung

Anhand der eingeteilten Priorität wird ausgewertet, welche Funktionen in den MVP eingebaut werden sollen. Dabei wird von allen Funktionen abgesehen, die mit der niedrigsten Priorität 3 markiert wurden. Funktionen mit Priorität 2 sind erwünscht, aber nicht notwendig für den MVP. Funktionen mit der höchsten Priorität müssen zwingend implementiert werden.

3.2 User Storys

Um die Anforderungen zu definieren, werden User Storys verwendet, anstatt detaillierte Anwendungsfälle aus folgenden Gründen:

Einfachheit: User Storys sind prägnant und konzentrieren sich auf den Nutzen für den Benutzer, was sie leicht verständlich macht.

Anpassungsfähigkeit: Sie sind anpassungsfähig, und da es sich um ein Projekt handelt, bei dem möglicherweise Funktionen iteriert werden, lassen sich User Storys leicht anpassen.

Die User Storys sind jeweils mit Akzeptanzkriterien ausgestattet, die erfüllt werden müssen, damit die Story als abgeschlossen gilt. Weiterhin wird die Relevanz spezifiziert, die aussagt, ob die User Story zwingend implementiert sein muss oder nicht. Die User Storys entsprechen nicht eins-zu-eins den Funktionen aus dem letzten Abschnitt, sondern wurden auf die MVP-Version angepasst.

3.2.1 Benutzerregistrierung und Konto

Name	Konto erstellen
Beschreibung	Als Benutzer möchte ich ein Konto erstellen können, um eine personalisierte Erfahrung zu haben.
Akzeptanzkriterien	- Möglichkeit ein Konto zu erstellen - Möglichkeit sich mit dem erstellten Konto anzumelden
Relevanz	Erforderlich

Name	Profil
Beschreibung	Als Benutzer möchte ich meine vergangenen Spiele gegen andere Spieler in meinem Profil ansehen können.
Akzeptanzkriterien	- Möglichkeit eigene Spiele auf einer Seite anzusehen
Relevanz	Erforderlich

3.2.2 Multiplayer

Name	Spiel erstellen
Beschreibung	Als Benutzer möchte ich eine Herausforderung erstellen können, die andere Benutzer annehmen können.
Akzeptanzkriterien	- Möglichkeit Spiele zu erstellen - Möglichkeit Spiele mit Zeitformat zu konfigurieren (Bedenkzeit und Inkrement) - Funktioniert auch für Benutzer ohne Benutzerkonto
Relevanz	Erforderlich

Name	Herausforderungen sehen
Beschreibung	Als Benutzer möchte ich eine Liste der bestehenden Herausforderungen sehen.
Akzeptanzkriterien	- Möglichkeit eine Liste der Herausforderungen zu sehen. In der Liste wird aufgeführt, was für ein Zeitformat das Spiel hat und wer die Herausforderung erstellt hat. - Funktioniert auch für Benutzer ohne Benutzerkonto
Relevanz	Erforderlich

Name	Herausforderung annehmen
Beschreibung	Als Benutzer möchte ich Herausforderungen aus der Liste annehmen können, um gegen den Herausforderer zu spielen.
Akzeptanzkriterien	- Möglichkeit Herausforderungen anzunehmen - Funktioniert auch für Benutzer ohne Benutzerkonto
Relevanz	Erforderlich

Name	Elo Bewertung
Beschreibung	Als Benutzer möchte ich nach meinen Fähigkeiten bewertet werden, nach dem Elo-System
Akzeptanzkriterien	- Benutzer erhalten nach einer bestimmten Anzahl von Spielen eine Bewertung, die ihren Fähigkeiten entspricht.
Relevanz	Erwünschenswert

Name	Paarung
Beschreibung	Als Benutzer möchte ich für Spiele automatisch mit Gegnern mit ähnlichen Fähigkeiten gepaart werden können.
Akzeptanzkriterien	- Möglichkeit in eine Paarungs-Warteschlaufe einzutreten - Sobald ein Partner gefunden wurde, beginnt das Spiel
Relevanz	Erwünschenswert

3.2.3 Schachcomputer und Analyse

Name	Schachcomputer herausfordern
Beschreibung	Als Benutzer möchte ich gegen einen Schachcomputer spielen können.
Akzeptanzkriterien	- Möglichkeit gegen einen Schachcomputer spielen zu können - Es wird eine Bewertung der Position mittels einer Bewertungsleiste angezeigt.
Relevanz	Erforderlich

Name	Analyse eigener Spiele
Beschreibung	Als angemeldeter Benutzer möchte ich meine vergangenen Spiele analysieren können.
Akzeptanzkriterien	- Möglichkeit eigene vergangene Spiele Zug für Zug durchzuspielen - Es wird eine Bewertung der Position mittels einer Bewertungsleiste angezeigt.
Relevanz	Erforderlich

3.2.4 Gameplay

Name	Schachregeln
Beschreibung	Als Benutzer möchte ich in einem Spiel alle legalen Züge in der jeweiligen Position machen können.
Akzeptanzkriterien	- Möglichkeit Züge auszuführen, die in der Position erlaubt sind.
Relevanz	Erforderlich

Name	Spielende
Beschreibung	Als Benutzer möchte, dass die Plattform Situationen wie Schachmatt und Remis automatisch erkennt und das Spiel beendet wird.
Akzeptanzkriterien	- Schachmatt wird erkannt - Remis wird erkannt
Relevanz	Erforderlich

Name	Schachuhr
Beschreibung	Als Benutzer möchte meine verbleibende Bedenkzeit sehen.
Akzeptanzkriterien	- Bedenkzeit wird pro Spieler angezeigt - Wenn die Bedenkzeit abläuft, wird das Spiel beendet.
Relevanz	Erforderlich

Name	Resignieren
Beschreibung	Als Benutzer möchte ich Online-Spiele aufgeben können.
Akzeptanzkriterien	- Möglichkeit ein Spiel aufzugeben
Relevanz	Erwünschenswert

Name	Remis anbieten
Beschreibung	Als Benutzer möchte ich in Online-Spielen Remis anbieten können.
Akzeptanzkriterien	- Möglichkeit Remis anzubieten - Möglichkeit Remis anzunehmen - Möglichkeit Remis abzulehnen
Relevanz	Erwünschenswert

Name	Spielübersicht
Beschreibung	Als Benutzer möchte ich in Spielen eine Übersicht der gemachten Züge sehen.
Akzeptanzkriterien	- Möglichkeit eine Historie aller gemachten Züge anzusehen - Möglichkeit über Züge zu navigieren, um eine vergangene Position anzusehen
Relevanz	Erforderlich

Name	Zug zurücknehmen
Beschreibung	Als Benutzer möchte ich in Spielen meine Züge zurücknehmen können.
Akzeptanzkriterien	- Möglichkeit Züge zurücknehmen zu können. - In Online-Spielen muss der Gegner den Vorschlag zur Rücknahme akzeptieren oder ablehnen können.
Relevanz	Erwünschenswert

Name	Notizen
Beschreibung	Als Benutzer möchte ich in Spielen Notizen machen können.
Akzeptanzkriterien	- Möglichkeit Notizen machen zu können. - Notizen können zu einem späteren Zeitpunkt wieder angesehen werden.
Relevanz	Erwünschenswert

Name	Chatfunktion
Beschreibung	Als Benutzer möchte ich in Echtzeit mit dem Gegner kommunizieren können.
Akzeptanzkriterien	- Möglichkeit Nachrichten im Chat zu schreiben.
Relevanz	Erwünschenswert

3.2.5 Technische Anforderungen

Zusätzlich zu den User Storys gibt es noch wenige technische Anforderungen, die von den Erfolgskriterien abgeleitet werden können. Es werden nicht alle technischen Erfolgskriterien erneut aufgeführt, sondern nur die, die relevant sind für die Architektur der Plattform.

Name	Echtzeit
Beschreibung	Online-Spiele müssen in Echtzeit ablaufen. Um Spiele in Echtzeit abzubilden, muss ein geeignetes Netzwerkprotokoll gewählt werden.

Name	Schachcomputer
Beschreibung	Für den Schachcomputer muss Stockfish eingesetzt werden.

Name	Skalierbarkeit
Beschreibung	Die Plattform muss horizontal skaliert werden können.

Name	Leistungsfähigkeit
Beschreibung	Die Plattform muss hohe Last ertragen können.

3.3 Gesamtarchitektur

In einem nächsten Schritt wird die Gesamtarchitektur der Plattform gestaltet. Hierbei handelt es sich um einen Leitfaden für die Entwicklung, an welchem sich orientiert werden kann. Es werden die nötigen Komponenten evaluiert, die Art der Architektur gewählt und anschliessend visuell dargestellt, wie die Plattform gestaltet werden kann.

3.3.1 Softwarekomponenten identifizieren

Um die Architektur zu definieren, müssen als erstes die notwendigen Softwarekomponenten gefunden werden. Die User Storys und die technischen Anforderungen aus dem letzten Kapitel geben Anhaltspunkte dafür. Alle User Storys sollen mit diesen Komponenten umgesetzt werden können.

3.3.1.1 Webseite

Da die Aufgabenbeschreibung eine Webseite vorgibt, wird auf jeden Fall eine Webseite benötigt. Die Webseite fungiert als Benutzerinterface und kapselt alle Benutzerinteraktionen ein.

3.3.1.2 API

Um die Webseite mit Daten zu befüllen, wird eine Art Schnittstelle benötigt, die die Daten liefert. Ob die Webapplikation und die API aus einer einzelnen Applikation bestehen, oder ob sie getrennt sind, ist zu diesem Zeitpunkt noch unklar.

3.3.1.3 Gameserver

Um der technischen Anforderungen der Echtzeit getreu zu werden, muss eine Art Gameserver her. Der Gameserver verbindet Spieler mit Kontrahenten und kapselt die Logik des Spiels ein.
Spezielle Anforderungen an den Gameserver:

- Skalierbarkeit
- Echtzeit

3.3.1.4 Zuggenerator

Um die Schachregeln zu implementieren, die legalen Züge zu errechnen und Status wie beispielsweise das Ende des Spiels zu erkennen wird ein Mechanismus benötigt, der diese Logik zur Verfügung stellt.

3.3.1.5 Datenbank

Um Daten, wie Benutzerkonten und Spiele der jeweiligen Benutzer zu speichern, wird eine Art Datenbank benötigt.

3.3.1.6 Schachcomputer

Der Schachcomputer Stockfish muss integriert werden.

3.3.2 Komponentendiagramm und Abhängigkeiten

Die erarbeiteten Softwarekomponenten werden im Folgenden visualisiert. Das Diagramm kann als Anhaltspunkt für Abhängigkeiten verwendet werden und gibt vor, welche Komponenten als erstes angegangen werden müssen.

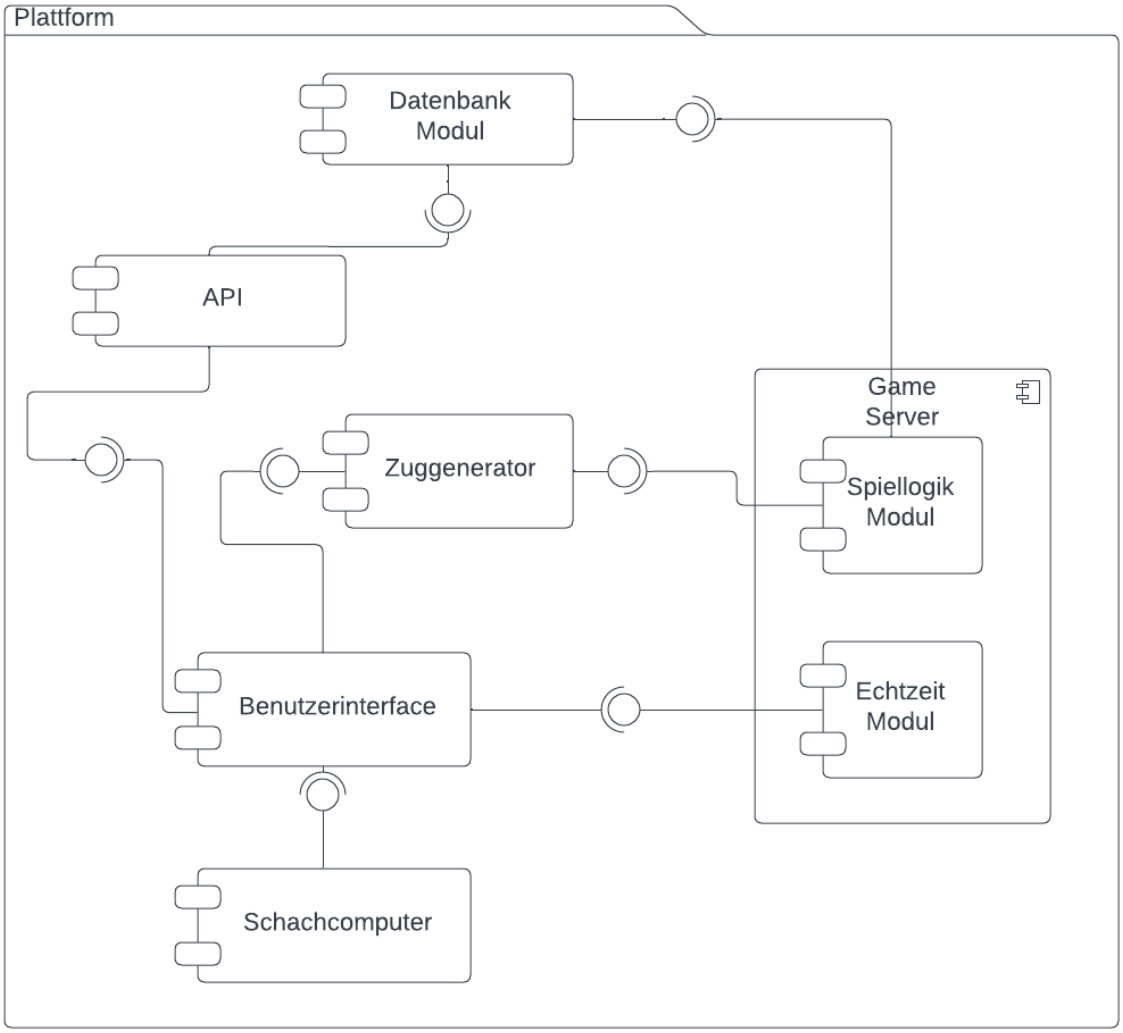

Abbildung 1 Komponentendiagramm

Anhand der Abhängigkeiten lässt sich nun eine Reihenfolge erstellen, in welcher die einzelnen Projekte angegangen werden müssen:

1. Zuggenerator
2. Datenbankmodul
3. Game Server
4. API und Benutzerinterface
5. Schachcomputer

3.3.3 Art der Architektur

Bei Softwareprojekten stellt sich zu Beginn oft die Frage was für eine Art von Architektur angestrebt wird. Hier gibt es zwei verschiedene Muster: Microservices und Monolithen.

Monolithen sind einzelne, einheitliche Anwendungen, bei denen alle Komponenten miteinander verflochten sind. Sie sind einfacher zu entwickeln und bereitzustellen, können aber komplex und

schwierig zu skalieren werden, wenn die Anwendung wächst. Bei jeder Aktualisierung muss die gesamte Anwendung neu erstellt und bereitgestellt werden.

Microservices hingegen bestehen aus kleinen, unabhängigen Diensten, die jeweils bestimmte Geschäftsfunktionen ausführen. Sie bieten mehr Flexibilität und Skalierbarkeit und ermöglichen die Verwendung unterschiedlicher Technologiestapel für verschiedene Dienste. Allerdings bringen sie Herausforderungen bei der Dienstkoordinierung und Datenkonsistenz mit sich.

Im Vergleich dazu eignen sich Monolithen für einfachere, klar definierte Anwendungen, während Microservices für komplexe, skalierbare und sich entwickelnde Anwendungen vorteilhaft sind, da sie Anpassungsfähigkeit und die Fähigkeit bieten, eine erhöhte Last und Funktionserweiterung effizient zu bewältigen [4].

Der Autor ist ein grosser Vertreter davon, Dinge simpel zu halten, solange es geht und erst wenn sich die Notwendigkeit ergibt, eine Komponente in Services aufzuteilen. In diesem Fall möchte er zuerst Technologien auswählen, bevor die Applikation durch unnötiges "overengineering" verkompliziert wird.

3.4 Technologiewahl

Anhand der erarbeiteten Komponenten kann nun festgelegt werden, welche Technologien verwendet werden sollen. Bevor beispielsweise ein Game Server von Grund auf neu gebaut wird, macht es Sinn sich zu informieren, was für Technologien es bereits gibt und wie diese in die Architekturlandschaft passen. Für jede Komponente wird im Folgenden eine Analyse angestellt, die zur Wahl einer passenden Technologie führen soll.

3.4.1 Gameserver

Hier gilt es die technischen Anforderungen zur Skalierbarkeit und Last im Blick zu behalten. Folgende Kandidaten sind während der Recherche aufgetaucht:

3.4.1.1 Nakama

Nakama ist ein robuster und skalierbarer Open-Source-Server, der für Echtzeit-Multiplayer-Spiele, soziale Funktionen und interaktive Anwendungen entwickelt wurde. Er bietet eine Reihe integrierter Funktionen, darunter Matchmaking, Bestenlisten, Multiplayer-Netzwerke und die Integration sozialer Medien. Nakama unterstützt verschiedene Client-Plattformen und Programmiersprachen, so dass Entwickler mit Leichtigkeit ansprechende Multiplayer-Erlebnisse schaffen können. [5]

Hauptmerkmale:

- Multiplattform-Unterstützung: Kompatibel mit iOS, Android, Unity, Unreal Engine und vielen anderen.
- Umfangreiche Funktionen: Bietet integriertes Matchmaking, Ranglisten, Echtzeit-Multiplayer-Unterstützung, etc.
- Modular und erweiterbar: Ermöglicht das Schreiben von benutzerdefinierten Modulen in Lua oder JavaScript, was die Flexibilität erhöht.
- Skalierbarkeit: Horizontale Skalierbarkeit, um eine wachsende Anzahl von Spielern aufzunehmen.

3.4.1.2 Photon

Photon ist ein beliebter Gameserver, der Echtzeit-Multiplayer-Gaming Funktionen für verschiedene Plattformen bietet. Er ist bekannt für seine Zuverlässigkeit, Skalierbarkeit und einfache Integration. [6]

Hauptmerkmale:

- Plattformübergreifende Unterstützung: Photon kann in Spiele integriert werden, die in Unity, Unreal Engine und anderen gängigen Spielentwicklungsumgebungen entwickelt wurden, und unterstützt iOS, Android, PC und Konsolen.
- Echtzeit-Multiplayer: Photon ist für Echtzeitanwendungen optimiert und garantiert niedrige Latenzzeiten und reaktionsschnelles Gameplay.
- Skalierbarkeit: Mit Photon Cloud können Spiele automatisch skaliert werden, um eine wachsende Anzahl von Spielern aufzunehmen.
- Einfacher Gebrauch: Entwickler profitieren von einer umfassenden Dokumentation, Tutorials und einer aktiven Community, die die Implementierung erleichtert.

3.4.1.3 Colyseus

Colyseus ist ein Open-Source Gameserver für Node.js, der aufgrund seiner Einfachheit und Benutzerfreundlichkeit besonders bei Indie-Entwicklern und kleinen Teams beliebt ist. [7]

Hauptmerkmale:

- Einfachheit: Colyseus ist bekannt für seine benutzerfreundliche API und minimale Einrichtungsanforderungen, die es Entwicklern leicht machen, Multiplayer-Funktionen schnell zu implementieren.
- JavaScript-Unterstützung: Da Colyseus auf Node.js basiert, ist es eine gute Option für Entwickler, die mit JavaScript und TypeScript vertraut sind, und gewährleistet Kompatibilität mit Webtechnologien.
- Open Source: Da es sich um Open Source handelt, ist es kostengünstig und bietet die Flexibilität, die Funktionalitäten je nach den Anforderungen des Spiels anzupassen und zu erweitern.
- Community: Die Colyseus-Community ist zwar kleiner als die von Photon, aber aktiv, und Entwickler können auf eine Reihe von Ressourcen und Support zugreifen.
- Skalierbarkeit: Colyseus hat Funktionen, um den Server horizontal zu skalieren und bietet zudem einen Cloud Service für integrierte Skalierbarkeit an.

3.4.1.4 Massgeschneiderte Lösung mit NodeJS und Socket.IO

Eine massgeschneiderte Lösung bedeutet die Implementation von Grund auf selbst zu bauen.

3.4.1.5 Evaluation

Kriterium	**Colyseus**	**Photon**	**Nakama**	**Massgeschneidert**
Einfachheit	Exzellent, Benutzerfreundliche API, schnelles Setup	Medium, verständliche Dokumentation, leichte Lernkurve	Medium, komplexe Funktionen, hohe Setup Zeit	Medium, volle Kontrolle über das Verhalten aber Komplexität der Entwicklung
Kompatibilität	Exzellent, JavaScript Support, Web-freundlich	Gut, C# als Sprache für eigene Logik, Web SDK	Exzellent, supported viele Sprachen	Exzellent, JavaScript basiert
Skalierbarkeit	Gut, geeignet für kleine bis mittelgrosse Spiele, Cloud Service	Exzellent, gebaut für Skalierbarkeit, Cloud Service	Exzellent, hoch skalierbar, gestaltet für enorme Benutzerzahlen	Variabel, hängt ab von der Implementation
Community & Support	Gut, aktive Community, gute Dokumentation	Exzellent, grosse Community, ausführliche Dokumentation	Gut, wachsende Community	Medium, Grosse Community bei NodeJS aber Selbstsupport für die eigene Implementation
Kosten	Niedrig, Open Source, keine initialen Kosten	Medium, Free-plan verfügbar, Kosten steigen mit grösseren Nutzerzahlen	Medium, hohe Kosten assoziiert mit Skalierbarkeit und fortgeschrittenen Funktionen	Niedrig, hohe Zeitkosten für Entwicklung

3.4.1.6 Entscheid

Die vorhin aufgestellte Evaluation hilft dabei einen Entscheid zu fällen. Der Autor hat sich für Colyseus entschieden - die Einfachheit der Umsetzung, der kostenfreie Open-Source Ansatz, die Sprache JavaScript und die gute Dokumentation haben überzeugt. Die Kandidaten Nakama und Photon bieten jeweils Funktionen, die für dieses Projekt nicht notwendig sind und kreieren unnötige Komplexität. Die massgeschneiderte Lösung ist zu viel Aufwand, und anstatt das Rad neu zu erfinden, nimmt Colyseus einen grossen Anteil der Arbeit ab.

3.4.2 Benutzerinterface

Anstatt hier eine detaillierte Analyse anzustellen, wird das React basierte Web Framework *Next.js* verwendet. Die Entscheidung, Next.js für die UI-Anwendung zu verwenden, wurde getroffen, beeinflusst durch umfangreiche Erfahrung des Autors mit diesem Framework. Die Vertrautheit mit dem Ökosystem wird einen rationalisierten Entwicklungsprozess erleichtern und sowohl Effizienz als auch Qualität sicherstellen. Next.js ist bekannt für seine serverseitigen Rendering-Fähigkeiten, die bemerkenswerte SEO-Vorteile und eine verbesserte Leistung beim Laden der ersten Seite bieten, was entscheidend für das Engagement der Benutzer ist. Der reichhaltige Funktionsumfang, einschliesslich integriertem Routing und Erweiterbarkeit, entspricht den Projektanforderungen und verspricht eine robuste und skalierbare Lösung. Die aktive Community und die umfassende Dokumentation untermauern das Vertrauen in Next.js und stellen sicher, dass während der gesamten Entwicklungsphase Zugang zu Support und Ressourcen besteht. [8]

Ein weiterer Grund für die Entscheidung ist, dass Next.js die Möglichkeit bietet API-Routen zu definieren. Somit entfällt die Notwendigkeit eine einzelne Applikation zu erstellen, die das

Benutzerinterface mit Daten versorgen kann. Next.js ist eine Server Applikation und kann sich somit direkt mit der Datenbank verbinden.

3.4.2.1 SSR (serverseitiges Rendering)

In einer traditionellen React-Anwendung, die ausschliesslich im Browser läuft, wird der Client-seitige Rendering-Ansatz verwendet. Hier erhält der Browser eine minimale HTML-Struktur, und JavaScript wird verwendet, um Inhalte dynamisch aufzufüllen. Dies kann zu einem langsameren anfänglichen Laden der Seite führen und ist möglicherweise nicht optimal für SEO, da Suchmaschinen-Crawler Inhalte, die clientseitig gerendert werden, nicht effektiv indizieren können.

Im Gegensatz dazu nutzt Next.js das serverseitige Rendering (SSR), um diese Einschränkungen zu überwinden. Mit SSR wird jede Seite auf dem Server vorgerendert, und der Browser erhält eine vollständig geformte HTML-Seite. Dies beschleunigt nicht nur das anfängliche Laden der Seite und bietet ein besseres Benutzererlebnis, sondern stellt auch sicher, dass der Inhalt für Suchmaschinen-Crawler sofort verfügbar ist, was die SEO-Leistung verbessert. Next.js automatisiert diesen Prozess und ermöglicht es Entwicklern, SSR nahtlos in ihre Anwendungen zu integrieren und so die interaktiven Fähigkeiten des clientseitigen React mit den Leistungs- und SEO-Vorteilen des serverseitigen Renderings zu kombinieren. [9]

3.4.3 Datenbank

Die Wahl der Datenbank hängt von der Art der Daten, die gespeichert werden sollen, ab. Anhand der User Storys ergeben sich nur zwei Datentypen: Benutzer und Spiele. Auf diese beiden Datentypen müssen Abfragen gemacht werden können, z.B. eine Liste von Spielen für einen bestimmten Spieler.

Für die Wahl der Technologie ist essenziell zuerst herauszufinden, was für eine Art der Datenbank eingesetzt werden soll. Dabei kommen zwei verschiedene Muster in den Sinn:

3.4.3.1 SQL

Der etablierte Standard für Datenbanken. SQL-Datenbanken sind Systeme zur Verwaltung und Bearbeitung strukturierter Daten, die SQL (Structured Query Language) verwenden, um in relationalen Tabellen gespeicherte Daten abzufragen und zu bearbeiten.

Anbieter: PostgreSQL, MySQL, etc.

Vorteile:

- Daten sind strukturiert, konsistent und folgen immer zwingend einem definierten Schema
- Sehr nützlich für simple Aggregationen über grosse Datensätze.
- Eine sehr etablierte Technologie, die seit Jahrzenten im Einsatz ist und deshalb unter Entwicklern gut bekannt ist.

Nachteile:

- Skalieren von SQL-Datenbanken kann schwierig sein, vor allem auf distribuierten Systemen (horizontales Skalieren) [10]
- Das Schema ist nicht flexibel. Änderungen am Schema können grosse Datenmigrationen nach sich ziehen
- Performance kann für komplexe Abfragen über grosse Datensätze leiden (zum Beispiel bei vielen Joins)

[11]

3.4.3.2 No-SQL

NoSQL-Datenbanken sind flexibel, skalierbar und für die Speicherung und Verwaltung grosser Mengen unstrukturierter oder halbstrukturierter Daten konzipiert. Sie unterstützen verschiedene Datenmodelle wie das Speichern in Dokumenten, Graphen oder Key-Value.

Anbieter: MongoDB, Cassandra, Redis, etc.

Vorteile:
- Flexibles Schema
- Nutzbar auf distribuierten Systemen
- Hohe Verfügbarkeit und Performance

Nachteile:
- Daten können inkonsistent werden
- Limitierte Abfragemöglichkeiten
- Weniger etablierte Technologie und schwierig handzuhaben

[11]

3.4.3.3 Entscheidung

Es wird entschieden eine SQL-Datenbank für die Plattform einzusetzen. Zum einen sind die Anforderungen simpel – die Daten sind strukturiert und haben klare Abhängigkeiten (Benutzer haben Spiele). Dazu kommt, dass der Autor zwar beide Muster schon angewendet hat, aber mit relationalen Datenbanken vertrauter ist und somit das Risiko von Problemen im Projekt minimiert wird.
Als relationale Datenbank wird PostgreSQL verwendet. PostgreSQL ist bekannt, weit verbreitet im Einsatz und hat sich für den Autor bereits in vergangenen Projekten als sehr zuverlässig herausgestellt. Dazu kommt, dass es Anbieter wie beispielsweise Neon [12] gibt, die PostgreSQL als distribuierte Datenbank anbieten. Damit kann der Nachteil der Skalierung von relationalen Datenbanken einfach umgangen werden.
PostgreSQL unterstützt mehr Funktionen und Arten der Speicherung als MySQL [13] und wird deswegen hier vorgezogen. Dabei muss gesagt werden, dass für die aktuellen Anforderungen an den MVP beide Datenbanken problemlos geeignet wären.

3.4.4 Zuggenerator

Da als Erfolgskriterium definiert ist, dass der Zuggenerator selbst geschrieben wird, besteht hier nicht die Möglichkeit eine Bibliothek oder bestehende Technologie zu verwenden. Die Implementation soll komplett von Grund auf entstehen, mittels Techniken der Schachprogrammierung. Hierfür macht es jedoch Sinn eine geeignete Programmiersprache auszuwählen. Zum Zeitpunkt der Entscheidung weiss der Autor nicht viel über Schachprogrammierung und hat keinerlei Erfahrung damit. Es ist nur bekannt, dass die Schachprogrammierung auf einer tiefen Abstraktionsebene stattfindet, mit Bitmanipulation, etc.

Da der Technologiestack bisher schwer auf JavaScript basiert (Colyseus, Next.js) tendiert der Autor dazu JavaScript für die Zuggenerierung zu verwenden für Konsistenz und Kompatibilität. Der Zuggenerator mit JavaScript kann somit ohne Probleme im Game Server und in der Next.js Applikation verwendet werden.

Dabei kommen allerdings ein paar Risiken auf:
- JavaScript ist grundsätzlich eine Single-Threaded Sprache [14]. NodeJS Workers oder Webworkers im Browser [15] können allerdings Abhilfe schaffen.
- JavaScript stellt dem Entwickler keine Funktionen, um Arbeitsspeicher zu manipulieren zur Verfügung. Arbeitsspeicherverwaltung passiert in JavaScript automatisch. [16]
- JavaScript ist auf Grund seiner Natur als interpretierte Sprache langsamer als Sprachen wie C, C++ oder Rust. Moderne Browser verbessern die Performance allerdings durch JIT-Compiler [17].

Leider liessen sich nicht viele Informationen zu der Eignung von JavaScript für einen Schach-Zuggenerator finden. Allerdings gibt es einige Projekte, die Züge in JavaScript generieren, zum Beispiel *chess.js* [18]. Somit ist das Vorhaben sicher nicht unmöglich und die Vorteile der

Verwendung von JavaScript im Ökosystem des Projekts überwiegen die Risiken. Somit basieren nun alle verwendeten Technologien, mit Ausnahme der Datenbank, auf JavaScript.

3.4.5 Schachcomputer

Dass als Schachcomputer Stockfish eingesetzt wird, wurde bereits definiert. Stockfish ist der stärkste und wohl auch bekannteste Schachcomputer. [19]
Stockfish ist grundsätzlich in C++ geschrieben. Bequemerweise gibt es auch für Stockfish eine JavaScript Version, basierend auf WebAssembly. WebAssembly (wasm) ist ein kompaktes Binärformat, mit dem in Sprachen wie C, C++ und Rust geschriebener Code in Webbrowsern mit nahezu nativer Geschwindigkeit ausgeführt werden kann [20]. Die Plattform Lichess.org verwendet beispielsweise eine WebAssembly Version im Browser, um Positionen zu evaluieren [21]. Diese Version von Stockfish ist öffentlich verfügbar und soll somit auch in diesem Projekt verwendet werden.

3.4.6 TypeScript

Da das gesamte Ökosystem nun auf JavaScript basiert, hat der Autor sich entschieden TypeScript in allen Projekten zu verwenden. TypeScript ist eine stark-typisierte Sprache, aufbauend auf JavaScript. Der Autor ist ein grosser Vertreter von TypeScript und hat umfangreiche Erfahrung. TypeScript hilft dabei, den Programmcode wartbar zu halten und Fehler bereits während der Entwicklung zu finden, anstatt während der Laufzeit. Ausserdem dient TypeScript auch als eine Form der Dokumentation des Programmcodes.

3.5 Datenbankdesign

Anhand der Anforderungen wurden Entitäten eruiert, die für die Umsetzung der Plattform kritisch sind. Das Schema ist glücklicherweise sehr simpel. Um Benutzeraccounts zu speichern, wird eine *User* Tabelle benötigt. Weiterhin sollen pro Benutzer Spiele in der Datenbank gespeichert werden, welche sie gespielt haben. Somit wird eine weitere Tabelle *Game* benötigt.
Damit sind alle User Storys, die mit Datenpersistenz zu tun haben, bereits abgedeckt. Folgendes Schema wurde erarbeitet:

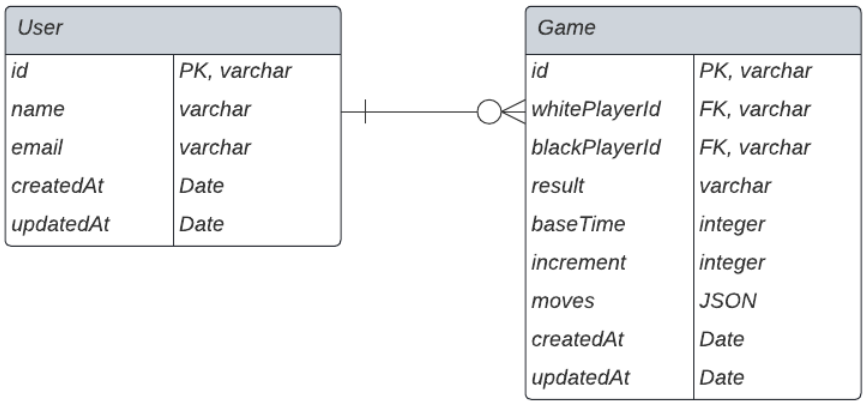

Abbildung 2 Datenbankschema

Es besteht keine Notwendigkeit die Züge in einer separaten, normalisierten Tabelle zu speichern, deshalb werden sie in der Game Tabelle als JSON-Feld gespeichert. Eine Zug-Tabelle würde rasant wachsen und da ein Zug nie unabhängig von einem Spiel behandelt wird (es werden keine Abfragen auf einzelne Züge gemacht), würde dies nur unnötige Komplexität bedeuten.

Es muss dazu gesagt werden, dass das Schema im Laufe der Entwicklung verändert werden kann, falls unvorhergesehene Fälle auftreten.

3.6 Authentifizierung

3.6.1 Benutzerinterface

Wie in den Erfolgskriterien definiert, soll die Authentifizierung gemäss heutigen Standards ablaufen. Auf den meisten Webseiten kann man sich mit Benutzername und Passwort registrieren, oder aber mit einem Drittanbieter wie beispielsweise Google. Diese Methode ist für Benutzer sehr komfortabel, da sie sich für die Webseite nicht ein separates Passwort merken müssen, sondern nur für die Seite des Drittanbieters. Diese Methode der Authentifizierung nennt sich Single-Sign-On (SSO) [22]. Um die Anforderung der Authentifizierung einfach zu halten, wird in dieser Arbeit bewusst davon abgesehen eine Authentifizierung mit Login und Passwort zu implementieren. Stattdessen soll GitHub als Anbieter zur Verfügung gestellt werden. Diese Entscheidung erleichtert den Entwicklungsprozess enorm, indem zeitaufwändige Funktionen, wie das Speichern von verschlüsselten Passwörtern, das Verifizieren von E-Mail-Adressen oder das Zurücksetzen von Passwörtern nicht selbständig implementiert werden müssen. Weiterhin müssen mit dieser Methode keine sensitiven Benutzerdaten gespeichert werden und Funktionen wie Multifaktor Authentifizierung sind schon eingebaut, indem der Anbieter diese unterstützt. Da es sich um einen MVP handelt, ist die GitHub Authentifizierung ausreichend. Zu einem späteren Zeitpunkt können problemlos weitere Methoden der Authentifizierung hinzugefügt werden.

Das Protokoll, auf welchem SSO aufbaut, nennt sich OpenIDConnect (OIDC). Es handelt sich dabei um eine detaillierte und komplizierte Spezifikation, allerdings gibt es zahlreiche Bibliotheken, die diesen Standard korrekt und zuverlässig implementieren. Eine solche Bibliothek soll eingesetzt werden, um den Bedarf für Authentifizierung abzudecken. Es wird davon abgesehen den gesamten OIDC-Standard in dieser Arbeit zu erklären, da dies den Rahmen sprengen würde. Der Standard definiert verschiedene Arten von Authentifizierung, beispielsweise für Browser, für Maschine zu Maschine, etc.

Simplifiziert ist der Ablauf so (OIDC Standard Flow) [23]:

1. Der Benutzer klickt auf den Button des Drittanbieters.
2. Der Benutzer wird zur Seite des Drittanbieters weitergeleitet, wo er sich einloggen muss, wenn er nicht schon eine Session hat.
3. Bei erfolgreicher Authentifizierung beim Drittanbieter wird der Benutzer zurück zur ursprünglichen Webseite geleitet. Dabei wird ein sogenannter *Authorization Code* mitgesendet.
4. Die ursprüngliche Applikation kann den Authorization Code dann verwenden, um eine Anfrage an den Drittanbieter zu machen, welcher den Code validiert und bei Erfolg ein *Access Token* zurückschickt.
5. Das Access Token ist ein Beweis für die Identität des Benutzers und kann validiert werden. Ab diesem Zeitpunkt ist der Benutzer eingeloggt.

Visuell dargestellt:

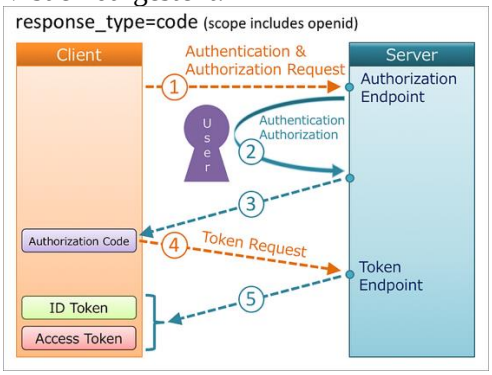

Abbildung 3 OpenIDConnect Ablauf [23]

3.6.2 Game Server

Benutzer sollen sich, wie in den User Storys beschrieben, auf der Webseite anmelden können. Allerdings muss auch der Game Server von unautorisierten Zugriffen geschützt werden. Dafür schickt die Next.js Applikation das Access Token an den Game Server, welcher dieses auf Gültigkeit verifizieren kann und somit die Identität des Benutzers erfährt.

3.7 Gesamtarchitektur

Aus den erarbeiteten Informationen lässt sich nun eine Gesamtarchitektur aufstellen. Dafür wurde ein Diagramm angefertigt, das die einzelnen Services und ihre Kommunikation untereinander illustriert.

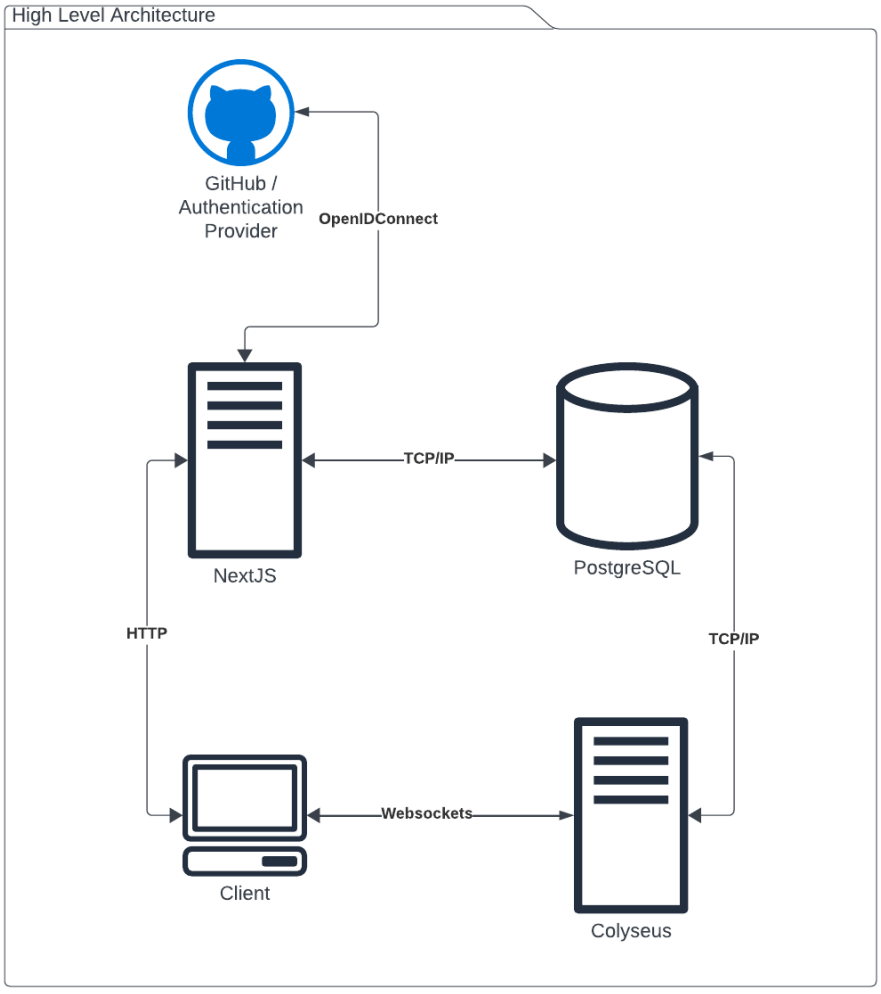

Abbildung 4 Gesamtarchitekturdiagramm

3.8 Wireframes

Für die User Storys wurden folgende Wireframes entwickelt. Diese Wireframes versuchen die User Storys abzudecken, sind aber nicht unbedingt vollständig oder definitiv und gelten nur als Anhaltspunkt für die Entwicklung.

3.8.1 Startseite

Auf der Startseite soll sofort ein Schachbrett mit der Standardaufstellung, sowie eine Bewertungsleiste angezeigt werden. Benutzer können einfach Züge machen, während sich die Bewertungsleiste, welche den Vorteil für eine jeweilige Seite darstellt, verändert (Balken links vom Brett). Weiterhin sind auf der Startseite Links zu den weiteren Seiten, beispielsweise dem Online-Spiel und dem Spiel gegen den Schachcomputer zu finden. Ausserdem gibt es eine Header-Navigation, die auf allen Seiten gleich vorhanden ist. Über den Button in der Navigation können sich Benutzer einloggen, oder wenn sie bereits eingeloggt sind auf ihr Profil zugreifen.

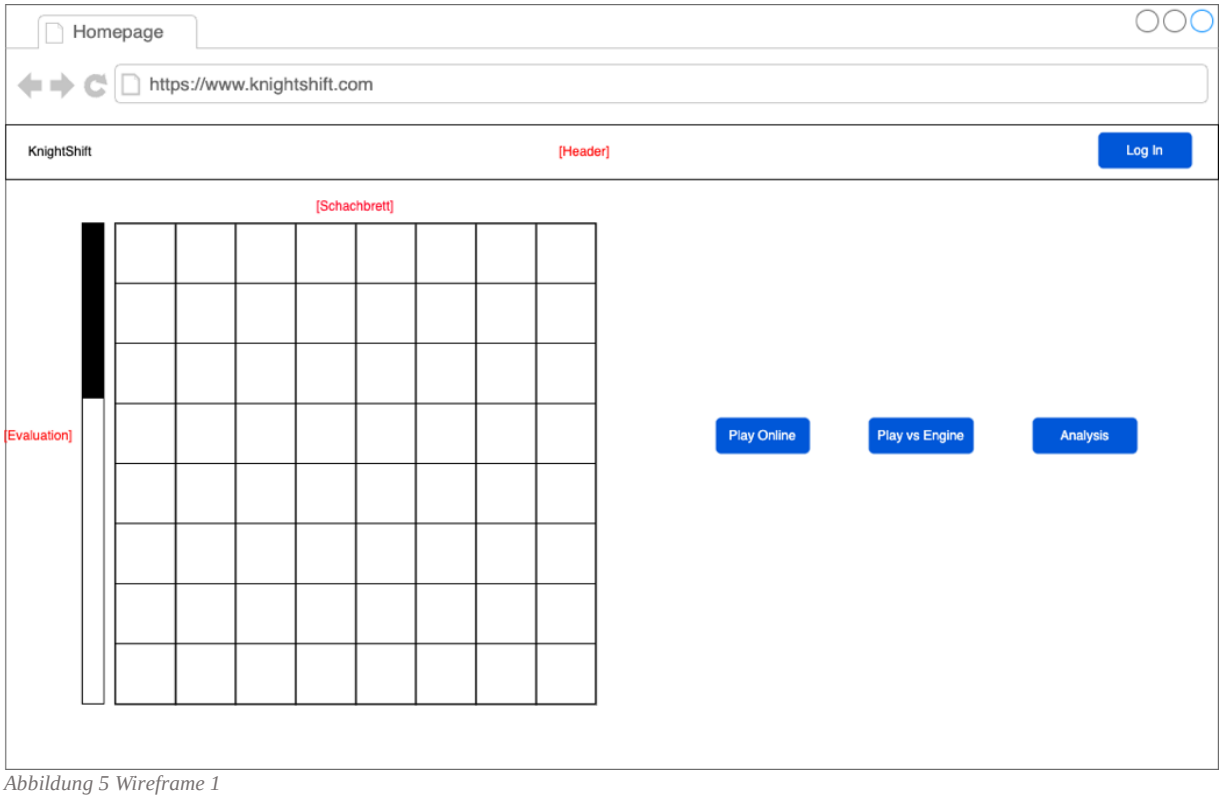

Abbildung 5 Wireframe 1

3.8.2 Analyse Brett / Spiel gegen Schachcomputer

Die Seite, um gegen den Schachcomputer zu spielen oder ein vergangenes Spiel zu analysieren sieht für beide Fälle gleich aus und stellt neben dem Brett eine Liste der gemachten Züge im Spiel dar, über welche navigiert werden kann.

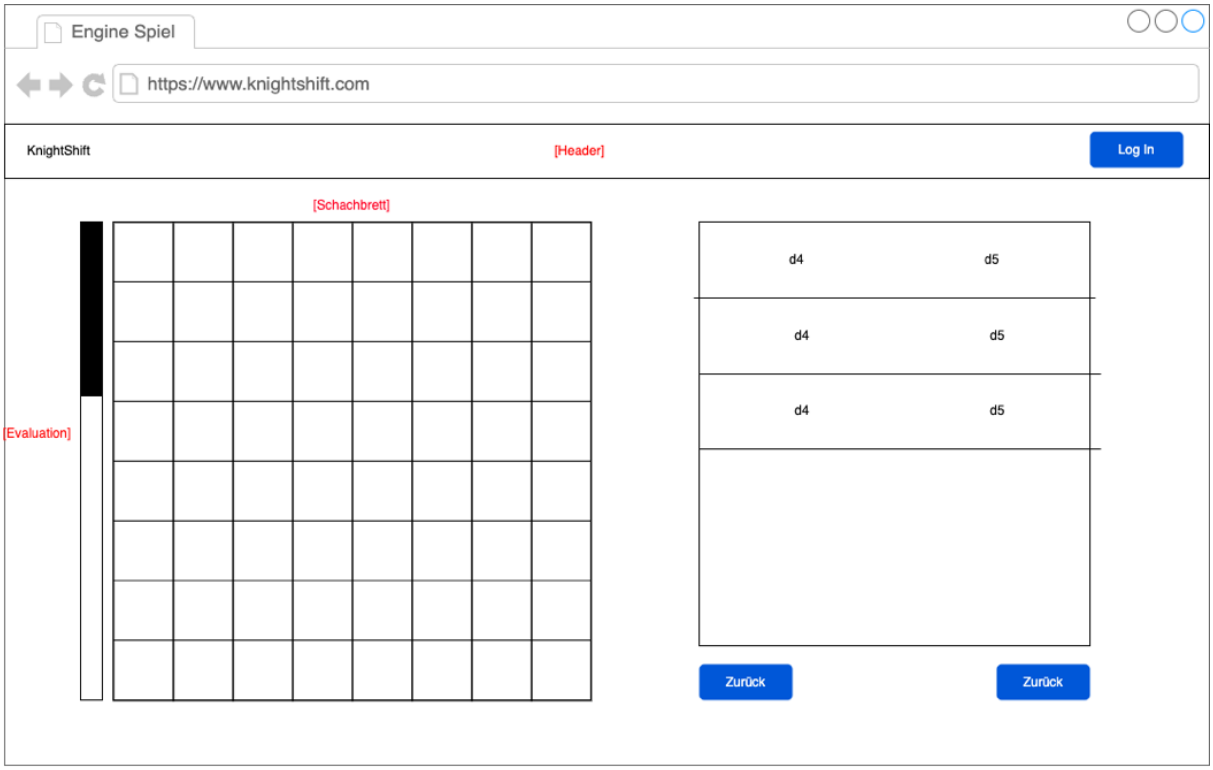

Abbildung 6 Wireframe 2

3.8.3 Spielsuche

Diese Seite existiert, um Spiele suchen zu können oder ein Spiel zu erstellen. Auf der linken Seite sind alle bestehenden Herausforderungen aufgelistet und Benutzer können diese annehmen. Auf der rechten Seite gibt es die Möglichkeit neue Herausforderungen zu erstellen mit einer Bedenkzeit und einem Inkrement.

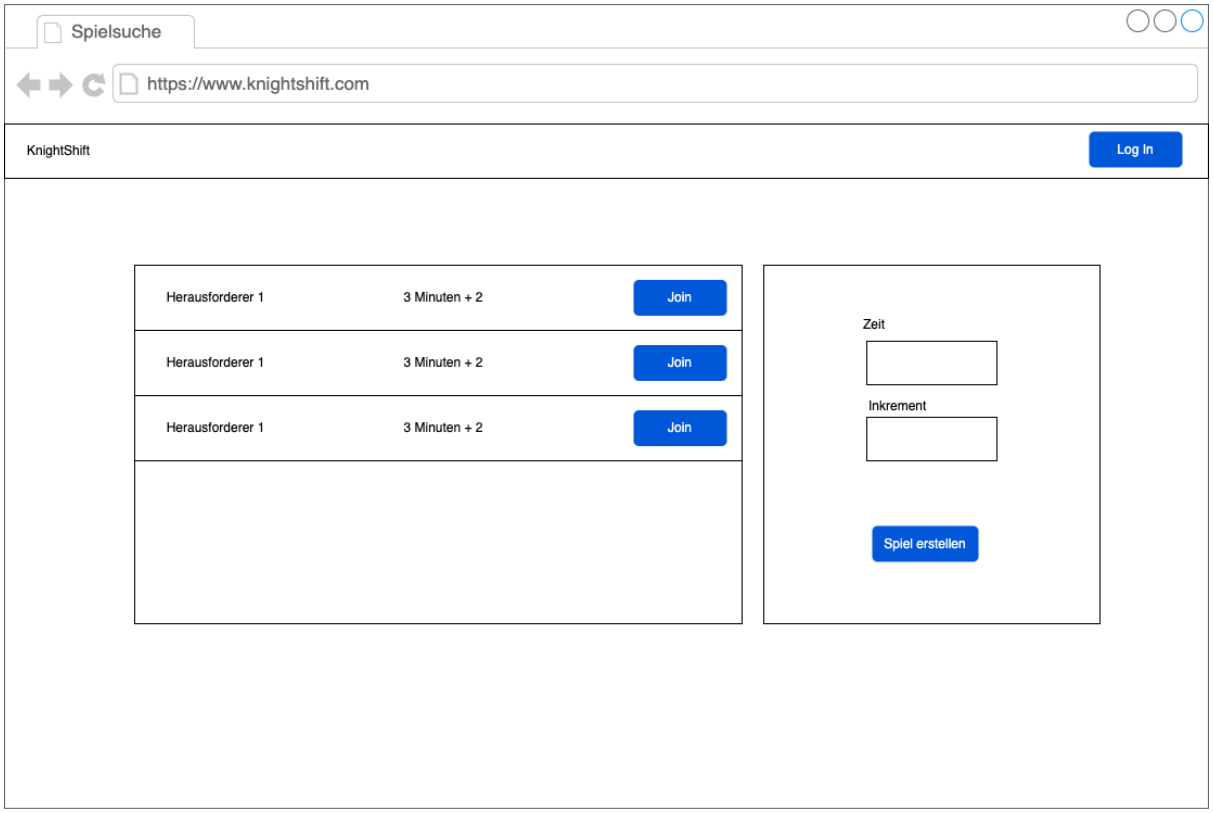

Abbildung 7 Wireframe 3

3.8.4 Online-Spiel

Diese Seite besteht erneut aus den gleichen Komponenten, mit dem Unterschied, dass Informationen zu den Spielern, sowie ihrer verbleibenden Bedenkzeit angezeigt werden, sowie Buttons existieren, um das Spiel zu resignieren oder ein Remis anzubieten. Dazu kommt, dass keine Bewertungsleiste angezeigt wird, da es sich hier um einen kompetitiven Spielmodus handelt.

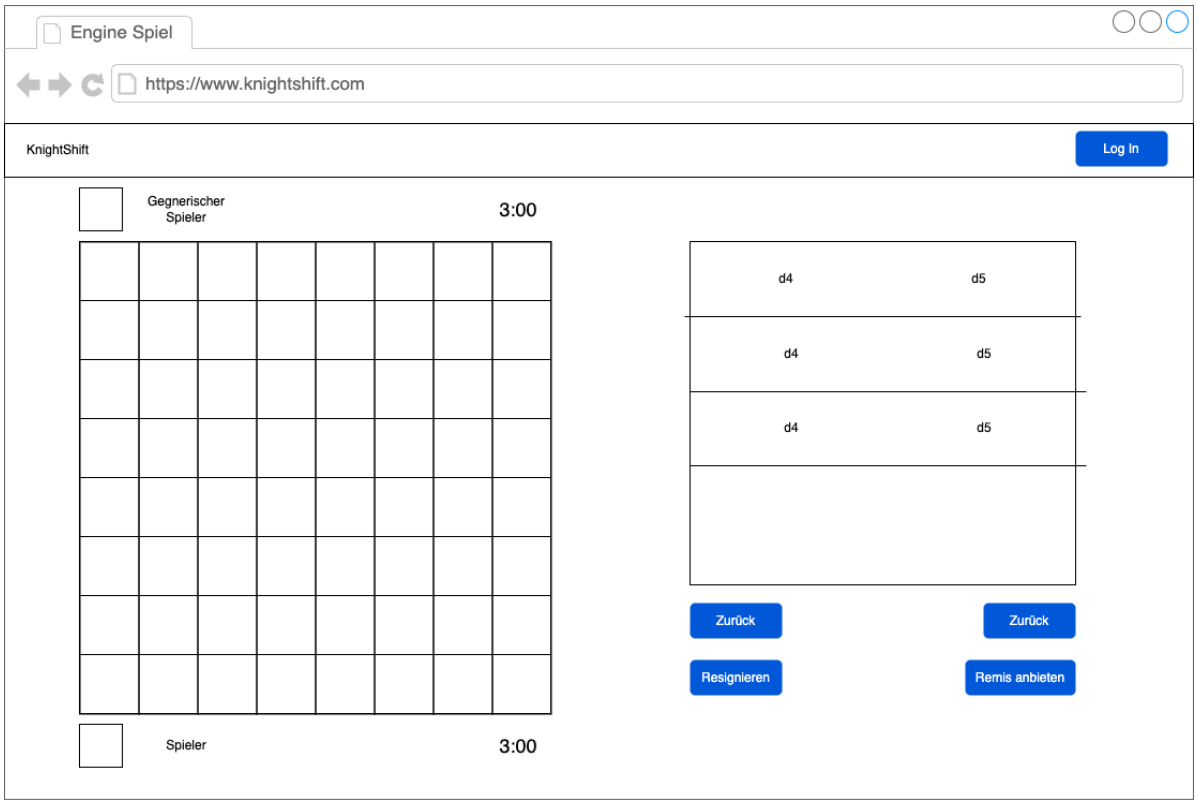

Abbildung 8 Wireframe 4

3.8.5 Profil

Die Profilseite, auf welche man navigieren kann, indem man auf angemeldet ist und auf den Profil-Button im Header drückt, zeigt einfach die vergangenen Spiele und Informationen dazu an. Ein einzelnes Spiel kann angesehen werden, indem der Benutzer auf den “Analysieren”-Button klickt.

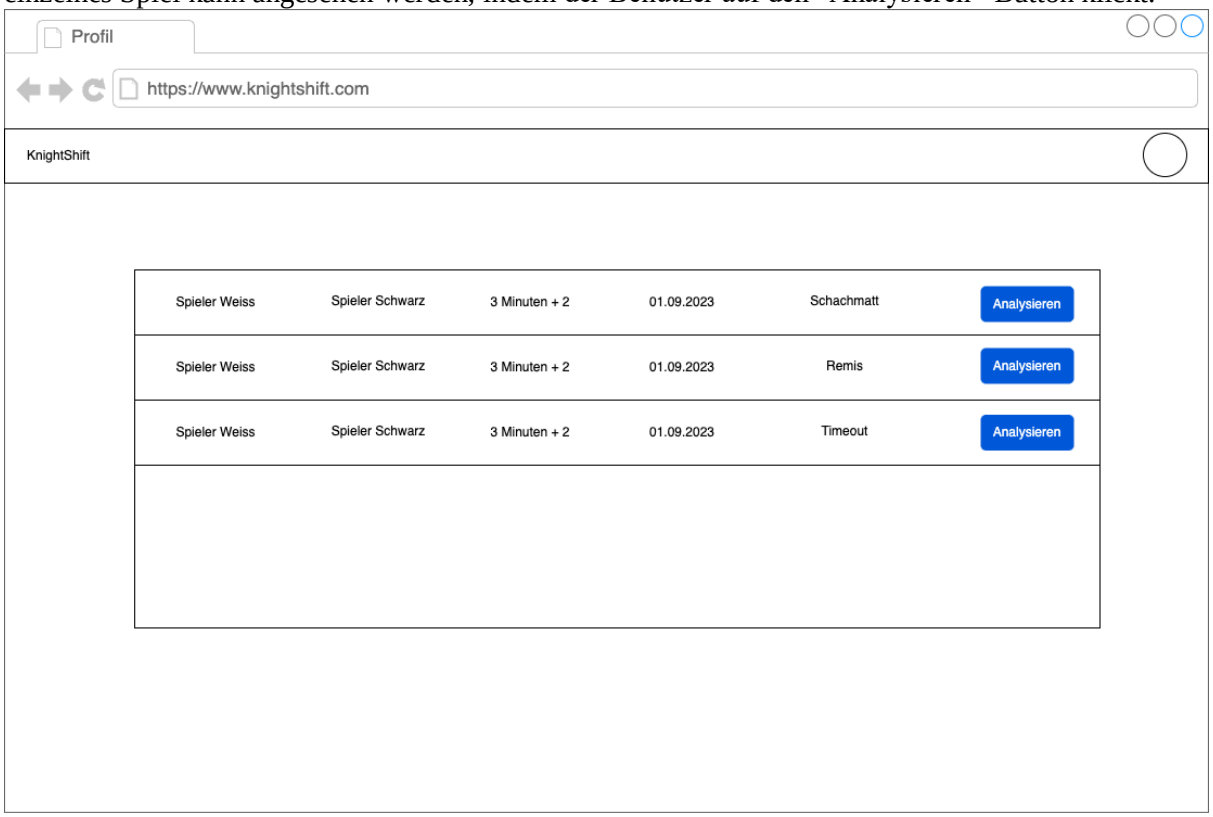

Abbildung 9 Wireframe 5

4 Entwicklungsumgebung

Bevor mit der eigentlichen Implementation begonnen werden kann, muss die Entwicklungsumgebung aufgesetzt werden. Im folgenden Teil wird erklärt, welche Werkzeuge der Autor verwendet hat und wie das Projekt strukturiert ist.

4.1 Werkzeuge

4.1.1 Hostgerät

Der Autor arbeitet auf einem Macbook Pro, 14-Zoll, 2021 mit Apple Silicon M1 Chip und 32 GB Arbeitsspeicher.

4.1.2 Editor

Als Editor verwendet der Autor *Visual Studio Code (*VS Code). VS Code ist ein Editor von Microsoft, der für jede Art von Technologie oder Sprache eingesetzt werden kann. Durch ein gigantisches Angebot von Erweiterungen kann der Editor zu einer vollen Entwicklungsumgebung, angepasst auf die Bedürfnisse des Benutzers, gemacht werden. Speziell verwendet der Autor die *Insider* Version von VS Code, welche unveröffentlichte Funktionen enthält.

4.1.3 TablePlus

Um die Datenbank zu verwalten, wird *TablePlus* eingesetzt, ein Programm, das Verbindungen zu verschiedensten Datenbanken ermöglicht. Über TablePlus können SQL-Abfragen gemacht und Einträge in einem visuellen Benutzerinterface manipuliert werden.

4.1.4 Docker

Das Projekt läuft in Docker-Containern auf dem Hostgerät des Autors.

4.1.5 Git

Als Versionskontrolle wird Git verwendet.

4.2 Monorepository

Das Projekt wird als Monorepository aufgebaut. Das bedeutet, dass sich alle Dateien in einem einzelnen Git Repository befinden (Colyseus, Next.js, Zuggenerator, PostgreSQL). Der Vorteil von dieser Technik ist, dass Dateien zwischen Projekten ausgetauscht werden können. Damit der Zuggenerator beispielsweise im Colyseus und Next.js Projekt angewendet werden kann, muss dieser nicht als mühsam als Paket veröffentlicht und installiert werden, sondern kann einfach am jeweiligen Ort importiert werden. Für die Organisation des Monorepository wird in diesem Projekt *Turborepo* verwendet. Turborepo hat Funktionalitäten wie beispielsweise das parallele Ausführen von Befehlen, sodass mehrere Projekte gleichzeitig gebaut oder gestartet werden können. [24]

4.2.1 Ordnerstruktur

Turborepo teilt Projekte in Pakete und Applikationen auf, welche sich in verschiedenen Ordnern befinden. Pakete sind beispielsweise Bibliotheken, die keine eigenständige Applikationen darstellen (in unserem Falle ist das der Zuggenerator). Applikationen wiederum sind selbsterklärend. Simplifiziert ist die Ordnerstruktur so:

- /
    - apps (Colyseus, Next.js)
    - packages (Zuggenerator, weitere falls nötig)

Jedes Projekt definiert sein eigenes *package.json*. Beispiel:

```
{
  "name": "zuggenerator",
  "version": "0.0.0",
  "scripts": {
  },
  "dependencies": {
  }
}
```

Um nun Programmcode aus diesem Projekt in einem anderen verwenden zu können, kann es als Abhängigkeit hinzugefügt werden.

```
{
  "name": "next.js",
  "version": "0.0.0",
  "scripts": {
  },
  "dependencies": {
    "zuggenerator": "*"
  }
}
```

4.3 Docker mit DevContainers

Das Projekt wird als VS Code DevContainer aufgesetzt. VS Code DevContainers oder Development Containers sind eine Funktion von Visual Studio Code, die es Entwicklern ermöglicht, eine konsistente und reproduzierbare Entwicklungsumgebung innerhalb eines Containers zu erstellen.

Ein DevContainer ist im Wesentlichen ein Docker-Container, der vollständig konfiguriert ist, um die Anforderungen eines bestimmten Entwicklungsprojekts zu erfüllen. Er kapselt alle notwendigen Abhängigkeiten, Tools und Konfigurationen, die für die Entwicklung, das Testen und die Ausführung einer Anwendung erforderlich sind, unabhängig vom Host Betriebssystem. Entwickler können innerhalb dieser containerisierten Umgebung direkt von VS Code aus arbeiten.

Sobald der Container läuft, arbeitet VS Code mit ihm wie mit einer lokalen Entwicklungsumgebung zusammen. Das VS Code-Fenster interagiert direkt mit den im Container installierten Tools, Sprachen und Laufzeiten. Dies macht es zu einem äusserst produktiven Tool für Entwickler, da sie die Abhängigkeiten auf ihrem lokalen Rechner nicht verwalten müssen. Anstatt beispielsweise PostgreSQL auf dem lokalen Rechner zu installieren, wird mit DevContainers automatisch eine PostgreSQL Instanz gestartet, basierend auf der korrekten Version. Nach dem initialen Konfigurieren des DevContainers entfällt somit der gesamte Installationsprozess. Da DevContainers auf Docker basiert können die Container auch unabhängig von VS Code gestartet werden, allerdings fehlt möglicherweise die Integration in die Entwicklungsumgebung. [25]

4.3.1 Konfiguration Docker Images

Es werden zwei Container gebraucht. Einer für PostgreSQL und ein Container, der als Entwicklungsumgebung verwendet wird. Im Entwicklungscontainer können dann die einzelnen Applikationen gestartet werden. DevContainers funktionieren mit *docker-compose*, einem Tool, das es erleichtert mehrere Container in einer YAML-Datei zu konfigurieren:

```yaml
version: "3.8"
networks:
  knight-shift-network:
    name: knight-shift-network
    driver: bridge
volumes:
  postgres_data:
services:
  application:
    container_name: app
    build:
      context: .
      dockerfile: Dockerfile
      args:
        VARIANT: 18-bullseye
        USER_UID: 1000
        USER_GID: 1000
    networks:
      - knight-shift-network
    volumes:
      - ..:/workspace:cached
      - ~/.ssh:/home/node/.ssh:ro
      - /var/run/docker.sock:/var/run/docker.sock
    command: sleep infinity
    user: node
    env_file:
      - ./workspace.env
    ports:
      - 3000:3000
      - 5432:5432
  postgres:
    container_name: postgres
    image: postgres:15.1
    restart: always
    network_mode: service:application
    env_file:
      - ./postgres.env
    volumes:
      - ./init-postgres.sql:/docker-entrypoint-initdb.d/init.sql
```

Das Skript definiert ein Netzwerk, damit Container untereinander kommunizieren können. Weiterhin werden die zwei erwähnten Container konfiguriert. Der erste Container ist der Container, in welchem anschliessend entwickelt wird und basiert auf einem selbstgeschriebenen Dockerfile, das im nächsten Abschnitt behandelt wird. Der Abschnitt *volumes* dient dafür Ordner oder Dateien des Host-Systems zu einem Ordner im Docker Container zu verlinken. Diese Volumes sind: Der Programmcode, der ssh Ordner (für Git) und ein die *docker.sock* Datei, die verlinkt wird, sodass die Docker Container von innerhalb des Containers gesteuert werden kann.

Der zweite Container basiert auf dem offiziellen PostgreSQL Docker Image, Version 15.1. Das Standardpasswort wird mittels Environment Variablen definiert. Zudem wird ein *init.sql* Skript als Volume in den *docker-entrypoint-initdb.d* Ordner innerhalb des Containers verlinkt. SQL-Skripts in

diesem Ordner innerhalb des Docker Containers werden automatisch ausgeführt, wenn der Docker Container erstellt wird. Das Skript erstellt in diesem Fall eine leere Datenbank.

```
CREATE DATABASE knightshift;
```

4.3.2 Dockerfile Entwicklungsumgebung

Der Container, der als Entwicklungsumgebung dient basiert auf einem Image von Microsoft, das speziell für DevContainers zur Verfügung gestellt wird. Es ist eine Debian Distribution, die NodeJS Version 18 bereits installiert hat. [26] Da die verwendeten Technologien (Colyseus und Next.js) auf NodeJS aufbauen ist das optimal. Im Dockerfile wird docker-compose und die Docker CLI installiert, damit die Container auch von innerhalb des Containers über die Kommandozeile gesteuert werden können.

```
ARG VARIANT=18
FROM mcr.microsoft.com/vscode/devcontainers/javascript-node:0-${VARIANT}

RUN apt-get update \
  && apt-get install -y apt-transport-https ca-certificates curl gnupg2 lsb-release
\
  && curl -fsSL https://download.docker.com/linux/$(lsb_release -is | tr
'[:upper:]' '[:lower:]')/gpg | apt-key add - 2>/dev/null \
  && echo "deb [arch=amd64,arm64] https://download.docker.com/linux/$(lsb_release -
is | tr '[:upper:]' '[:lower:]') $(lsb_release -cs) stable" | tee
/etc/apt/sources.list.d/docker.list \
  && apt-get update \
  && apt-get install -y docker-ce-cli

RUN LATEST_COMPOSE_VERSION=$(curl -sSL
"https://api.github.com/repos/docker/compose/releases/latest" | grep -o -P
'(?<="tag_name": ").+(?=")') \
  && curl -sSL
"https://github.com/docker/compose/releases/download/${LATEST_COMPOSE_VERSION}/dock
er-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose \
  && chmod +x /usr/local/bin/docker-compose

ARG USER_UID=1000
ARG USER_GID=$USER_UID
RUN if [ "$USER_GID" != "1000" ] || [ "$USER_UID" != "1000" ]; then groupmod --gid
$USER_GID node && usermod --uid $USER_UID --gid $USER_GID node; fi

RUN groupadd docker
RUN usermod -a -G docker node
```

4.3.3 DevContainers konfigurieren

Um DevContainers zu nutzen, wird noch eine Konfigurationsdatei benötigt. Diese befindet sich im Ordner *.devcontainer* und lässt VS Code wissen, dass es sich beim Projekt um ein DevContainer Projekt handelt.

In der Datei wird die *docker-compose* Datei referenziert, die Ports konfiguriert, die an das Hostbetriebssystem weitergeleitet werden sollen und Erweiterungen definiert, die beim Start installiert werden sollen. Zuletzt kann mit *postCreateCommand* ein Skript angegeben werden sollen, welches

nach dem ersten Start des DevContainers ausgeführt werden soll. Dieses Skript installiert im Falle dieses Projekts einfach die verwendeten Abhängigkeiten im Monorepository mittels dem Paketmanager *npm*.

```
{
  "name": "knight-shift",
  "dockerComposeFile": "docker-compose.yml",
  "service": "application",
  "workspaceFolder": "/workspace",
  "customizations": {
    "vscode": {
      "extensions": [
        ...
      ],
    }
  },

  "forwardPorts": [3000, 8080, 6379],
  "postCreateCommand": "bash -i .devcontainer/init.sh",
  "remoteUser": "node",
  "portsAttributes": {
    "3000": {
      "label": "Frontend"
    },
    "5432": {
      "label": "Postgres"
    },
    "8080": {
      "label": "Game Server"
    },
    "6379": {
      "label": "Redis"
    }
  }
}
```

4.4 Lokales Aufsetzen des Projekts (Anleitung)

Um die Entwicklungsumgebung auf einem neuen Gerät aufzusetzen können folgende Schritte gemacht werden:

4.4.1 DevContainers

Als Voraussetzung muss Docker und VS Code auf dem Host Rechner installiert sein. Zusätzlich wird die VS Code Erweiterung *Remote Development* benötigt.

Link zur Erweiterung:
https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.vscode-remote-extensionpack

Ist diese installiert, kann das Projekt in VS Code geöffnet und über die Kommandopalette (Strg + P) von VS Code das Kommando *Dev Containers: Reopen in Container* ausgeführt werden:

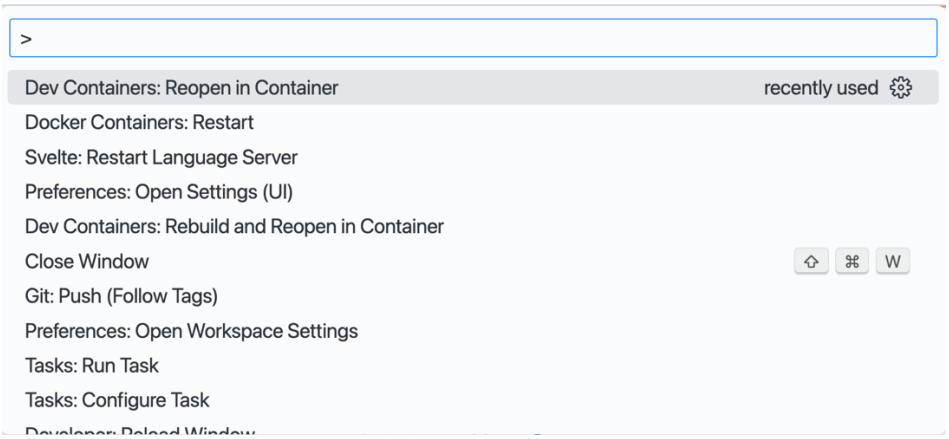

Abbildung 10 VSCode Befehlspalette

Das Kommando erstellt nun die Docker Container anhand der Konfiguration und öffnet ein neues VS Code Fenster. Hat alles funktioniert, befindet sich der Benutzer nun im Kontext des Docker Containers. Dies kann überprüft werden, indem in der Kommandozeile innerhalb von VS Code der Befehl `cat /etc/*-release` ausgeführt wird.

```
node ➜ /workspace (main) $ cat /etc/*-release
PRETTY_NAME="Debian GNU/Linux 11 (bullseye)"
NAME="Debian GNU/Linux"
VERSION_ID="11"
VERSION="11 (bullseye)"
VERSION_CODENAME=bullseye
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"
```
Abbildung 11 Überprüfung Debian System

4.4.2 Umgebungsvariabeln

Damit die Authentifizierung korrekt funktioniert, werden ein paar Umgebungsvariablen benötigt. Dafür muss eine Datei mit dem Namen *.env* im Pfad /apps/web erstellt werden mit folgendem Inhalt:

```
NEXTAUTH_SECRET=7GHhwdM+21Xuw79c6IRuNK8bx1OXMvZA0aJBXEWCdlM=
NEXTAUTH_URL=http://localhost:3000

GITHUB_CLIENT_ID=<einfügen>
GITHUB_CLIENT_SECRET=<einfügen>
```

GitHub Client ID und Secret sind sensitive Informationen und können nicht in diesem Dokument aufgeführt werden. Allerdings können diese zwei Werte mit folgender Anleitung generiert werden: https://docs.github.com/en/apps/creating-github-apps/registering-a-github-app/registering-a-github-app

Die Applikation funktioniert auch, wenn dieser Schritt übersprungen wird, allerdings kann man sich nicht registrieren oder anmelden.

4.4.3 Starten der Applikationen

Nachdem die Container gestartet sind, können folgende Schritte gemacht werden, um die Projekte lokal laufen zu lassen:

4.4.3.1 Datenbank migrieren

Zuerst muss die Datenbank migriert werden. Hier wird in der Dokumentation etwas vorgegriffen, denn es wurde noch nicht dokumentiert, wie die Datenbank aufgesetzt wird. Dies folgt in einem späteren Kapitel. Um die Datenbank zu migrieren, kann folgender Befehl in der Kommandozeile innerhalb des Containers laufengelassen werden:

```
npm run prisma:migrate
```

4.4.3.2 Gameserver und Next.js Applikation starten

Wenn die Datenbank migriert ist, können die Projekte gestartet werden. Dafür müssen diese zuerst gebaut werden:

```
npm run build
```

Anschliessend können die Applikationen gestartet werden:

```
npm start
```

Turborepo startet mit diesem Befehl den Gameserver und die Next.js Applikation. Das Benutzerinterface ist dann unter http://localhost:3000 mit einem Browser erreichbar.

5 Zuggenerator

Im Folgenden wird die Gestaltung und Implementation des Zuggenerators behandelt. In einem ersten Schritt werden alle Regeln, die für den Zuggenerator relevant sind, eruiert. Anschliessend werden daraus Anforderungen erstellt, die dann implementiert werden können. Für die Implementation wird die Theorie jeweils fort zu erklärt. Schachprogrammierung ist ein komplexes Thema, weswegen der Autor es für nötig hält Theorie und Praxis zu vermischen. Einzelne Subthemen bauen aufeinander auf und es wird oft auf Konzepte referenziert, die in einem vorherigen Abschnitt behandelt wurden.

5.1 Regeln für den Zuggenerator

Das Schachspiel hat einige Regeln, die im Zuggenerator korrekt abgebildet werden müssen. Bei Fehlern im Zuggenerator könnten beispielsweise möglicherweise illegale Züge gemacht werden, also Züge die in der jeweiligen Position nicht erlaubt sind. Es ist von hoher Wichtigkeit alle relevanten Regeln des Spiels zu beachten, um eine faire Plattform zu bauen. Um sicherzustellen, dass alle Regeln beachtet wurden, wird in diesem Projekt als Referenz das offizielle Regelbuch der *FIDE*-Organisation (Abkürzung für *Fédération Internationale des Échecs*, zu Deutsch: Internationaler Schachverband) verwendet. *FIDE* ist der Dachverband des Schachsports und regelt grosse internationalen Schachwettbewerbe, wie beispielsweise die Weltmeisterschaften. In offiziellen Wettbewerben rund um die Welt wird mit dem *FIDE*-Regelbuch gespielt. Im Folgenden wird das Regelbuch analysiert und alle Regeln, die für den Zuggenerator relevant sind, werden aufgeführt. Grund für die Ausführung, ist dass die Regeln später zu Anforderungen umformuliert werden. [27]

5.1.1 Allgemeine Regeln

5.1.1.1 Spieler

Schach wird mit zwei Spielern gespielt, welche beide jeweils Figuren einer Farbe steuern. Typischerweise sind die Farben Schwarz und Weiss. Die Spieler sind abwechselnd am Zug. Der Spieler mit den weissen Figuren darf immer den ersten Zug machen.

5.1.1.2 Schachbrett

Das Schachbrett ist ein Raster (8x8) bestehend aus 64 gleichgrossen Quadraten, deren Farbe schwarz und weiss alterniert. Das Brett befindet sich zwischen den zwei Spielern, sodass das nahe Eckfeld rechts des Spielers ein weisses Feld ist. Die waagerechten *Reihen* sind nummeriert von 1-8 und die senkrechten *Linien* werden alphabetisch mit A-H bezeichnet. Diese Bezeichnungen werden immer von der Seite des weissen Spielers betrachtet. Felder können somit mit der Linie und der Reihe, wie in einem Koordinatensystem referenziert werden. Beispiel: e8, d5, a1

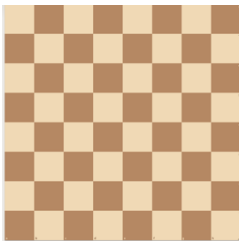

Abbildung 12 Leeres Schachbrett

5.1.1.3 Aufstellung

Jeder Spieler startet mit einer Aufstellung von 16 Figuren, die jeweils die ersten zwei Reihen, die am nächsten zum Spieler sind, besetzen.

Abbildung 13 Schachbrett mit Aufstellung

5.1.2 Figuren und Bewegungen

Figuren werden jeweils von einer Grafik repräsentiert und haben einzigartige Eigenschaften. Figuren können nicht auf Felder gezogen werden, welche von einer Figur derselben Farbe besetzt ist. Wenn eine Figur auf ein Feld gezogen wird, das von einer Figur der gegnerischen Farbe besetzt ist, wird die gegnerische Figur aus dem Spiel genommen. Das nennt sich *Schlagen*. Die hier verwendeten Grafiken werden zur Verfügung gestellt von *Wikimedia Commons*. [28]

5.1.2.1 Bauer

In ihrem ersten Zug können Bauern ein oder zwei Felder nach vorne ziehen, in den folgenden Zügen nur ein Feld. Bauern schlagen diagonal, indem sie ein Feld diagonal vorwärts ziehen, um eine gegnerische Figur zu schlagen. Bauern können nicht vorwärts schlagen. Bauern können nicht rückwärts und nicht durch andere Figuren hindurch ziehen.

5.1.2.2 Springer

Springer ziehen in L-Form, zwei Felder in eine Richtung und dann ein Feld in die andere Richtung dazu. Springer sind die einzigen Figuren, die über andere Figuren "springen" können, das heisst, dass sie ein Feld erreichen können, selbst wenn eine Figur im Weg steht.

5.1.2.3 Läufer

Läufer ziehen diagonal auf den Feldern der Farbe, auf der sie stehen. Sie können beliebig viele Felder entlang einer Diagonalen ziehen, solange sie nicht durch andere Figuren hindurch ziehen.

5.1.2.4 Turm

Türme ziehen horizontal oder vertikal, beliebig viele Felder entlang einer Reihe oder Linie. Wie Läufer können sie nicht durch andere Figuren hindurch ziehen.

5.1.2.5 Dame

Die Dame kann sich in jede Richtung entlang einer Reihe, Linie oder Diagonalen bewegen. Im Grunde genommen kann die Dame sich wie ein Läufer und ein Turm kombiniert bewegen. Die Dame kann ebenfalls nicht durch andere Figuren hindurch ziehen.

5.1.2.6 König

Der König kann sich ein Feld in jede Richtung, entlang einer Reihe, Linie oder Diagonalen, bewegen. Der König darf nicht ins Schach ziehen, d.h. er darf nicht auf ein Feld ziehen, wo er von einer gegnerischen Figur angegriffen würde. Der König kann nicht geschlagen werden. Ausserdem dürfen zwei König nicht auf angrenzenden Feldern stehen. Es muss immer ein Feld Abstand sein.

5.1.3 Spezielle Züge

5.1.3.1 En Passant

Zieht ein Bauer von seiner Ausgangsstellung aus zwei Felder vorwärts und landet auf einem Feld links oder rechts neben einem gegnerischen Bauern, darf der Gegner den Bauern schlagen, als wäre er nur ein Feld vorwärts gezogen. Diese Regel gilt nur für einen Zug - Entscheidet der Gegner sich dagegen, *En Passant* zu nutzen, verfällt das Recht im nächsten Zug.

5.1.3.2 Beförderung

Bauern werden zu einer beliebigen anderen Figur (ausser König), wenn sie die hintere Reihe des Gegners erreichen. Der Spieler muss eine Figur auswählen, zu welcher er den Bauern befördern möchte.

5.1.3.3 Rochieren

Unter speziellen Bedingungen kann ein Spieler eine Bewegung mit dem König und einem Turm in einem einzelnen Zug machen. Es ist das einzige Mal in Schach wo mit einem Zug zwei Figuren bewegt werden können. Rochieren ist auf beide Seiten möglich:

5.1.3.3.1 Rochieren auf die Seite des Königs

Der König zieht zwei Felder nach rechts (von e1 nach g1 für Weiss, oder von e8 nach g8 für Schwarz).
Der Turm springt über den König und landet auf dem Feld unmittelbar links von der Stelle, wo der König landet (von h1 nach f1 für Weiss oder von h8 nach f8 für Schwarz).

5.1.3.3.2 Rochieren auf die Seite der Dame

Der König zieht zwei Felder nach links (von e1 nach c1 für Weiss, oder von e8 nach c8 für Schwarz).
Der Turm springt über den König und landet auf dem Feld unmittelbar rechts von der Stelle, wo der König landet (von a1 nach d1 für Weiss oder von a8 nach d8 für Schwarz).

5.1.3.3.3 Bedingungen

Rochieren ist nur unter folgenden Bedingungen erlaubt.

- Es stehen keine Figuren zwischen dem König und dem Turm.
- Der König und der betroffene Turm haben sich im Spiel noch nicht bewegt.
- Es wurde noch nicht rochiert. Rochieren ist pro Seite nur einmal pro Spiel möglich.
- Der König steht nicht im Schach.
- Der König geht nicht durch ein Schach. Die Felder, über welche sich der König bewegt, dürfen nicht vom Gegner attackiert sein.
- Der König steht nach der Rochade nicht im Schach.

5.1.4 Weitere Regeln

5.1.4.1 Schach

Ein Spieler steht im Schach, wenn sein König von einer gegnerischen Figur bedroht wird. Der Spieler muss in seinem nächsten Zug aus dem Schach ziehen.

5.1.4.2 Schachmatt

Ein Spieler ist schachmatt, wenn sein König im Schach steht und es keine legalen Züge gibt, um den König aus dem Schach zu entfernen. Der Spieler, der schachmatt ist, verliert die Partie.

5.1.4.3 Legale Züge

Ein Zug ist nicht erlaubt, wenn der König in der daraus resultierenden Position im Schach steht.

5.1.5 Remis

Das Schachspiel endet unter folgenden Bedingungen mit einem Unentschieden.

5.1.5.1 Patt

Die Partie endet in einem *Patt*, wenn der Spieler, der am Zug ist, nicht im Schach steht aber keine legalen Züge mehr hat und somit nicht ziehen kann.

5.1.5.2 Dreifachrepetition

Wenn die exakt gleiche Position drei Mal in einem Spiel vorkommt, endet das Spiel in einem Remis.

5.1.5.3 50-Zug-Regel

Beide Spieler haben fünfzig Züge gemacht, ohne einen Bauern zu ziehen oder eine Figur zu schlagen.

5.1.5.4 Unzureichende Figuren

Beide Spieler haben nicht genügend Figuren, um schachmatt zu setzen.

5.2 Anforderungen

Abgeleitet aus den erforschten Regeln werden nun konkrete Anforderungen an den Zuggenerator aufgestellt.

Anforderung	Beschreibung
Standardbewegungen	Alle Figuren können ihre Standardbewegungen ausführen, gem. beschriebenen Regeln, sofern die Bedingungen zutreffen.
Rochade	Spieler können rochieren, sofern die Bedingungen zutreffen, gem. beschriebenen Regeln.
Beförderung	Bauern können befördert werden, gem. beschriebenen Regeln.
En Passant	Bauern können En Passant ausführen, gem. beschriebenen Regeln.
Schach	Könige, die im Schach stehen, müssen sich im nächsten Zug aus dem Schach bewegen.
Schachmatt	Das Spiel endet, wenn ein Spieler schachmatt ist, und es können keine weiteren Züge mehr gemacht werden.
Legale Züge	Nur Züge können gemacht werden, die den jeweiligen Spieler in der resultierenden Position nicht im Schach stehen lassen.
Remis	Das Spiel endet im Remis, wenn die beschriebenen Bedingungen zutreffen.

5.3 Planung Schnittstelle

Bevor mit der eigentlichen Implementierung begonnen wird, ist es ratsam, sich intensiv darüber Gedanken zu machen, wie das Endprodukt später genutzt wird. Dieser präventive und durchdachte Ansatz wird als "API-first" bezeichnet [29]. Dabei steht zu Beginn die Definition der Schnittstelle im Mittelpunkt. Es sollte genau analysiert werden, welche Informationen und Funktionen der spätere Nutzer oder ein anderer Dienst von der Schnittstelle erwartet und wie die Interaktion gestaltet sein soll. Die API-first-Strategie bietet mehrere Vorteile:

Test-driven Entwicklung: Indem man sich zuerst auf die API konzentriert, kann ein Test-driven Ansatz verfolgt werden. Das bedeutet, dass automatisierte Tests für die Schnittstelle erstellt werden, noch bevor ein einziges Stück des Hauptcodes geschrieben wurde. Diese Tests dienen dann als Leitfaden für die Implementierung. Während des Entwicklungsprozesses werden diese Tests kontinuierlich durchgeführt. Erst wenn alle Tests erfolgreich abgeschlossen sind, kann davon ausgegangen werden, dass die Implementierung korrekt und gemäss den Vorgaben ist.

Klare Dokumentation: Die vorab definierte Schnittstelle kann als eine Art Leitfaden oder Dokumentation für den Entwickler dienen. Sie gibt nicht nur einen klaren Überblick über die zu erwartenden Funktionen und den Umgang mit der API, sondern erleichtert auch die Einarbeitung und den Überblick für alle, die später mit dem Code arbeiten oder ihn überprüfen möchten.

Strukturierte Vorgehensweise: Die API-first Methode fördert eine organisierte und strukturierte Arbeitsweise. Durch die klare Definition der Schnittstelle zu Beginn des Projekts wird das Risiko reduziert, dass während der Implementierung unerwartete Herausforderungen oder Änderungen auftreten.

5.3.1 Anforderungen an die Schnittstelle

Der Name des Zuggenerators ist Programm. Folgende Anforderungen muss die Schnittstelle erfüllen.

- Es kann ein Schachspiel erstellt werden, von der Standardaufstellung ausgehend.
- Es kann ein Schachspiel erstellt werden, von einer beliebigen Position ausgehend.
- Die Schnittstelle bietet dem Konsumenten eine Möglichkeit alle legalen Züge der jeweiligen Position zurückzugeben.
- Die Schnittstelle stellt eine Methode zur Verfügung, um einen Zug in der aktuellen Position zu machen.
- Die Schnittstelle stellt eine Repräsentation der aktuellen Position zur Verfügung.
- Die Schnittstelle stellt Informationen zum Status des Spiels, beispielsweise schachmatt, zur Verfügung.

5.3.2 Schnittstellendesign

Anhand der im vorherigen Abschnitt eruierten Anforderungen hat der Autor folgende Schnittstelle entworfen.

Der Zuggenerator soll eine Klasse sein, die mit einem Parameter instanziiert werden kann. Dieser Parameter beschreibt eine vollständige Schachposition. Um eine Schachposition zu beschreiben, wird die *Forsyth-Edwards Notation (FEN)* verwendet. [30]

5.3.2.1 Forsyth-Edwards Notation

Diese Notation erlaubt es den vollständigen Status einer Schachposition als eine Zeichenkette darzustellen.

Beispiel einer *FEN* (Diese *FEN* beschreibt die Standardposition):

```
rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
```

Die Notation besteht aus sechs Teilen, die jeweils mit einem Leerzeichen getrennt sind.

1. Der erste Teil beschreibt, wo sich die Figuren auf dem Brett befinden. Die Figuren werden jeweils mit einem Buchstaben repräsentiert. Kleingeschriebene Buchstaben stehen für schwarze Figuren und grossgeschriebene Buchstaben für weisse Figuren. Für nichtbesetzte Felder wird eine Zahl, nämlich die Anzahl nicht besetzter Felder nebeneinander verwendet. Die Reihen auf dem Schachbrett sind getrennt mit einem Schrägstrich und die Reihen werden von oben nach unten betrachtet. Folgend die Abkürzungen für die Figuren:

Figur	**Buchstabe**
Turm	R/r
Läufer	B/b
Springer	N/n
Dame	Q/q
König	K/k
Bauer	P/p

2. Der zweite Teil beschreibt, welche Farbe aktuell am Zug ist. Für weiss wird ein *w* verwendet und für schwarz ein *b.*

3. Der dritte Teil beschreibt die Rochaderechte für beide Farben. Ein *K*, wenn auf die Seite des Königs rochiert werden darf und ein *Q*, wenn auf die Seite der Dame rochiert werden darf. Für Schwarz werden Buchstaben kleingeschrieben.
4. Der vierte Teil beschreibt das *En-Passant* Feld. Wenn die Bedingungen für *En-Passant* zutreffen, ist dieser Teil der FEN definiert als Reihe + Linie. Beispiel: *e3*
5. Der fünfte Teil beschreibt die *Half-Move-Clock*, ein numerischer Wert, der für die *50-Zug-Regel* verwendet wird, die im vorherigen Kapitel behandelt wurde. Der Wert ist ein Zähler der Anzahl Züge, die gemacht wurden, ohne einen Bauern zu bewegen oder eine Figur zu schlagen.
6. Der sechste Teil, ebenfalls ein numerischer Wert, beschreibt die Anzahl Zyklen, die im Spiel gemacht wurden. Ein Zyklus beinhaltet jeweils den Zug von Weiss und Schwarz.

Anhand dieser Informationen kann die grobe Signatur der Zuggenerator Klasse definiert werden. Die Klasse soll wie folgt instanziiert werden können:

```
new ChessBoard(
  "rnbqkbnr/pp1ppppp/8/2p5/4P3/5N2/PPPP1PPP/RNBQKB1R b KQkq - 1 2"
);
```

Dabei macht es Sinn, wenn ein Zuggenerator auch direkt von der Ausgangsposition aus instanziiert werden kann, ohne eine *FEN* als Parameter mitzugeben, da das vermutlich der häufigste Fall ist. Dies kann mit einer statischen Methode erreicht werden:

```
ChessBoard.fromInitialPosition();
```

Die Klasse soll nach dem Instanziieren intern die *FEN* in ein Objekt übersetzen und anhand von diesen Informationen die legalen Züge errechnen. Die Züge sollen als öffentlicher Getter auf der Klasse verfügbar sein. Züge werden ebenfalls als eine Klasse repräsentiert. In folgendem Anwendungsfall, wird über den Getter *legalMoves* auf ein Array von Zug-Klassen zugegriffen. Die eigentliche Implementation folgt in einem späteren Kapitel.

```
const board = ChessBoard.fromInitialPosition();
const moves = board.legalMoves;
```

Eine Zug-Klasse kann wiederum verwendet werden, um einen Zug zu machen, beispielsweise so:

```
board.makeMove(move);
```

Es sollen weitere Getter zur Verfügung gestellt werden, zum Beispiel für Rochaderechte, En-Passant Feld, Brettrepräsentation, etc. Diese werden aus Gründen der Ausführlichkeit nicht speziell aufgeführt. Das Schnittstellendesign gilt nur als grobe Orientierung für die Implementation.

5.3.3 Klassendiagramm

Zur Veranschaulichung der Schnittstelle ist ein Klassendiagramm nach der UML [31] hilfreich.

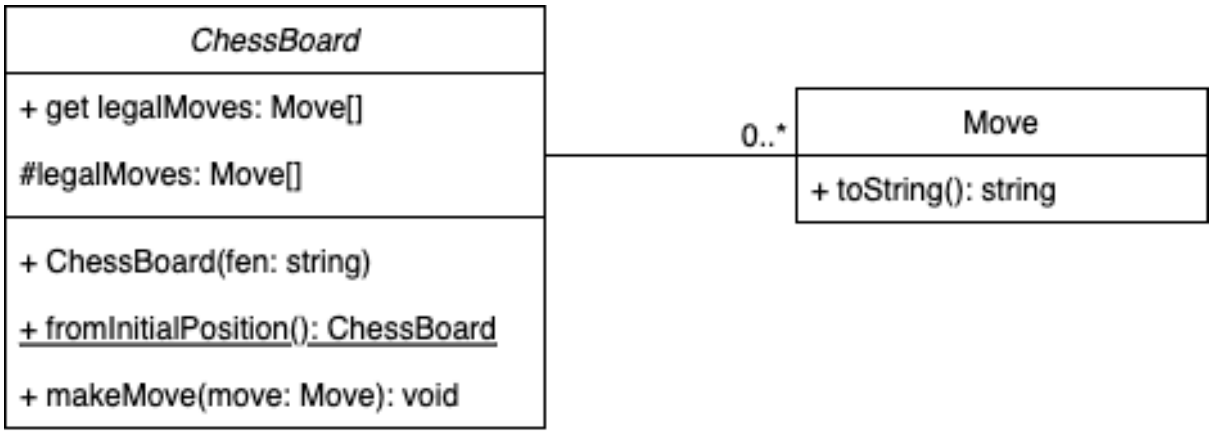

Abbildung 14 Klassendiagramm Zuggenerator

5.3.4 Flussdiagramm

Folgendes Diagramm stellt den Prozess, den der Zuggenerator durchläuft, bildlich dar.

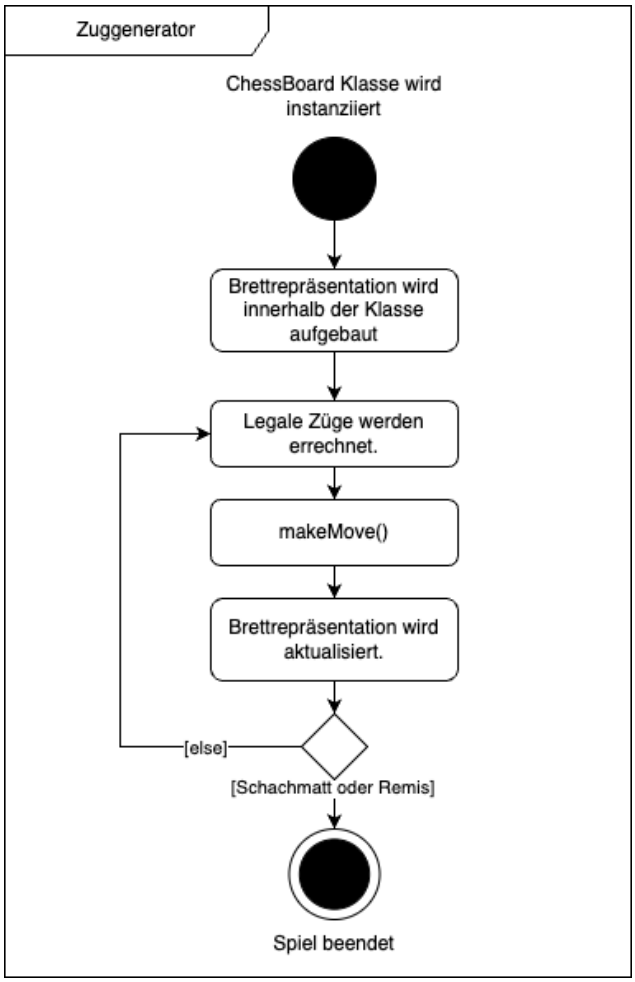

Abbildung 15 Flussdiagramm Zuggenerator

5.4 Brett Repräsentation und Datenstrukturen

Ein Zuggenerator braucht eine interne Brett Repräsentation für die Errechnung der legalen Züge. Eine Schachposition muss vollständig im Zuggenerator abgebildet werden, damit Züge korrekt generiert werden können. Dazu gehören alle Informationen aus der im vorherigen Kapitel beschriebenen *FEN*.

- Position der Figuren
- Wer ist am Zug?
- Rochaderechte
- *En-Passant* Feld
- *Half-Move-Clock*
- Anzahl Zyklen / ganze Züge (eher unwichtig für Zuggeneration, da nicht Teil einer Regel, aber wird zur Vollständigkeit trotzdem integriert)

Im Folgenden wird für jede dieser Informationen eine passende Datenstruktur bestimmt.

5.4.1 Position der Figuren

In der Schachprogrammierung Theorie gibt es einige verschiedene Arten von Datenstrukturen, die geeignet sind für die Position der Figuren [32]. Eine passende Datenstruktur zu wählen ist wichtig, da diese hilft Züge einfach und performant zu berechnen und Randbedingungen abzudecken. Folgend werden die zwei populärsten Ansätze verglichen und anschliessend die gewählte Lösung dokumentiert. Es ist nötig ein wenig in die Theorie einzugehen, um den Lösungsweg des Autors verständlich zu machen.

5.4.1.1 Mailbox Repräsentation

Die Mailbox besteht aus einem Array, das das Schachbrett darstellt. Es gibt zwei gängige Ansätze, dieses Array zu strukturieren [33]:

120-Elemente-Array: Dies ist die klassische Mailbox-Darstellung. Das Array enthält sowohl tatsächliche Brettfelder (64 Elemente) als auch "Off-Board"-Felder (56 Elemente). Die Off-Board-Felder vereinfachen die Zuggenerierung, indem sie eine Pufferzone um das Brett herum bilden, die verhindert, dass Figuren während der Zugberechnung auf die andere Seite "wandern".

64-Elemente-Array: Eine kompaktere Darstellung, die direkt auf die 64 Felder des Schachbretts abgebildet wird. Sie ist einfacher, aber es fehlt der Puffer ausserhalb des Brettes, was die Zugberechnung etwas komplexer macht.

Der Index des Feldes bzw. seine Linie und seine Reihe wirken wie eine Adresse für einen Briefkasten, der leer sein oder eine Schachfigur enthalten kann. Daher auch der Name Mailbox.

Pseudocode Beispiel für eine 64-Elemente-Mailbox (8x8):

```
[
    ROOK, KNIGHT, BISHOP, QUEEN, KING, BISHOP, KNIGHT, ROOK,
    PAWN, PAWN,   PAWN,   PAWN,  PAWN, PAWN,   PAWN,   PAWN,
    null, null,   null,   null,  null, null,   null,   null,
    null, null,   null,   null,  null, null,   null,   null,
    null, null,   null,   null,  null, null,   null,   null,
    null, null,   null,   null,  null, null,   null,   null,
    PAWN, PAWN,   PAWN,   PAWN,  PAWN, PAWN,   PAWN,   PAWN,
    ROOK, KNIGHT, BISHOP, QUEEN, KING, BISHOP, KNIGHT, ROOK
];
```

Die Figuren können hierbei verschieden abgebildet werden beispielsweise mit einem Zeichen, wie in der *FEN*, mit kleingeschriebenen Buchstaben für Schwarz und grossgeschriebenen Buchstaben für Weiss.

5.4.1.2 Bitboard Repräsentation

Ein Bitboard, auch als Bitset bezeichnet, ist in der Computertechnik ein Array von einzelnen Bits. Im Kontext der Schachprogrammierung ermöglicht diese Datenstruktur eine effiziente Darstellung und Manipulation der Position von Schachfiguren auf dem Brett. [34] Ein typisches Schachbrett hat 64 Felder, sodass ein 64-Bit-Integer verwendet werden kann, um das gesamte Brett darzustellen.

In einem Bitboard wird jedes Bit eines 64-Bit-Integers einem Feld auf dem Schachbrett zugeordnet. Ein gesetztes Bit an einer bestimmten Position zeigt an, dass ein bestimmtes Merkmal – z.B. das Vorhandensein einer Figur – auf dem zugehörigen Schachfeld vorhanden ist, während ein nicht gesetztes Bit das Gegenteil anzeigt. Zum Beispiel könnte man ein Bitboard verwenden, um alle Felder zu markieren, auf denen Bauern einer bestimmten Farbe stehen. Anders als bei der Mailbox, können aber auch andere Merkmale in einem Bitboard abgebildet werden, beispielsweise welche Felder von einem bestimmten Läufer angegriffen werden, oder auf welche Felder sich ein Springer bewegen kann.

Folgendes Bitboard stellt beispielsweise die Positionen aller weisser Bauern in der Standardposition dar (formatiert für Lesbarkeit mit Bezeichnung der Reihen und Linien):

```
8    0 0 0 0 0 0 0 0
7    0 0 0 0 0 0 0 0
6    0 0 0 0 0 0 0 0
5    0 0 0 0 0 0 0 0
4    0 0 0 0 0 0 0 0
3    0 0 0 0 0 0 0 0
2    1 1 1 1 1 1 1 1
1    0 0 0 0 0 0 0 0

     A B C D E F G H
```

Da es sich bei einem Bitboard nur aus einem Array von Bits handelt, kann es auch als Integer Zahl dargestellt werden. Obiges Bitboard als Integer:

```
71776119061217280
```

Mit Bitboards können nun sehr effiziente Berechnungen mittels bitweiser Operationen gemacht werden. Die Bits im Set können mit AND, OR, XOR, etc. einfach und effizient manipuliert werden. Dieses Vorgehen ist weniger intuitiv als die Mailbox Variante aber hat zum Vorteil, dass sie sehr effizient ist. Diese bitweisen Berechnungen sind bei der Zuggenerierung äusserst hilfreich.

Ein Beispiel dafür:
Das linke Bitboard markiert alle Felder, die von einem weissen Springer auf dem Feld C5 angegriffen werden (Die Position des Springers ist mit einem Asterisk markiert, zur Veranschaulichung). Das rechte Bitboard, markiert alle Felder, auf denen sich weisse Figuren befinden. Auf diese Felder kann der Springer nicht ziehen.

```
8    0 0 0 0 0 0 0 0
7    0 1 0 1 0 0 0 0
6    1 0 0 0 1 0 0 0
5    0 0 * 0 0 0 0 0
4    1 0 0 0 1 0 0 0
3    0 1 0 1 0 0 0 0
2    0 0 0 0 0 0 0 0
1    0 0 0 0 0 0 0 0

     A B C D E F G H
```

```
8    0 0 0 0 0 0 0 0
7    0 0 0 0 0 0 0 0
6    0 0 0 0 0 0 0 0
5    0 0 0 0 0 0 0 0
4    0 0 0 0 1 0 0 0
3    0 1 0 0 0 0 0 0
2    1 0 1 1 0 1 1 1
1    1 0 1 1 1 1 1 1

     A B C D E F G H
```

Mit einer AND und einer NOT-Operation kann nun herausgefunden werden, auf welche Felder der Springer ziehen darf. Zuerst wird das rechte Bitboard mit der NOT-Operation negiert, sodass alle Bits, die aktuell auf 1 sind auf 0 sind und umgekehrt. Danach wird die AND-Operation mit beiden Bitboards angewendet. Das Resultat ist ein Bitboard, in welchem nur die Bits gesetzt sind, die **nicht** im rechten Bitboard vorhanden sind **und** im linken Bitboard gesetzt sind.

Operation:

```
KnightAttacks & ~WhitePieces
```

Resultat:

```
8    0 0 0 0 0 0 0 0
7    0 1 0 1 0 0 0 0
6    1 0 0 0 1 0 0 0
5    0 0 * 0 0 0 0 0
4    1 0 0 0 0 0 0 0
3    0 0 0 1 0 0 0 0
2    0 0 0 0 0 0 0 0
1    0 0 0 0 0 0 0 0

     A B C D E F G H
```

5.4.1.3 Entscheidung

Beide Varianten werden nun gegeneinander abgewogen und aus den Resultaten eine Entscheidung gefällt.

	Bitboards	**Mailbox**
Vorteile	Performance	Tiefe Komplexität
Nachteile	Hohe Komplexität	Performance

Anstatt sich für eine einzelne Methode zu entscheiden können auch beide Methoden kombiniert werden. Der Autor hat sich für folgende Lösung entschieden:

Es wird eine Mailbox 8x8 Repräsentation gespeichert, um schnell herausfinden zu können, wo sich eine Figur auf dem Brett befindet. Das kann beispielsweise angewendet werden, um Figuren zu verschieben oder das Schlagen von Figuren einfach abzubilden. Anstatt nach einem Zug mehrere

Bitboards aktualisieren zu müssen, kann einfach die Mailbox bearbeitet werden. Die Bitboards können dann anhand der Mailbox neu generiert werden. Die Mailbox enthält die Information, wo sich jede Figur aufhält und es ist somit simpel Bitboards zu generieren. Die Bitboards wiederum, werden verwendet, um die legalen Züge mit hoher Leistung zu generieren.

Während Bitboards überlegene Leistung und Effizienz bieten, ist die Mailbox einfach und unkompliziert. Durch die Verwendung eines hybriden Ansatzes kann ein ausgewogener Kompromiss gefunden werden, der die Stärken beider Strukturen nutzt. Diese Methode stellt sicher, dass der Zuggenerator nicht nur effizient und schnell, sondern auch überschaubar und einfacher zu debuggen und zu warten ist. Es ist ein Beweis für das Sprichwort, dass in der Programmierung und im Design Flexibilität und Anpassungsfähigkeit oft zu den robustesten Lösungen führen. [35] [36]

5.4.2 Wer ist am Zug?

Die Datenstruktur, um zu speichern, wer aktuell am Zug ist, ist weniger kompliziert als die Wahl der Position der Figuren im vorherigen Kapitel. Eine Möglichkeit ist es, die aktive Farbe mit einer Zeichenkette, beispielsweise *white* oder *black* zu signalisieren. Der Autor hat sich allerdings entschieden Integer-Werte zu verwenden: 0 für Weiss und 1 für Schwarz. Einer der Vorteile davon über die Zeichenkette besteht darin, dass mit einer XOR-Operation, die gegnerische Farbe herausgefunden werden kann, anstelle eines If-Statements. Beispiel (weiss ist am Zug):

```
activeColor ^ 1 = 1
```

Ein numerisches Enum bietet sich hierfür sehr gut an. Die Definition des Enums *Color* im Programmcode wird in einem späteren Kapitel ausgewiesen.

5.4.3 Rochaderechte

Für die Rochaderechte bietet sich erneut an, ein Bitboard zu wählen. Dieses Bitboard besteht allerdings nur aus vier Bits. Jedes Bit indiziert, ob die Rochade auf eine bestimmte Seite noch möglich ist. [37]
Wenn beide Farben auf beide Seiten rochieren dürfen, ist das Bitboard `1111`.
Das erste Bit indiziert, ob Weiss auf die Seite des Königs rochieren darf und das zweite tut dasselbe für die Seite der Dame. Dasselbe gilt für das dritte und vierte Bit, allerdings für Schwarz. Wenn eine Seite Rochaderechte verliert, wird das entsprechende Bit auf 0 gesetzt.

Es muss dazu gesagt werden, dass die Rochaderechte auch mit vier Boolean-Flags indiziert werden könnten. Hier ist keine detaillierte Analyse nötig, denn beide Optionen sind valid und haben wohl keine grosse Auswirkung auf die Performance.

5.4.4 En-Passant Feld

Das *En-Passant* Feld wird als numerischer Index (0 – 63) gespeichert. Das macht Zugriffe in Bitboards und Mailbox sehr einfach. Der Wert kann auch undefiniert sein, für den Fall, dass *En-Passant* nicht möglich ist.

5.4.5 Half-Move-Clock

Die *Half-Move-Clock* ist ein einfacher Zähler und es kann somit ein Integer verwendet werden, der jedes Mal, wenn ein Zug gemacht wird inkrementiert wird, sofern es kein Bauer Zug ist und keine Figur geschlagen wurde. In diesem Fall würde der Zähler zurückgesetzt werden.

5.4.6 Anzahl Zyklen

Die Anzahl Zyklen ist ebenfalls ein einfacher Zähler und wird jedes Mal inkrementiert, wenn weiss einen Zug macht.

5.5 Implementation Schnittstelle

Anhand der erarbeiteten Informationen kann die Schnittstelle nun programmiert werden.
Folgende Komponenten werden benötigt:

- Eine Funktion, die die *FEN* in die Brettrepräsentation übersetzt
- Die *ChessBoard* Klasse

5.5.1 ChessBoard Klasse

In der *ChessBoard* Klasse sollen alle Informationen aus der *FEN* als Klassenattribute gespeichert werden, damit sie anschliessend verarbeitet werden können.

Es muss gespeichert werden, wo sich die Figuren auf dem Brett befinden. Dafür wird wie im vorherigen Kapitel besprochen die Mailbox verwendet. Eine Funktion nimmt die *FEN* entgegen und gibt die Brett Repräsentation als Objekt zurück. Dann werden alle Werte als Klasseneigenschaften gespeichert.

Folgend die rudimentäre Implementation der *ChessBoard* Klasse. Zum Verständnis wurden die genutzten Typendefinitionen in TypeScript hinzugefügt. Dazu gehört das Enum *Color*, sowie *BoardSquares*, (Typ für die Mailbox, bestehend aus einem 64-Elemente Array von Buchstaben, die die Figuren identifizieren oder null, falls das Feld leer ist). Die *Move* Klasse, die im vorherigen Kapitel «Schnittstellendesign» geplant wurde, wird in einem späteren Abschnitt behandelt, wenn es darum geht die Züge zu generieren. Aktuell ist die *Move* Klasse als leere Klasse zu betrachten, die nichts macht.

```
export class ChessBoard {
  #activeColor: Color;
  #boardSquares: BoardSquares;
  #legalMoves: Array<Move> = [];
  #halfMoveClock: number;
  #fullMoveNumber: number;
  #enPassantSquare: number | null;
  #castlingRights: number;

  constructor(fen: string) {
    const boardState = parseFEN(fen);
    this.#activeColor = boardState.activeColor;
    this.#boardSquares = boardState.boardSquares;
    this.#castlingRights = boardState.castlingAvailability;
    this.#enPassantSquare = boardState.enPassantSquare;
    this.#halfMoveClock = boardState.halfMoveClock;
    this.#fullMoveNumber = boardState.fullMoveNumber;

    this.generateMoves();
  }

  public static fromInitialPosition() {
    return new ChessBoard(STARTING_FEN);
  }

  public makeMove(move: Move) {
    // TODO
  }
```

```
  private generateMoves(color = this.activeColor) {
    // TODO
  }
}

export type PieceString = "P" | "K" | "N" | "R" | "B" | "Q";
export type ColoredPieceString = PieceString | Lowercase<PieceString>;
export type BoardSquares = Array<ColoredPieceString | null>;

export enum Color {
  White, // 0
  Black, // 1
}
```

Für private Eigenschaften wurde statt dem TypeScript Schlüsselwort *private* ein Hash-Zeichen verwendet. Das Hash-Zeichen ist die JavaScript Version von privaten Eigenschaften, gem. ECMAScript Spezifikation. Grund dafür ist, dass das TypeScript Schlüsselwort *private* nicht vor Zugriffen von aussen schützt, im Gegensatz zur JavaScript Version. [38]

Der Programmcode definiert die *ChessBoard* Klasse, mit Konstruktor und den absolut Nötigsten Funktionen. Die Getter für die Eigenschaften wurden entfernt, aus Gründen der Ausführlichkeit.

5.6 Errechnen legaler Züge

Mit der in den vorherigen Kapiteln erarbeiten Ausgangslage, kann nun damit begonnen werden, die Züge für die Figuren auf dem Brett zu errechnen.

5.6.1 Hilfsmittel

In den folgenden Kapiteln wurde zur Inspiration und zum Verständnis der öffentliche Quellcode einer didaktischen Schachengine, die auf Bitboards basiert, als Referenz verwendet [39]. Während die Engine in C verfasst ist, wurde der in diesem Projekt präsentierte Code eigenständig in TypeScript geschrieben und angepasst, wobei einige Konzepte und Algorithmen aus der Quelle inspiriert wurden.

5.6.2 Move Klasse

Als Voraussetzung für die Generierung von Zügen muss eine Datenstruktur für die Züge her, also macht es Sinn da anzufangen. Die Datenstruktur muss folgende Informationen beinhalten können [39]:

- Die Figur, die gezogen wird
- Das Ursprungsfeld
- Das Zielfeld
- Ob der Zug eine Beförderung ist, und wenn ja, zu was für eine Figur
- Ob mit dem Zug eine Figur geschlagen wird
- Ob der Zug ein Bauer ist, der zwei Schritte nach vorne zieht (Vorbedingung für *En-Passant* Regel)
- Ob der Zug *En-Passant* ist
- Ob der Zug eine *Rochade* ist

Alle Informationen, abgesehen vom Ursprungsfeld und dem Zielfeld könnten wohl weggelassen werden, da sie von der Position abgeleitet werden können. Es gibt niemals zwei oder mehr legale Züge die exakt das gleiche Ursprungs- und Zielfeld haben. Allerdings kann das Speichern dieser

Informationen in einer Datenstruktur den Programmcode beim eigentlichen Ausführen des Zuges vereinfachen und effizienter, übersichtlicher und lesbarer gestalten.

Der Autor hat sich hier im Kapitel «Schnittstellendesign» für eine Klasse entschieden, die alle diese Informationen beinhaltet:

```
export class Move {
  constructor(
    public sourceSquare: number,
    public targetSquare: number,
    public piece: ColoredPieceString,
    public promotionPiece: PieceString | null = null,
    public capture: boolean = false,
    public doublePawnPush: boolean = false,
    public enPassant: boolean = false,
    public castling: boolean = false
  ) {}
}
```

Das Ursprungsfeld und das Zielfeld sind jeweils der Index des Feldes in der Mailbox oder einem Bitboard (0-63). Mit diesen Eigenschaften ist die Zugklasse schon fast komplett, allerdings nicht wirklich für Menschen lesbar.

5.6.2.1 SAN

Um Züge für Benutzer der Schachplattform darzustellen, macht es Sinn sie in ein Format umzuwandeln, das für Menschen lesbar ist. Im kompetitiven Schachspiel an einem physischen Brett ist es üblich, dass beide Spieler jeden Zug auf einem Notizblock aufschreiben. Dafür wird die Algebraische Notation, spezifisch die *Standard Algebraische Notation (SAN)* verwendet. Die *SAN* ist ein kompaktes und standardisiertes Format. Sie wird breit genutzt in Schachliteratur, Turnieren und Schachprogrammen. [40]

Die *SAN* stellt sich aus folgenden Elementen zusammen, welche ebenfalls die Anforderungen für die Implementation der *Move* Klasse ergeben:

1. Figur Notation:
    - König (K): Bezeichnet mit "K".
    - Dame (Q): Bezeichnet mit "Q".
    - Turm (R): Bezeichnet mit "R".
    - Läufer (B): Bezeichnet mit "B".
    - Springer (N): Bezeichnet mit "N".
    - Bauer: Keine Bezeichnung.
2. Zugschreibweise:
    - Reguläre Züge: Das Zielfeld der Figur, z.B. *e4*, *d5*.
    - Bauern-Schläge: Das Feld, aus dem der Bauer schlägt, gefolgt von *x* und dem Zielfeld, z.B. *dxe5*.
    - Figuren-Schläge: Die Notation der Figur, gefolgt von *x* und dem Zielfeld, z. B. *Nxe5*.
3. Besondere Züge:
    - Rochade: *O-O* für die Rochade auf der Königsseite, *O-O-O* für die Rochade auf der Damenseite.
    - En-Passant: Es wird *e.p.* angehängt (optional).
    - Beförderung: Das Zielfeld, gefolgt von einem Gleichheitszeichen und der Figur, zu der der Bauer befördert wird, z.B. *e8=Q*.
4. Schach und Schachmatt:
    - Schach: Es wird ein + angehängt (optional)

- Schachmatt: es wird ein # angehängt (optional)

5. Zweideutigkeiten (Ambiguation):
   - Wenn zwei (oder mehr) identische Figuren auf dasselbe Feld ziehen können, enthält die Zugnotation auch die Linie (Buchstabe) oder die Reihe (Nummer) des Ausgangsfeldes, oder beides, falls erforderlich.
     Wenn z.B. zwei Türme auf das Feld *e1* ziehen können, einer von *a1* und ein anderer von *f1*, werden ihre Züge als *Rae1* oder *Rfe1* bezeichnet.

Beispiele für SAN-Notationen:
e4: Ein Bauer zieht nach *e4*.
Nf3: Ein Springer zieht nach *f3*.
Bxe5: Ein Läufer schlägt eine Figur auf *e5*.
O-O: Rochade auf der Königsseite.
dxe8Q: Ein Bauer auf der d-Linie schlägt die Figur auf *e8* und wird zur Dame.

5.6.2.2 Implementation toString() Methode der Zug-Klasse

Anhand obiger Informationen kann eine *toString()* Methode in die Zug-Klasse eingebaut werden. Der Autor hat davon abgesehen, Schach und Schachmatt Notationen einzufügen, da dies nicht speziell notwendig ist und um Aufwand zu sparen. Die folgende Funktion implementiert die SAN, gemäss obigen Spezifikationen. Helferfunktionen werden nicht speziell aufgeführt.

```
public toString(): string {
    if (this.isKingSideCastle()) {
      return "O-O";
    }

    if (this.isQueenSideCastle()) {
      return "O-O-O";
    }

    const target = getRankFile(this.targetSquare);

    const targetRank = 8 - target.rank;
    const targetFile = String.fromCharCode(97 + target.file);

    const piece =
      this.piece.toUpperCase() === "P" ? "" : this.piece.toUpperCase();
    const promotion = this.promotionPiece
      ? "=" + this.promotionPiece.toUpperCase()
      : "";
    const capture = this.capture ? "x" : "";
    const enPassant = this.enPassant ? "e.p." : "";

    const disambiguation = this.getDisambiguation();

    return
`${piece}${disambiguation}${capture}${targetFile}${targetRank}${promotion}${enPassa
nt}`;
}
```

5.6.3 Helfer Funktionen

In einem vorherigen Kapitel wurden bitweise Operationen bereits kurz behandelt. In den folgenden Abschnitten werden diese intensiv angewendet, um Bitboards zu manipulieren. Deshalb macht es Sinn einen Überblick über die Operationen zu verschaffen und wie sie angewendet werden können.

Im Folgenden wird vermehrt der Typ *Bitboard* verwendet. Das ist nur ein Alias für den Typen *bigint* in JavaScript. Der *number* Type in JavaScript kann zwar 64 Bits als floating point Zahlen repräsentieren, aber bei bitweisen Operationen, werden die Zahlen zu 32 Bit Integers umgewandelt. Deshalb muss der Datentyp *bigint* verwendet werden, der dafür da ist grössere Zahlen zu repräsentieren als *number*. [41] [42]
Aliase wie dieser machen den Programmcode lesbarer.

```
export type Bitboard = bigint;
```

Für die Operationen wurden Helfer-Funktionen geschrieben, die den Programmcode lesbarer machen.

5.6.3.1 AND

Die AND-Operation setzt alle Bits auf 1, die in beiden Bitboards auf 1 gesetzt sind und alle anderen auf 0.

```
export const and = (bitboard1: Bitboard, bitboard2: Bitboard) =>
  bitboard1 & bitboard2;
```

Beispiel: `1011 & 1101 = 1001`

5.6.3.2 NOT

Die NOT-Operation kehrt alle Bits um.

```
export const not = (bitboard: Bitboard) => ~bitboard;
```

Beispiel: `~1011 = 0100`

5.6.3.3 OR

Die OR-Operation setzt alle Bits auf 1, die entweder im linken oder im rechten Operanden auf 1 gesetzt sind.

```
export const or = (bitboard1: Bitboard, bitboard2: Bitboard) =>
  bitboard1 | bitboard2;
```

Beispiel: `1000 | 0101 = 1101`

5.6.3.4 XOR

Die XOR-Operation setzt Bits auf 1, die in genau einem der Operanden 1 sind und den Rest auf 0.

```
export const xor = (bitboard1: Bitboard, bitboard2: Bitboard) =>
  bitboard1 ^ bitboard2;
```

Beispiel: `0111 ^ 1011 = 1100`

5.6.3.5 LSB-Index

Die LSB-Index Operation sucht nach der Position des ersten gesetzten Bits von rechts.

```
export const lsb = (bitboard: Bitboard) => bitboard & -bitboard;
export const lsbIndex = (bitboard: Bitboard) =>
  Math.log2(Number(lsb(bitboard)));
```

Beispiel: `lsbIndex(1100) = 2`
In diesem Fall ist das erste gesetzte Bit von rechts an der dritten Stelle, also Index 2.

5.6.3.6 Left Shift

Die Left Shift Operation schiebt Bits nach links um eine bestimmte Anzahl.

```
export const shiftLeft = (bitboard: Bitboard, shift: number) =>
  bitboard << BigInt(shift);
```

Beispiel: `0010 << 5 = 001000000`

5.6.3.7 Right Shift

Die Right Shift Operation tut dasselbe wie die Left Shift Operation, nur schiebt sie die Bits nach rechts.

```
export const shiftRight = (bitboard: Bitboard, shift: number) =>
  bitboard >> BigInt(shift);
```

Beispiel: `1000 >> 2 = 10`

5.6.3.8 Bits setzen

Mit den Operationen, die vorhin definiert wurden, können Bits an einem spezifischen Index gesetzt, geholt oder gelöscht werden.

```
export const setBit = (bitboard: Bitboard, index: number) => {
  return bitboard | (1n << BigInt(index));
};

export const getBit = (bitboard: Bitboard, index: number) =>
  (bitboard >> BigInt(index)) & 1n;

export const clearBit = (bitboard: Bitboard, index: number) =>
  bitboard & ~(1n << BigInt(index));
```

5.6.4 Bitboards generieren

Um mit den Bitboards für Figuren zu arbeiten, müssen diese zuerst generiert werden. Wie in einem vorherigen Kapitel besprochen, werden die Bitboards anhand der Mailbox generiert. Dieser Prozess ist simpel und kann mit einfachen bitweisen Operationen durchgeführt werden:

```
export const bitboardForPiece = (
  boardSquares: BoardSquares,
  piece: ColoredPieceString
): Bitboard => {
  let bitboard = 0n;

  for (let i = 0; i < boardSquares.length; i++) {
    if (boardSquares[i] === piece) {
      bitboard |= shiftLeft(1n, i);
    }
  }
  return bitboard;
};
```

Die Funktion nimmt die Mailbox und die Bezeichnung der Figur, für welche ein Bitboard erstellt werden soll entgegen. Sie geht jeden Eintrag in der Mailbox durch und falls die Figur im jeweiligen Eintrag der Mailbox dieselbe ist, die der Funktion mitgegeben wurde, setzt sie das Bit im Bitboard am jeweiligen Index auf 1 mit einem Left-Shift und einer OR-Operation. Am Ende gibt die Funktion das Bitboard zurück.

Diese Funktion kann nun für jede Figur aufgerufen werden, um die Bitboards zu generieren. Folgende Funktion generiert die Bitboards für alle Figuren, weiss und schwarz.

```
export function generatePieceBitboards(
  boardSquares: BoardSquares
): BitboardsByPiece {
  const pieces: PieceString[] = ["P", "N", "B", "R", "Q", "K"];

  return pieces.reduce((acc, piece) => {
    const lowerCasePiece = piece.toLowerCase() as PieceString;
    acc[piece] = [
      bitboardForPiece(boardSquares, piece),
      bitboardForPiece(boardSquares, lowerCasePiece),
    ];
    return acc;
  }, {} as BitboardsByPiece);
}
```

Der Rückgabewert der Funktion ist ein Objekt. Der Schlüssel für den Eintrag im Objekt ist jeweils die Bezeichnung der Figur und der Wert ist ein Array mit zwei Einträgen. Der erste ist das Bitboard für die weissen Figuren und der zweite ist das Bitboard für die schwarzen Figuren. Folgend die Typendefinition von *BitboardsByPiece* in TypeScript.

```
export type PieceString = "P" | "K" | "N" | "R" | "B" | "Q";
export type BitboardsByPiece = Record<PieceString, [Bitboard, Bitboard]>;
```

Der Rückgabewert wird nun als Klasseneigenschaft in der *ChessBoard* Klasse gespeichert, sodass einfach darauf zugegriffen werden kann. Jedes Mal, wenn sich die Mailbox verändert, also wenn ein Zug gemacht wurde, kann die Funktion erneut aufgerufen werden, um die Bitboards zu aktualisieren.

5.6.5 Zuggenerierung Theorie

Mit den in den vorherigen Kapiteln besprochenen Ressourcen können nun die Züge generiert werden. Für das Verständnis hilft es, zuerst einen Einblick in die Theorie der Zuggenerierung zu verschaffen.

5.6.5.1 Typen von Figuren

Figuren müssen in der Schachprogrammierung anhand ihrer Bewegungen unterschieden werden. Es gibt grundsätzlich zwei verschiedene Muster für Bewegungen [43]:

- Nicht-schiebende Figuren
- Schiebende Figuren

Nicht-schiebende Figuren haben festgelegte Bewegungen. Sie sind in der Art ihrer Bewegung limitiert und können nicht endlos weit ziehen. Durch diese Limitation ist die Zuggenerierung für nicht-schiebende Figuren um einiges simpler als für schiebende Figuren. Für diese Figuren kann eine feste Liste von möglichen Zügen erstellt werden. Es muss nur überprüft werden, ob die Felder, auf die die Figur nach den Regeln ziehen darf, nicht besetzt sind. Nicht-schiebende Figuren sind Bauern, Springer und Könige.

Im Gegensatz dazu, sind schiebende Figuren etwas komplexer handzuhaben. Auf einem unendlich grossen Schachbrett könnten diese Figuren unendlich weit ziehen, sie haben also eine unbestimmte Anzahl an mögliche Züge. Wenn eine gegnerische Figur den Weg blockiert, kann die schiebende Figur diese schlagen und dort ihren Zug beenden. Wenn jedoch eine Figur des eigenen Teams den Weg blockiert, ist dieser Pfad blockiert und die Figur kann nicht weiterziehen. Diese Figuren sind Läufer, Turm und die Dame. Ein Turm am Rand des Brettes kann sich beispielsweise in einer

bestimmten Richtung nur ein Feld weit bewegen, wenn direkt neben ihm eine Figur steht, oder bis zum anderen Ende des Brettes, wenn der Weg frei ist.
Für schiebende Figuren muss in jede Richtung, in die sie ziehen kann, Feld für Feld überprüft werden, ob die Figur bis dahinziehen kann. Dieses Vorgehen ist äusserst rechenaufwändig. Glücklicherweise gibt es alternative Möglichkeiten, Züge für schiebende Figuren zu generieren, namentlich *Magische Bitboards*. Dies wird im Folgenden behandelt. [44]

5.6.5.2 Angriffstabellen und Angriffsmasken

Bevor wir mit den *Magischen Bitboards* weiterfahren, macht es Sinn über Angriffstabellen und Angriffsmasken zu reden. Eine Angriffsmaske ist nichts anderes als ein Bitboard, das Bits / Felder auf 1 gesetzt hat, welche von einer Figur an einer bestimmten Position angegriffen werden. In einem vorherigen Kapitel wurde bereits eine Angriffsmaske gezeigt, nämlich von einem Springer auf dem Feld C5. Hier nochmal zur Verständlichkeit. [45]

```
8    0 0 0 0 0 0 0 0
7    0 1 0 1 0 0 0 0
6    1 0 0 0 1 0 0 0
5    0 0 * 0 0 0 0 0
4    1 0 0 0 1 0 0 0
3    0 1 0 1 0 0 0 0
2    0 0 0 0 0 0 0 0
1    0 0 0 0 0 0 0 0

     A B C D E F G H
```

Angriffsmasken können für alle Figuren generiert werden.
Beispiel Angriffsmaske für einen Bauern auf E2.

```
8    0 0 0 0 0 0 0 0
7    0 0 0 0 0 0 0 0
6    0 0 0 0 0 0 0 0
5    0 0 0 0 0 0 0 0
4    0 0 0 0 0 0 0 0
3    0 0 0 1 0 1 0 0
2    0 0 0 0 * 0 0 0
1    0 0 0 0 0 0 0 0

     A B C D E F G H
```

Für nicht-schiebende Figuren können aus Angriffsmasken Angriffstabellen erstellt werden, da die Angriffe für jedes Feld immer gleich sind, unabhängig von der Brettaufstellung. Das heisst, dass für jedes Feld eine Angriffsmaske erstellt wird und in einem Array mit 64 Einträgen gespeichert wird. Diese Angriffstabelle kann dann für jedes beliebige Feld eingesetzt werden, um die Angriffe einer nicht-schiebenden Figur herauszufinden. Es muss gesagt sein, dass für Bauern zwei Angriffstabellen erstellt werden müssen, einmal für Schwarz und einmal für Weiss (da Bauern nur in eine Richtung schlagen können, abhängig von der Farbe). Angriffstabellen können im Voraus generiert werden, da sie wie gesagt unabhängig von der Brettaufstellung sind. Somit kann Rechenzeit während der eigentlichen Zuggenerierung gespart werden, mit dem Nachteil, dass mehr Arbeitsspeicher verbraucht wird. Angriffstabellen für nicht-schiebende Figuren sind allerdings sehr klein. Der Arbeitsspeicherbedarf kann ausgerechnet werden (ohne Overhead der Datenstrukturen).

$$7\ Figuren * 64\ Felder * 64\ Bit = 3.584\ Kilobyte$$

5.6.5.3 Magic Bitboards

Für schiebende Figuren ist das Vorgehen nicht so einfach, wie im vorherigen Abschnitt bereits angedeutet wurde. Welche Felder eine schiebende Figur angreift, ist abhängig von der aktuellen Brettaufstellung. Hier kommen Magische Bitboards ins Spiel.

Die Herausforderung besteht darin, Züge für eine schiebende Figur mit unterschiedlichen Besetzungen zu generieren. Eine Besetzungskonfiguration stellt dar, welche Felder auf dem Schachbrett von anderen Figuren besetzt sind. Da sich die Besetzung im Laufe des Spiels ändern kann, müssen wir einen Weg finden, jede Besetzung effizient auf das entsprechende Angriffsbitboard abzubilden.

Dafür werden magische Zahlen vorgeneriert. Magische Zahlen sind vorberechnete Werte, die als Brücke zwischen Besetzungskonfigurationen und Angriffsbitboards dienen. Sie ermöglichen es uns, eine gegebene Besetzung auf ein eindeutiges Angriffsbitboard mit einer einfachen und schnellen Operation abzubilden. Das Ziel ist es schlussendlich, eine vorberechnete Angriffstabelle zu haben, die pro Feld Angriffsbitboards für jede mögliche Besetzung auf dem Brett enthält, indexiert von einem Hash, der mit der magischen Zahl generiert wurde. Mithilfe der magischen Zahl und der tatsächlichen Brettbesetzung im Spiel kann dann auf das zutreffende Angriffsbitboard nachgeschlagen werden. Dieser Vorgang ermöglicht eine äusserst schnelle und effiziente Zuggenerierung für schiebende Figuren. Der Algorithmus, um magische Zahlen zu finden basiert auf der Brute-Force Methode. [44] [46]

Die Angriffstabellen für Läufer und Türme werden bei dieser Methode sehr gross. Für Läufer beispielsweise gibt es für jedes Feld eine maximale Anzahl von 512 verschiedenen Besetzungskonfigurationen. Wenn ein Läufer auf dem Feld C5 steht, gibt es 9 Felder, auf welchen eine Figur stehen könnte, die den Läufer blockt. Figuren ganz am Rand werden dabei nicht gezählt, weil der Läufer ohnehin nicht über sie hinweg ziehen kann. Somit lassen sich alle Kombinationen der Besetzung ausrechnen. [46]

$$2^9 = 512$$

Für Türme ist der Wert noch höher. Es gibt eine maximale Anzahl von 12 Feldern, auf welchen eine blockierende Figur stehen könnte.

$$2^{12} = 4096$$

Somit lässt sich erneut der Arbeitsspeicherbedarf ausrechnen.

Für Läufer:
$512\ Kombinationen * 64\ Felder * 64\ Bit = \ 262.144\ Kilobyte$

Für Türme:
$4096\ Kombinationen * 64\ Felder * 64\ Bit = 2.097\ Megabyte$

Dazu kommt der Overhead der Datenstrukturen. Diese Werte sind aber auf moderner Recheninfrastruktur problemlos vertretbar. Erneut, wie bei den klassischen Angriffstabellen vom vorherigen Kapitel, wird hier ein Kompromiss zwischen Zeitkomplexität und Platzkomplexität gemacht. [47]

5.6.6 Nicht-schiebende Figuren

Zur Implementation werden zuerst die unkomplizierten Fälle, die der nicht-schiebenden Figuren behandelt.

5.6.6.1 Bauern

Wie im Theorieabschnitt beschrieben, müssen Angriffstabellen für Bauern erstellt werden. Die Angriffstabelle wird erstellt, wenn das Programm gestartet wird, ist danach immer zugänglich und wird nicht verändert.

```
export const maskPawnAttacks = (square: number, color: Color): Bitboard => {
  const pawnOccupancy = setBit(0n, square);
  let attacks = 0n;

  const { file, rank } = getRankFile(square);
  if (color === Color.White) {
    if (file !== Files.A && rank !== Ranks.Eight) {
      attacks |= shiftRight(pawnOccupancy, 9); // up right
    }
    if (file !== Files.H && rank !== Ranks.Eight) {
      attacks |= shiftRight(pawnOccupancy, 7); // up left
    }
  } else {
    if (file !== Files.H && rank !== Ranks.One) {
      attacks |= shiftLeft(pawnOccupancy, 9); // down right
    }
    if (file !== Files.A && rank !== Ranks.One) {
      attacks |= shiftLeft(pawnOccupancy, 7); // down left
    }
  }

  return attacks;
};
```

Zu Beginn wird ein Bitboard namens *pawnOccupancy* erstellt, das nur ein einzelnes Bit gesetzt hat, nämlich das Bit, das dem Feld entspricht, auf dem sich der Bauer befindet. Ein weiteres leeres Bitboard namens *attacks* wird initialisiert, um die möglichen Angriffe zu speichern.

Je nach Farbe und Position des Bauern werden nun Bits in das Bitboard *attacks* gesetzt. Dazu wird das Bitboard mit der Position des Bauern nach links oder rechts verschoben, und das Ergebnis wird in das Bitboard *attacks* übertragen. Da ein Schachbrett 8 Felder in einer Reihe hat, kann durch das Verschieben des Bits um 7 oder 9 Felder die diagonale Zugrichtung erzeugt werden, in der Bauern schlagen können.

Diese Funktion kann nun für jedes der 64 Felder einmal aufgerufen werden und in einer Tabelle gespeichert werden:

```
export function initPawnAttackTable() {
  // Initialize two tables of 64 squares, one for each color
  const table: [Bitboard[], Bitboard[]] = [
    new Array(64).fill(0n),
    new Array(64).fill(0n),
  ];

  for (let square = 0; square <= Squares.H1; square++) {
    table[Color.White][square] = maskPawnAttacks(square, Color.White);
    table[Color.Black][square] = maskPawnAttacks(square, Color.Black);
```

```
  }

  return table;
}
```

Mit dieser Voraussetzung können nun die Züge generiert werden:
In der *ChessBoard* Klasse wird eine neue Methode erstellt, die *generatePawnMoves* heisst.
Diese Methode nimmt die Farbe, für welche die Züge generiert werden sollen, entgegen.

```
private generatePawnMoves(color = this.activeColor): Move[] {
    const pseudoLegalPawnMoves: Move[] = [];
    const pawns = this.#bitboards.P[color];

    let pawn = pawns;
    while (pawn) {
        const fromSquare = lsbIndex(pawn);
        // Implementation hier
        pawn = clearBit(pawn, fromSquare);
    }
    return pseudoLegalPawnMoves;
}
```

In der Methode werden die Bauern, der respektiven Farbe aus den generierten Bitboards ausgelesen. Dann wird über das Bitboard iteriert. Solange das Bitboard wahr ist, also mindestens ein Bit beinhaltet, läuft die Schleife weiter. Mit *lsbIndex* wird der Index vom ersten Bit, also vom ersten Bauer, von rechts aus dem Bitboard geholt. Das ist das Feld, auf welchem der Bauer steht. Am Ende der Schleife wird das Bit im Bitboard gelöscht, sodass *lsbIndex* beim nächsten Aufruf den Index des nächsten Bauern zurückgibt.

Innerhalb der Schlaufe können nun die Züge generiert werden. Das Erstellen der *Move* Klasse wurde aus Gründen der Ausführlichkeit mit Kommentaren ersetzt.
Folgend werden die normalen Züge, also ohne Schlagen, für Bauern erzeugt. Dies beinhaltet die Beförderungszüge, einen Schritt nach vorne und zwei Schritte nach vorne. Der Programmcode beinhaltet einige Helfer Funktionen, die ebenfalls nicht ausgeführt werden. Sie machen hauptsächlich bitweise Operationen, um eine bestimmte Kondition festzustellen.

```
let targetSquare = fromSquare + 8 * direction;

if (!isSquareOccupied(targetSquare, this.#bitboards)) {
  if (isPawnOnPromotionRank(fromSquare, color)) {
    // Zug-Klassen erstellen für Beförderungszüge
    // Ein Zug für jede Figur, zu welcher befördert werden kann wird erstellt
  } else {
    // Ein Schritt vorwärts Zug
    if (isPawnOnStartingRank(fromSquare, color)) {
      targetSquare += 8 * direction;
      if (!isSquareOccupied(targetSquare, this.#bitboards)) {
        // Zwei Schritte vorwärts
      }
    }
  }
}
```

Als nächstes muss *En-Passant* behandelt werden. Voraussetzung ist, dass die *ChessBoard* Eigenschaft *enPassantSquare* gesetzt ist. Das Setzen dieser Eigenschaft wird in einem zukünftigen Kapitel behandelt.
Wenn die Eigenschaft gesetzt ist, kann mit bitweisen Operationen herausgefunden werden, ob der Bauer, für welchen die Züge generiert wird, dieses Feld angreift. Mittels der Angriffstabelle kann dies einfach herausgefunden werden.

```
if (
  this.enPassantSquare &&
  pawnAttackTables[this.activeColor][fromSquare] &
    shiftLeft(1n, this.enPassantSquare)
) {
  // En-Passant Zug Klasse erstellen
}
```

Als letztes müssen die normalen Angriffe erstellt werden. Es wird erneut die Schlaufen-Technik angewendet, um über alle Bits zu iterieren, die in der Angriffsmaske **und** in einem Bitboard, das alle gegnerischen Figuren beinhaltet, auf 1 gesetzt sind.

```
let attacks = and(
  pawnAttackTables[color][fromSquare],
  allPiecesBitboard(this.#bitboards, enemyColor)
);

while (attacks) {
  targetSquare = lsbIndex(attacks);

  if (isPawnOnPromotionRank(fromSquare, color)) {
    // Beförderungszüge erstellen
  } else {
    // Normales Schlagen Zug erstellen
  }
  attacks = clearBit(attacks, targetSquare);
}
```

Somit sind nun alle Bauernzüge generiert.

5.6.6.2 Springer

Die Züge für Springer zu generieren, verläuft nach ähnlichem Verfahren, ist aber simpler, da keine speziellen Züge für Springer existieren.
Erneut wird zuerst die Angriffsmaske und Tabelle erstellt.

```
export const maskKnightAttacks = (square: number): Bitboard => {
  const knightOccupancy = setBit(0n, square);
  let attacks = 0n;

  const { file, rank } = getRankFile(square);

  if (rank > Ranks.Seven && file !== Files.H)
    attacks |= shiftRight(knightOccupancy, 15); // up 2 right 1
  if (rank > Ranks.Seven && file !== Files.A)
```

```
    attacks |= shiftRight(knightOccupancy, 17); // up 2 left 1

  if (rank !== Ranks.Eight && file > Files.B)
    attacks |= shiftRight(knightOccupancy, 10); // up 1 left 2
  if (rank !== Ranks.Eight && file < Files.G)
    attacks |= shiftRight(knightOccupancy, 6); // up 1 right 2

  if (rank < Ranks.Two && file !== Files.A)
    attacks |= shiftLeft(knightOccupancy, 15); // down 2 left 1
  if (rank < Ranks.Two && file !== Files.H)
    attacks |= shiftLeft(knightOccupancy, 17); // down 2 right 1

  if (rank !== Ranks.One && file < Files.G)
    attacks |= shiftLeft(knightOccupancy, 10); // down 1 right 2
  if (rank !== Ranks.One && file > Files.B)
    attacks |= shiftLeft(knightOccupancy, 6); // down 1 left 2

  return attacks;
};
```

Folgend können nun die Züge in der *ChessBoard* Klasse generiert werden. Hierbei muss nur beachtet werden, dass Springer nicht auf Felder ziehen können, die von einer Figur der gleichen Farbe besetzt ist.

```
private generateKnightMoves(color = this.activeColor): Move[] {
    const pseudoLegalKnightMoves: Move[] = [];
    const enemyColor = color ^ 1;

    let knights = this.#bitboards.N[color];
    while (knights) {
      const fromSquare = lsbIndex(knights);
      const potentialAttacks = knightAttackTable[fromSquare];

      let legalAttacks = and(
        potentialAttacks,
        not(allPiecesBitboard(this.#bitboards, color))
      );

      while (legalAttacks) {
        const toSquare = lsbIndex(legalAttacks);

        // Zug erstellen.
        legalAttacks = clearBit(legalAttacks, toSquare);
      }
      knights = clearBit(knights, fromSquare);
    }

    return pseudoLegalKnightMoves;
}
```

5.6.6.3 König

Für den König dasselbe Vorgehen - Angriffsmaske und Tabelle:

```
export const maskKingAttacks = (square: number): Bitboard => {
  const kingOccupancy = setBit(0n, square);
  let attacks = 0n;

  const { file, rank } = getRankFile(square);

  // horizontal and vertical attacks
  if (rank !== Ranks.Eight) attacks |= shiftRight(kingOccupancy, 8); // up
  if (rank !== Ranks.One) attacks |= shiftLeft(kingOccupancy, 8); // down
  if (file !== Files.H) attacks |= shiftLeft(kingOccupancy, 1); // right
  if (file !== Files.A) attacks |= shiftRight(kingOccupancy, 1); // left

  // diagonal attacks
  if (rank !== Ranks.Eight && file !== Files.H)
    attacks |= shiftRight(kingOccupancy, 7); // up right
  if (rank !== Ranks.Eight && file !== Files.A)
    attacks |= shiftRight(kingOccupancy, 9); // up left
  if (rank !== Ranks.One && file !== Files.H)
    attacks |= shiftLeft(kingOccupancy, 9); // down right
  if (rank !== Ranks.One && file !== Files.A)
    attacks |= shiftLeft(kingOccupancy, 7); // down left

  return attacks;
};
```

Für die Generierung der Züge in der *ChessBoard* Klasse müssen zwei Punkte speziell beachtet werden. Zum einen die Rochade und zum anderen, dass der König nicht in den Umkreis des gegnerischen Königs zieht.

```
private generateKingMoves(color = this.activeColor) {
    const pseudoLegalKingMoves: Move[] = [];
    const enemyColor = color ^ 1;

    const king = this.#bitboards.K[color];
    const fromSquare = lsbIndex(king);

    const potentialAttacks = kingAttackTable[fromSquare];

    const enemyKing = lsbIndex(this.#bitboards.K[enemyColor]);
    const enemyKingRadius = kingAttackTable[enemyKing];

    let legalAttacks = and(
      potentialAttacks,
      not(allPiecesBitboard(this.#bitboards, color)),
      not(enemyKingRadius)
    );

    while (legalAttacks) {
      const targetSquare = lsbIndex(legalAttacks);
```

```
        const isCapture = !!getBit(
          allPiecesBitboard(this.#bitboards, enemyColor),
          targetSquare
        );

        // Zug erstellen

        legalAttacks = clearBit(legalAttacks, targetSquare);
      }
}
```

Bei den legalen Angriffen wird der Radius des gegnerischen Königs herausgefiltert.

Die Rochadezüge müssen alle Bedingungen für die Rochade überprüfen:

- König steht auf dem richtigen Feld
- Turm steht auf dem richtigen Feld
- König zieht nicht durch oder in ein Schach
- König ist nicht im Schach
- Rochaderechte sind noch vorhanden
- Es steht keine Figur im Weg

```
if (
    color === Color.White &&
    this.castlingRights & CastlingAvailabily.WhiteKing &&
    this.boardSquares[Squares.E1] === "K" &&
    this.boardSquares[Squares.H1] === "R" &&
    !isSquareOccupied(Squares.F1, this.#bitboards) &&
    !isSquareOccupied(Squares.G1, this.#bitboards) &&
    !isSquareAttacked(Squares.E1, Color.Black, this.#bitboards) &&
    !isSquareAttacked(Squares.F1, Color.Black, this.#bitboards) &&
    !isSquareAttacked(Squares.G1, Color.Black, this.#bitboards)
) {
      // Königsseite Rochade Weiss Zug
}
```

Dies wird für alle vier Rochaden gemacht.
Somit sind nun alle Züge für nicht-schiebende Figuren generiert.

5.6.7 Schiebende Figuren

Nun zu den schiebenden Figuren. Dafür müssen, wie im Theorieabschnitt besprochen, zuerst magische Zahlen generiert werden. Es muss dazu gesagt werden, dass es im Internet viele Sets von vorgenerierten magischen Zahlen gibt, die verwendet werden könnten. Der Autor hat sich allerdings aus didaktischen Gründen dazu entschieden, die Zahlen selbst zu generieren.

Anschliessend werden die magischen Zahlen verwendet, um die Angriffstabellen zu erstellen und die Züge für Türme, Läufer und Damen zu berechnen.

5.6.7.1 Magic Number Generation

Das Vorgehen für die Generierung von magischen Zahlen verläuft wie folgt für jedes Feld auf dem Schachbrett:

1. Es wird eine Angriffsmaske generiert für Läufer und Türme, die Felder ganz am Rand des Bretts ignoriert, da die Figur nicht durch das Feld hindurch ziehen kann.
2. Es werden die Bits in der Angriffsmaske gezählt (*relevantBits*). Diese Zahl wird verwendet, um die mögliche Anzahl von Permutationen (Besetzungskonfigurationen) für das jeweilige Feld zu errechnen.
3. Mit den *relevantBits* werden nun die Besetzungskonfigurationen generiert. Für jedes Feld wird jede mögliche Kombination von Blockern errechnet.
4. Für jede Besetzungskonfiguration wird ein Angriffsbitboard errechnet. Das Angriffsbitboard hat alle Bits gesetzt, auf die die Figur ziehen kann, bis sie geblockt wird, oder am Rand ankommt.
5. Anschliessend wird eine Schleife gestartet, die den Brute-Force Mechanismus übernimmt. Die Schleife läuft beispielsweise eine Million Mal.
6. In der Schleife wird ein Kandidat für die magische Zahl generiert. Das ist eine 64-Bit zufällige Ganzzahl.
7. Der generierte Kandidat wird dann für jede Besetzungskonfiguration getestet. Die Zahl wird mit der Besetzungskonfiguration multipliziert und ein Right-Shift angewendet, um den Hash-Index zu berechnen. Der Hash-Index ist eine Zahl zwischen 0 und *relevantBits*.
8. Die Angriffsmaske wird dann in einer Tabelle gespeichert, mit dem Hash-Index als Index.
9. Wenn die Tabelle bereits einen Eintrag hat, dann ist eine Kollision entstanden und der Kandidat ist ungültig und der nächste Kandidat wird getestet. Wenn nicht, wurde erfolgreich eine magische Zahl gefunden.

[44] [48]

```
export function findMagicNumber(square: number, bishop: boolean): bigint {
  let blockerConfigurations: Bitboard[] = [];
  let attacks: Bitboard[] = [];

  // Schritt 1
  const attackMask: Bitboard = bishop
    ? maskBishopAttacks(square)
    : maskRookAttacks(square);

  // Determine the hash index mask based on the piece type.
  const hashIndexMask = bishop ? 0x1ffn : 0xfffn;

  // Schritt 2
  const relevantBits = countTruthy(attackMask);
  const occupancyVariationsCount = 1 << relevantBits; // 2 ^ relevantBits

  // Schritt 3 und 4
  for (let index = 0; index < occupancyVariationsCount; index++) {
    const occupancy = generateOccupancyPermutation(
      index,
      relevantBits,
      attackMask
    );
    blockerConfigurations[index] = occupancy;
    attacks[index] = bishop
```

```
      ? bishopAttacks(square, occupancy)
      : rookAttacks(square, occupancy);
  }

  // Schritt 5
  for (let attempt = 0; attempt < 10000000; attempt++) {
    // Schritt 6
    const magicCandidate = generateMagicNumberCandidate();
    let usedAttacks: Bitboard[] = [];

    let fail: boolean = false;

    for (let index = 0; index < occupancyVariationsCount; index++) {
      // Schritt 7
      const hashIndex = Number(
        ((blockerConfigurations[index] * magicCandidate) >>
          BigInt(64 - relevantBits)) &
          hashIndexMask
      );

      // Schritt 8 und 9
      if (usedAttacks[hashIndex] === undefined) {
        usedAttacks[hashIndex] = attacks[index];
      } else if (usedAttacks[hashIndex] != attacks[index]) {
        fail = true;
        break;
      }
    }

    // Magische Zahl gefunden!.
    if (!fail) {
      return magicCandidate;
    }
  }
  // Falls nach 1 Million versuchen keine magische Zahl gefunden wurde.
  return 0n;
}
```

Aus Gründen der Ausführlichkeit werden die Funktionen *generateOccupancyPermutation*, *bishopAttacks*, *rookAttacks*, *maskBishopAttacks* und *maskRookAttacks* nicht speziell aufgeführt.

Die Funktion kann nun für je Läufer und Türme und pro Feld aufgerufen werden, um die magischen Zahlen für jedes Feld zu errechnen. Die magischen Zahlen werden anschliessend in eine JSON-Datei gespeichert, sodass sie während der Laufzeit des Programms wieder verwendet werden können.

5.6.7.2 Angriffstabellen erstellen

Als nächster Schritt müssen die Angriffstabellen erstellt werden, die alle Besetzungskonfigurationen beinhalten. Dies wurde bei der Generierung der magischen Zahlen bereits gemacht, allerdings sind dieses nicht zur Verfügung während der Laufzeit.

Die Funktionalität ist noch einmal sehr ähnlich. Nun werden die generierten magischen Zahlen benutzt, um die Tabelle zu füllen:

```
export function initSliderAttacks(bishop: boolean) {
  const attackTable: bigint[][] = Array(64)
    .fill(null)
    .map(() => []);

  const hashIndexMask = bishop ? 0x1ffn : 0xfffn;

  for (let square = 0; square <= Squares.H1; square++) {
    const attackMask = bishop
      ? bishopAttackMasks[square]
      : rookAttackMasks[square];

    const relevantBits = countTruthy(attackMask);
    const attackCount = 1 << relevantBits;
    for (let i = 0; i < attackCount; i++) {
      const blockerConfiguration = generateOccupancyPermutation(
        i,
        relevantBits,
        attackMask
      );
      const hashIndex = Number(
        ((blockerConfiguration *
          (bishop ? bishopMagicNumbers[square] : rookMagicNumbers[square])) >>
          BigInt(64 - relevantBits)) &
          hashIndexMask
      );

      attackTable[square][hashIndex] = bishop
        ? bishopAttacks(square, blockerConfiguration)
        : rookAttacks(square, blockerConfiguration);
    }
  }

  return attackTable;
}
```

Die Funktion generiert pro Feld erneut alle möglichen Besetzungskonfigurationen, sowie den Hash-Index und speichert die Konfiguration in der Tabelle. Dies geschieht beim Start der Applikation und die Tabelle ist somit für das Programm verfügbar. [49]

5.6.7.3 Angriffsbitboards erstellen

Mittels der generierten Tabelle und den magischen Zahlen kann das Programm nun für jede beliebige Position äusserst effizient die Angriffe eines Läufers, Turms oder einer Dame errechnen.

```
function getSliderAttacks(
  square: number,
  board: Bitboard,
  bishop: boolean
): bigint {
  const magicNumber = bishop
    ? bishopMagicNumbers[square]
    : rookMagicNumbers[square];

  const attackMask = bishop
    ? bishopAttackMasks[square]
    : rookAttackMasks[square];

  const blockerConfiguration = and(board, attackMask);

  const relevantBits = countTruthy(
    bishop ? bishopAttackMasks[square] : rookAttackMasks[square]
  );

  const attackTable = bishop ? bishopAttackTable : rookAttackTable;

  const hashIndexMask = bishop ? 0x1ffn : 0xfffn;
  const hashIndex = and(
    shiftRight(blockerConfiguration * magicNumber, 64 - relevantBits),
    hashIndexMask
  );

  return attackTable[square][Number(hashIndex)];
}
```

Die Funktion sucht für das gegebene Feld die magische Zahl heraus und erstellt eine Besetzungskonfiguration, maskiert mit der Angriffsmaske der Figur von diesem Feld aus. Das Resultat ist ein Bitboard mit Bits gesetzt für alle Figuren, die die Figur blockieren. Mit dieser Besetzungskonfiguration kann nun erneut der Hash-Index mit der gleichen Methode errechnet werden und die Angriffe der Figur in der Angriffstabelle nachgeschlagen werden.
Folgend ein Beispiel. Es sollen die Angriffe für einen Läufer auf C3 errechnet werden.

Besetzung des Bretts:

```
8    0 0 0 0 0 0 0 0
7    0 0 0 0 0 0 0 0
6    0 1 1 0 0 1 0 1
5    0 0 1 1 0 0 0 0
4    0 0 0 0 1 0 0 0
3    0 0 * 0 0 0 0 0
2    1 1 0 1 0 1 0 1
1    0 0 0 0 0 0 0 0

     A B C D E F G H
```

Angriffsmaske des Läufers:

```
8    0 0 0 0 0 0 0 0
7    0 0 0 0 0 0 1 0
6    0 0 0 0 0 1 0 0
5    0 0 0 0 1 0 0 0
4    0 1 0 1 0 0 0 0
3    0 0 * 0 0 0 0 0
2    0 1 0 1 0 0 0 0
1    0 0 0 0 0 0 0 0

     A B C D E F G H
```

Besetzungskonfiguration (alle Bits, die in der Besetzung **und** in der Angriffsmaske auf 1 sind):

```
8    0 0 0 0 0 0 0 0
7    0 0 0 0 0 0 0 0
6    0 0 0 0 0 1 0 0
5    0 0 0 0 0 0 0 0
4    0 0 0 0 0 0 0 0
3    0 0 * 0 0 0 0 0
2    0 1 0 1 0 0 0 0
1    0 0 0 0 0 0 0 0

     A B C D E F G H
```

Mit dieser Besetzungskonfiguration kann nun der Hash-Index ausgerechnet werden, um in der Angriffstabelle nachzuschlagen.
Resultat Angriffstabelle:

```
8    0 0 0 0 0 0 0 0
7    0 0 0 0 0 0 0 0
6    0 0 0 0 0 1 0 0
5    1 0 0 0 1 0 0 0
4    0 1 0 1 0 0 0 0
3    0 0 * 0 0 0 0 0
2    0 1 0 1 0 0 0 0
1    0 0 0 0 0 0 0 0
     A B C D E F G H
```

Wie man sieht, sind alle Felder auf 1 gesetzt, auf die der Läufer ziehen oder schlagen kann.

5.6.7.4 Züge generieren

Mit der erarbeiteten Funktionalität können nun Züge für Läufer, Türme und Damen in der *ChessBoard* Klasse errechnet werden, ähnlich wie das auch schon bei nicht-schiebenden Figuren gemacht wurde.

```
private generateQueenMoves(color = this.activeColor) {
    const pseudoLegalQueenMoves: Move[] = [];
    const enemyColor = color ^ 1;

    let queens = this.#bitboards.Q[color];
    while (queens) {
      const fromSquare = lsbIndex(queens);
      const potentialAttacks = getQueenAttacks(
        fromSquare,
        allPiecesBitboard(this.#bitboards)
      );
      let legalAttacks = and(
        potentialAttacks,
        not(allPiecesBitboard(this.#bitboards, color))
      );

      while (legalAttacks) {
        const targetSquare = lsbIndex(legalAttacks);

        const isCapture = !!getBit(
          allPiecesBitboard(this.#bitboards, enemyColor),
          targetSquare
        );

        // Zug hinzufügen

        legalAttacks = clearBit(legalAttacks, targetSquare);
      }

      queens = clearBit(queens, fromSquare);
    }

    return pseudoLegalQueenMoves;
}
```

Dabei funktionieren die Figuren alle gleich. Für Läufer, Türme und Damen gibt es auch keine Spezialzüge. Das Angriffsbitboard wird mit der im vorherigen Kapitel beschriebenen Methode geholt und darüber iteriert. Dabei wird sichergestellt, dass Figuren derselben Farbe nicht im Angriffsbitboard enthalten sind.

Somit können nun alle Züge für alle Figuren generiert werden.

5.6.8 Zug ausführen

Was noch fehlt, ist die Möglichkeit generierte Züge auszuführen. Dafür wird die *makeMove* Methode in der *ChessBoard* Klasse implementiert. Da fast alle nötigen Informationen bereits in den generierten Zugklassen enthalten ist, ist dieses Unterfangen verhältnismässig simpel.

Es wird zuerst eine Kopie der Mailbox gemacht.

```
const boardCopy = this.boardSquares.slice();
```

Anschliessend werden die Figuren in der Kopie verschoben:

```
const sourcePiece = boardCopy[move.sourceSquare];
boardCopy[move.sourceSquare] = null;
boardCopy[move.targetSquare] = sourcePiece;
```

Das Ursprungsfeld wird geleert und das Zielfeld wird mit der Figur gesetzt.
Wenn der Zug eine Beförderung ist, wird auf dem Zielfeld die beförderte Figur gesetzt.

```
if (move.promotionPiece) {
    const promotionPieceColor = this.activeColor
        ? move.promotionPiece.toLowerCase()
        : move.promotionPiece;
    boardCopy[move.targetSquare] = promotionPieceColor as ColoredPieceString;
}
```

Wenn der Zug ein doppelter Bauernzug ist, wird das *En-Passant* Feld gesetzt:

```
if (move.doublePawnPush) {
  const direction = this.activeColor ? -1 : 1;
  this.#enPassantSquare = move.targetSquare + 8 * direction;
} else {
  this.#enPassantSquare = null;
}
```

Wenn der Zug eine Rochade ist, werden die Figuren an die richtige Position gesetzt.

```
if (move.castling) {
  if (move.targetSquare === Squares.G1) {
    boardCopy[Squares.H1] = null;
    boardCopy[Squares.F1] = "R";
  }
  if (move.targetSquare === Squares.C1) {
    boardCopy[Squares.A1] = null;
    boardCopy[Squares.D1] = "R";
  }
  if (move.targetSquare === Squares.G8) {
    boardCopy[Squares.H8] = null;
    boardCopy[Squares.F8] = "r";
  }
  if (move.targetSquare === Squares.C8) {
    boardCopy[Squares.A8] = null;
    boardCopy[Squares.D8] = "r";
  }
}
```

Am Ende werden die Bitboards anhand der aktualisierten Mailbox neu generiert und es wird überprüft, ob der König im Schach steht. Falls ja, ist der Zug ungültig.

```
const newBitboards = generatePieceBitboards(boardCopy);
const kingPosition = lsbIndex(newBitboards.K[this.activeColor]);
if (isSquareAttacked(kingPosition, enemyColor, newBitboards)) {
```

```
  throw new IllegalMoveError(move);
}
```

Falls der Turm oder der König sich bewegt haben, werden die Rochaderechte den Umständen entsprechend entzogen.

```
if (move.piece.toUpperCase() === "K") {
  this.#castlingRights =
    this.castlingRights & (this.activeColor ? 0b1100 : 0b0011);
}

if (move.piece.toUpperCase() === "R") {
  if (move.sourceSquare === Squares.A1 && this.activeColor === Color.White) {
    this.#castlingRights &= ~CastlingAvailabily.WhiteQueen;
  }

  if (move.sourceSquare === Squares.H1 && this.activeColor === Color.White) {
    this.#castlingRights &= ~CastlingAvailabily.WhiteKing;
  }

  if (move.sourceSquare === Squares.A8 && this.activeColor === Color.Black) {
    this.#castlingRights &= ~CastlingAvailabily.BlackQueen;
  }

  if (move.sourceSquare === Squares.H8 && this.activeColor === Color.Black) {
    this.#castlingRights &= ~CastlingAvailabily.BlackKing;
  }
}
```

Nun bleibt nur noch die Änderungen in die Klasseneigenschaften zu schreiben und die Zähler (*Half-Move-Clock* und Zyklen) zu inkrementieren und anschliessend die Züge für die neue Position zu errechnen.

```
const newBitboards = generatePieceBitboards(boardCopy);

const kingPosition = lsbIndex(newBitboards.K[this.activeColor]);
if (isSquareAttacked(kingPosition, enemyColor, newBitboards)) {
  throw new IllegalMoveError(move);
}

this.#halfMoveClock++;
if (move.piece.toUpperCase() === "P" || move.capture) {
  this.#halfMoveClock = 0;
}

if (this.activeColor === Color.White) {
  this.#fullMoveNumber = this.fullMoveNumber + 1;
}

this.#activeColor = this.activeColor ^ 1;
this.#boardSquares = boardCopy;
this.#bitboards = newBitboards;
```

```
this.generateMoves();
```

5.6.9 Pseudolegale Züge

Die Züge die aktuell generiert werden sind pseudolegal. Das heisst, es wird bei der Zuggenerierung nicht überprüft, ob der König nach dem Zug im Schach stehen würde, sondern erst bei der Ausführung der Züge. Das ist suboptimal, da der Zuggenerator illegale Züge zurückgibt. Eine einfache Möglichkeit das zu prüfen, ist es für jeden Zug, der generiert wird, die *makeMove* Funktion auszuführen. Diese wirft wie vorhin beschrieben einen Fehler, wenn der König danach im Schach steht. Diese Züge werden dann herausgefiltert. Dafür dürfen bei der Ausführung der *makeMove* Methode die Klasseneigenschaften nicht aktualisiert werden, denn es wird nur getestet, ob der Zug theoretisch möglich wäre.

Aus diesem Grund wird die Methode *makeMove* in *executeMove* umbenannt und nimmt einen neuen Paremeter *commit* entgegen, der bestimmt, ob die Änderungen an den Klasseneigenschaften vorgenommen werden sollen. Die Methode, die alle Züge generiert, ruft nun diese *executeMove* Methode auf, ohne Änderungen vorzunehmen.

```
private generateMoves(color = this.activeColor) {
    const pseudoLegalMoves = [
      ...this.generatePawnMoves(color),
      ...this.generateKnightMoves(color),
      ...this.generateBishopMoves(color),
      ...this.generateRookMoves(color),
      ...this.generateQueenMoves(color),
      ...this.generateKingMoves(color),
    ];

    this.#legalMoves = pseudoLegalMoves.filter((move) => {
      try {
        this.executeMove(move, false);
        return true;
      } catch (e) {
        if (e instanceof IllegalMoveError) {
          return false;
        }
        throw e;
      }
    });
}
```

Somit produziert der Zuggenerator nun nur noch legale Züge.

5.7 Schachmatt

Das Ende des Spiels durch Schachmatt zu bestimmen ist äusserst simpel. Es gibt zwei Konditionen für Schachmatt, die erfüllt werden müssen:

- König wird angegriffen
- Der Spieler hat keine legalen Züge

```
private detectCheckmate() {
    const enemyColor = this.activeColor ^ 1;
    return (
      this.legalMoves.length === 0 &&
      isSquareAttacked(
        lsbIndex(this.#bitboards.K[this.activeColor]),
        enemyColor,
        this.#bitboards
      )
    );
}
```

Diese Funktion wird nun in der *executeMove* Methode aufgerufen, nachdem ein Zug durchgeführt wurde. Wenn das Resultat der Funktion wahr ist, ist das Spiel zu Ende.

5.8 Remis

Die letzte Funktion, die noch fehlt, ist das Erkennen von Unentschieden. Wie in den Regeln erarbeitet, gibt es verschiedene Arten von Untentschieden. Dafür wird eine neue Methode *detectDraw* in der *ChessBoard* Klasse erstellt. Die Methode wird nach jedem Zug aufgerufen und gibt nicht nur zurück, ob das Spiel unentschieden ist, oder nicht, sondern auch aus welchem Grund das Unentschieden entstanden ist.

```
private detectDraw(): GameOutcome | false {
    // Implementation
}
```

Der Typ *GameOutcome* stellt sich ausfolgenden Typen zusammen:

```
export type GameOutcome = {
  result: GameResult;
  type: DrawType | EndGameType;
};

export enum GameResult {
  WhiteWins = "white_wins",
  BlackWins = "black_wins",
  Draw = "draw",
}

export enum DrawType {
  Agreement = "agreement",
  Stalemate = "stalemate",
  DeadPosition = "dead_position",
  FiftyMoveRule = "fifty_move_rule",
  ThreefoldRepetition = "threefold_repetition",
```

```
}

export enum EndGameType {
  Checkmate = "checkmate",
}
```

5.8.1 Patt

Patt entsteht, wenn der König nicht im Schach steht, aber der Spieler keine legalen Züge hat und somit nicht ziehen kann. Dies ist ähnlich simpel zu erkennen, wie Schachmatt:

```
const kingPosition = lsbIndex(this.#bitboards.K[this.activeColor]);
if (
  this.legalMoves.length === 0 &&
  !isSquareAttacked(kingPosition, this.activeColor ^ 1, this.#bitboards)
) {
  return { result: GameResult.Draw, type: DrawType.Stalemate };
}
```

5.8.2 50 Zug Regel

Da in der *executeMove* Methode die *Half-Move-Clock* inkrementiert wird, muss für die Entscheidung, ob das Spiel ein Unentschieden nach der 50-Zug Regel ist, nur überprüft werden, ob der Zähler bei 50 angelangt ist.

```
if (this.halfMoveClock >= 50) {
  return { result: GameResult.Draw, type: DrawType.FiftyMoveRule };
}
```

5.8.3 Dreifachrepetition

Um die Konditionen für eine Dreifachrepetition zu erfüllen, muss die exakt gleiche Position drei Mal vorkommen, unabhängig von den Positionen dazwischen. Dazu gehören auch Rochaderechte, *En-Passant* und die Seite, die am Zug ist.

Dafür könnte eine Historie der gesamten Brettrepräsentation in der *ChessMove* Klasse gespeichert werden. Das ist aber nicht besonders speichereffizient. In der Schachprogrammierung hilft hier das Konzept von Zobrist-Hashes.

5.8.3.1 Zobrist-Hash

Die Kernidee ist, jeder potenziellen Spielsituation oder einem Teil davon eine einzigartige Zufallsnummer zuzuweisen. Mithilfe dieser Nummern wird ein Hash-Wert erstellt, der das gesamte Spielbrett widerspiegelt. Dieser Wert ermöglicht es, Daten zu dieser spezifischen Spielbrettkonfiguration zu speichern oder abzurufen. [50]

Für jede Kombination von Schachfigur und ihrer Platzierung auf dem Brett erstellen wir eine eindeutige Zufallsnummer. Es gibt 6 unterschiedliche Figurentypen für jede Farbe und 64 Felder. Daher wird eine Nummer für den weissen Bauern auf jedem Feld erstellt, dann für den schwarzen Bauern auf jedem Feld und so weiter für jede Figur.

Zusätzlich werden zwei Zahlen erzeugt, um zu bestimmen, welche Farbe am Zug ist. Für die *En-Passant*-Regel werden acht Nummern erstellt, eine für jede Reihe. Da die Regel nur für die Farbe gilt, die gerade zieht, benötigen wir keine getrennten Zahlen für beide Farben.

Abschliessend erstellen wir 16 Zahlen für die Rochaderechte, je eine für jede mögliche Kombination der Rechte. Da es 2^4 Möglichkeiten gibt, erhalten wir 16 Kombinationen.

Diese festgelegten Zahlen werden gespeichert und stehen dem Programm während seiner Ausführung zur Verfügung. Als Zufallszahlen werden 64 Bit Ganzzahlen verwendet, um Kollisionen zu verhindern.

```
const pieces = ["P", "N", "B", "R", "Q", "K", "p", "n", "b", "r", "q", "k"];

const pieceKeys: Record<string, bigint> = {};

for (let i = 0; i < 64; i++) {
  for (const piece of pieces) {
    const key = `${piece}${i}`;
    pieceKeys[key] = random64();
  }
}
const enPassantKeys: bigint[] = [];

for (let i = 0; i < 8; i++) {
  enPassantKeys[i] = random64();
}

const castlingKeys: bigint[] = [];

for (let i = 0; i < 16; i++) {
  castlingKeys[i] = random64();
}

const sideKeys = [random64(), random64()];
```

Mit den generierten Zahlen kann nun für jede beliebige Position ein eindeutiger Wert generiert werden, indem die zutreffenden Zufallszahlen mit XOR vermischt werden.

```
export function computeZobristHash(
  boardSquares: BoardSquares,
  castlingRights: number,
  enPassantSquare: number | null,
  side: Color
): bigint {
  let hash = 0n;

  for (let square = 0; square < 64; square++) {
    const piece = boardSquares[square];
    if (!piece) continue;
    hash = xor(hash, zobristKeys.pieces[`${piece}${square}`]);
  }

  if (enPassantSquare) {
    const { file } = getRankFile(enPassantSquare);
    hash ^= zobristKeys.enPassant[file];
  }
```

```
  hash ^= zobristKeys.castling[castlingRights];
  hash ^= zobristKeys.side[side];

  return hash;
}
```

Es wird über alle Felder auf dem Brett iteriert und falls eine Figur sich darauf befindet, wird aus den vorgenerierten Zobrist Zahlen, die entsprechende herausgesucht und anschliessend mit der *hash* Variable XOR-d.

Dasselbe passiert für das *En-Passant* Feld, die Rochaderechte und die Seite, die am Zug ist. Das Resultat ist eine maximum 64-Bit Ganzzahl. Gleiche Positionen produzieren so immer den gleichen Wert und können einfach und effizient verglichen werden.

Die *computeZobristHash* Funktion wird nach jedem Zug aufgerufen und der Rückgabewert in der *ChessBoard* Klasse in einem Array gespeichert. Es ist nun trivial drei gleiche Werte in diesem Array zu finden.

```
if (hasTriplets(this.#zobristHashHistory)) {
  return { result: GameResult.Draw, type: DrawType.ThreefoldRepetition };
}
```

5.8.4 Unzureichende Figuren

Das letzte Szenario für ein Unentschieden ist, wenn keine Seite genügend Figuren hat, um den Gegner schachmatt zu setzen.

Fälle in denen das zutrifft:

- König gegen König
- König gegen König und Läufer
- König gegen König und Springer
- König und Läufer gegen König und Läufer (es können auch mehrere Läufer sein), solange beide Läufer auf Feldern der gleichen Farbe stehen.
-

Diese Zustände können mit den Bitboards einfach überprüft werden. Der Autor hält es nicht für nötig den Code hier auszuführen.

```
if (this.detectDeadPosition()) {
  return { result: GameResult.Draw, type: DrawType.DeadPosition };
}
```

5.9 Notiz an die lesende Person

Mit der dokumentierten Implementierung des Zuggenerators ist die Funktionalität des Programms nun vollständig und es ist voll funktionsfähig. Es ist jedoch wichtig zu beachten, dass die Programmcode-Abschnitte in diesem Dokument teilweise vereinfacht wurden, um das Verständnis zu erleichtern. In der Realität wurde eine erweiterte Funktionalität implementiert, die für die eigentliche Programmfunktionalität weniger wichtig ist. Zum Beispiel speichert das Programm eine Historie der durchgeführten Züge, auf die andere Programme zugreifen können. Zudem wurde eine Event Funktionalität eingebaut, die Konsumenten der *ChessBoard* Klasse benachrichtigt, wenn ein Zug gemacht wurde.

5.10 Evaluation

5.10.1 Performance Benchmarks

Um die Effizienz des Zuggenerators zu testen, bieten sich Leistungs-Benchmarks an. Es macht Sinn herauszufinden, wie lange die Zuggeneration für eine beliebige Position dauert. Folgende Messung wurde dafür etabliert:

Messung	Leistungs-Benchmark
Beschreibung	Es sollen simulierte Schachspiele durchgeführt werden, indem zufällige legale Züge gemacht werden. Dabei wird gemessen, wie lange der Zuggenerator in Millisekunden braucht, um einen Zug zu machen und die Züge für die resultierende Position zu berechnen.
Erfolgskriterium	Die Messungen sollen im Durchschnitt unter 10 Millisekunden liegen.
Parameter	Die Messungen werden für folgende Fälle durchgeführt: - 200 Spiele - 500 Spiele - 1000 Spiele
Iterationen	Jeder Testfall wird drei Mal durchgeführt.
Infrastruktur	Der Test wird in einem Docker Container auf dem Rechner des Autors laufengelassen. Der Gameserver Prozess läuft in einem Docker Container, der folgende Ressourcen zur Verfügung hat: - 2 CPU-Kerne - 8GB Memory Gerätespezifikation: Apple Macbook Pro, 14-Zoll, 2021 - 32GB Memory - Apple Silicon M1 Prozessor

5.10.1.1 Implementation Benchmark

Die Plattform *NodeJS*, auf welcher das Programm aufbaut, stellt nativ Funktionen für exakte Zeitmessungen zur Verfügung.

```
const time = process.hrtime();
```

Die *hrtime* Funktion startet eine Messung. Die Funktion kann dann mit dem Rückgabewert des ersten Aufrufs erneut ausgeführt werden, um die Differenz zu erhalten.

```
const diff = process.hrtime(time);
```

Hierfür wird eine Hilfsfunktion verwendet:

```
export const benchmark = () => {
  const time = process.hrtime();

  return () => {
    const diff = process.hrtime(time);
    return diff[0] * 1000 + diff[1] / 1000000;
  };
};
```

Die Funktion startet die Messung und gibt eine Funktion zurück, die aufgerufen werden kann, um die Messung zu stoppen und die Differenz in Millisekunden zurückzugeben.

Ein kleines Programm simuliert nun die Spiele, rechnet die durchschnittliche Zeit pro Spiel aus und errechnet am Ende den Durchschnitt aller Spiele aus.

```
const totalBenchmarks: number[] = [];
const benchmarkGame = () => {
  let board = ChessBoard.fromInitialPosition();
  let benchmarks: number[] = [];

  while (!board.gameOutcome) {
    const move =
      board.legalMoves[Math.floor(Math.random() * board.legalMoves.length)];

    const stopBenchmark = benchmark();
    board.makeMove(move);
    const time = stopBenchmark();
    benchmarks.push(time);
  }

  return calculateAverage(benchmarks);
};

for (let i = 0; i < 1000; i++) {
  const avg = benchmarkGame();
  totalBenchmarks.push(avg);
}

const totalAvg = calculateAverage(totalBenchmarks);
console.log("Total Average time in ms", totalAvg);
```

5.10.1.2 Resultate

Folgend die Resultate des Leistungs-Benchmarks:

Testfall	Durchgang 1	Durchgang 2	Durchgang 3
200 Spiele	0.4734839536799834ms	0.46874514357250185ms	0.46986839362082833ms
500 Spiele	0.46869839509834776ms	0.4757708480603567ms	0.4727342475920482ms
1000 Spiele	0.47502995916685964ms	0.46524110686996667ms	0.46606582927007506ms

5.10.1.3 Auswertung

Die Resultate übertreffen die Erwartungen um ein Weites. Durch alle Testfälle und Durchgänge ist der Wert konsistent unter einer halben Millisekunde und weicht nur um einen Bruchteil einer Millisekunde ab. Somit kann gesagt werden, dass der Zuggenerator enorm effizient ist.

5.10.2 Automatisierte Tests

Um sicherzustellen, dass die Funktionalität korrekt und fehlerfrei implementiert ist, können automatisierte Tests angewendet werden. Hierfür werden offizielle Spiele von Schachmeistern verwendet. Diese sind öffentlich im Internet in verschiedenen Formaten verfügbar. [51]

Testfall	Simulation von offiziellen Spielen von Schachmeistern
Beschreibung	Es werden Spiele vom ehemaligen Schachweltmeister *Magnus Carlson* gegen unterschiedliche Kontrahenten mit dem Zuggenerator simuliert. Es werden zufällige Spiele ausgewählt, die in einer Datenbank verfügbar sind.
Erfolgskriterium	Die Spiele laufen fehlerfrei ab.
Parameter	Der Test soll für 25 verschiedene Spiele automatisiert durchgeführt werden.

5.10.2.1 Implementation Automatisierter Test

Die Spiele müssen heruntergeladen und in das korrekte Format gebracht werden (ein Array von Zügen). Anschliessend können sie in einem Test nacheinander ausgeführt werden:

```
describe("ChessBoard Class", () => {
  it("should play the full game without errors", () => {
    gamesDb.forEach((game) => {
      const board = ChessBoard.fromInitialPosition();

      game.forEach((move) => {
        board.makeMove(move);
      });
    });
  });
});
```

Wenn kein Fehler produziert wird, ist das Spiel korrekt durchgelaufen.

5.10.2.2 Resultate

Spieler Weiss	**Spieler Schwarz**	**Resultat**
Carlsen M.	Brameld A.	Erfolg
Carslen M.	Fant G.	Erfolg
Tallaksen G.	Carlsen M.	Erfolg
Carlsen M.	Nilssen J.	Erfolg
Grubert C.	Carlsen M.	Erfolg
Carlsen M.	Johansen KR.	Erfolg
Sorensen H.	Carlsen M.	Erfolg
Carlsen M.	Moen A.	Erfolg
Hagberg O.	Carlsen M.	Erfolg
Carlsen M.	Flores R.	Erfolg
Carlsen M.	Breivik L.	Erfolg
Kabashaj A.	Carlsen M.	Erfolg
Carlsen M.	Rukovci S.	Erfolg
Aarefjord C.	Carlsen M.	Erfolg
Carlsen M.	Johansen S.	Erfolg
Badea B.	Carlsen M.	Erfolg
Carlsen M.	Vegh E.	Erfolg
Carlsen M.	Hole O.	Erfolg
Hersvik D.	Carlsen M.	Erfolg

Carlsen M.	Hermansson E.	Erfolg
Hitzgerova G.	Carlsen M.	Erfolg
Carlsen M.	Caoili A.	Erfolg
Petrov M.	Carlsen M.	Erfolg
Carlsen M.	Thorhallson T.	Erfolg
Carlsen M.	Lahlum H.	Fehler

5.10.2.3 Auswertung

Wie man in den Resultaten sehen kann, sind alle Spiele, bis auf das letzte erfolgreich durchgelaufen. Nach intensivem Debugging hat sich herausgestellt, dass der Grund warum das letzte Spiel nicht erfolgreich war, die Dreifachrepetition Regel für Unentschieden war. In Spielen über dem Brett (nicht online) können Spieler jeweils wählen, ob sie Gebrauch von der Dreifachrepetition Regel machen wollen oder nicht. In diesem Fall wurde dies nicht gemacht und das Spiel ging weiter. Der Zuggenerator hat allerdings automatisch das Spiel beendet, was der Auslöser für den Fehlschlag war.

Hierfür muss nichts unternommen werden, um das Problem zu lösen, da kein Fehler in der Funktionalität besteht. Der Test ist somit ein Erfolg.

6 Spielserver

In folgendem Abschnitt wird die Umsetzung des Spielservers behandelt. Zuerst werden die auf den Spielserver spezifischen Anforderungen analysiert und darauffolgend wird die Implementation angegangen. Wie bereits in einem vorherigen Kapitel beschrieben, wird für den Spielserver die Bibliothek *Colyseus* verwendet.

6.1 Anforderungsspezifikation

Anhand der allgemeinen Anforderungen an die Plattform können spezifische Anforderungen an den Spielserver erstellt werden.

Anforderung	**Beschreibung**
Echtzeit	Es wird ein Protokoll für die Kommunikation verwendet, das Echtzeit Kommunikation zulässt.
Authentifizierung	Clients müssen die Erlaubnis haben eine Verbindung mit dem Server zu starten.
Spiel erstellen	Clients können eine Verbindung mit dem Server herstellen, um ein Spiel zu starten.
Zeitformate	Spiele können mit verschiedenen Zeitformaten erstellt werden.
Spiel beitreten / Paarung	Clients können sich mit einem erstellten Spiel verbinden, um gegen den Herausforderer anzutreten.
Authentifizierte Spieler	Clients können authentifiziert sein, wodurch ihre vergangenen Spiele in der Datenbank gespeichert werden.
Anonyme Spieler	Clients können anonym sein.
Zuweisung Farbe	Wenn zwei Clients gepaart wurden, wird ihnen zufallsmässig eine Farbe der Figuren zugeteilt.
Zug machen	Clients können, solange sie am Zug sind, einen Zug machen.
Zug verifizieren	Um Betrug zu vermeiden, wird verifiziert, ob der Client am Zug ist und ob der Zug gültig ist.
Resignieren	Clients können zu jedem Zeitpunkt das Spiel abbrechen, indem sie resignieren und somit das Spiel verlieren.
Remis anbieten	Clients können zu jedem Zeitpunkt ein Remis anbieten.
Remis akzeptieren / ablehnen	Wenn der gegnerische Client ein Remis anbietet, kann ein Client dieses akzeptieren, um das Spiel sofort zu beenden oder ablehnen, um fortzufahren.
Uhr	Wenn mit einem Zeitformat gespielt wird, startet die Uhr, nachdem der erste Zug gemacht wurde und wechselt ab dann pro Client ab.
Zeit abgelaufen	Wenn die Bedenkzeit eines Clients abläuft, wird das Spiel abgebrochen und der Client verliert.
Abbruch	Wenn ein Client die Verbindung zum Server beendet, während ein Spiel läuft, wird das Spiel abgebrochen und der Client verliert.
Skalierbarkeit	Es muss die Möglichkeit bestehen, den Server horizontal zu skalieren, sodass grössere Lasten ertragen werden können.

6.2 Einführung in Colyseus

Im Folgenden wird kurz erklärt wie Colyseus funktioniert. Wichtig zu wissen ist, dass Colyseus neben dem Server auch eine Bibliothek für Browserclients anbietet, die Verbindung und Kommunikation mit Räumen abstrahiert und vereinfacht.

6.2.1 Räume

Colyseus als Spielserver bietet Funktionalität für Online-Spiele, die ansonsten aufwändig wären, selbst zu programmieren. Das Konzept von Colyseus basiert auf Räumen. Die Räume in Colyseus fungieren als individuelle, isolierte Umgebungen, in denen die Spieler miteinander und mit der Spiellogik interagieren. Jeder Raum hat seinen eigenen Satz von Regeln, Spielerinteraktionen und Spielzuständen. Räume können als separate Instanzen von Spielen oder Ebenen betrachtet werden, in denen eine Teilmenge aller angeschlossenen Spieler in Echtzeit interagiert. Die Spieler können Räumen dynamisch beitreten und sie verlassen, und jeder Raum handhabt seine eigene Logik unabhängig von den anderen. Räume können mit Optionen konfiguriert werden, also beispielsweise Spieleinstellungen. [52]

Im Falle des Schachspiels wird wohl nur ein einziger Raumtyp benötigt, der die Funktionalität für das Spiel einkapselt. Für jedes Spiel zwischen zwei Spielern wird ein Raum erstellt und am Ende des Spiels wieder geschlossen. Um mit Clients zu kommunizieren, gibt es für Räume zwei Möglichkeiten, die im Folgenden erörtert werden.

6.2.2 Zustand

Der Zustand bezieht sich auf die aktuellen Daten und Bedingungen eines Spiels in einem Raum zu einem bestimmten Zeitpunkt. Er kapselt alle dynamischen Informationen, wie zum Beispiel den Spielstand und alle anderen variablen Inhalte. Der Status wird auf dem Server verwaltet und aktualisiert und über alle mit einem Raum verbundenen Clients in Echtzeit synchronisiert. Dadurch wird sichergestellt, dass alle Spieler einen einheitlichen und aktuellen Überblick über die Spielumgebung und -aktivitäten haben. Clients haben nicht die Möglichkeit den Zustand des Raumes direkt zu verändern, aber sie können Nachrichten an den Server schicken, der diese verarbeitet und dann den Zustand aktualisiert.

So funktioniert die Zustandssynchronisation zwischen Server und Client:

1. Beim Verbinden mit einem Raum, sendet der Server dem Client den gesamten Zustand.
2. Wenn der Server nun den Zustand ändert, werden die Änderungen von Colyseus gesammelt.
3. In einem Intervall (standardmässig 50ms) werden alle gesammelten Änderungen vom Server an den Client geschickt. Dabei werden ausschliesslich echte Änderungen geschickt und nicht der gesamte Zustand. Dieser Vorgang nennt sich *patch*.
4. Der Client wendet die Änderungen automatisch an und kann dann die Daten verarbeiten.

Mit dieser Technik ist der aktuelle Zustand vom Server auf allen Clients in fast Echtzeit verfügbar.

6.2.3 Nachrichten

Da Clients den Zustand nicht direkt beeinflussen können, benötigen sie eine Methode zur Kommunikation mit dem Server. Hierzu können sie Nachrichten an den Server senden. Der Server bestimmt dabei, welche Art von Nachrichten akzeptiert werden und welche Daten in diesen Nachrichten enthalten sein müssen. Jede Nachricht ist durch eine Zeichenkette gekennzeichnet und kann zusätzliche Daten beinhalten.

Ebenso hat der Server die Fähigkeit, Nachrichten an die Clients zu senden. Hierbei kann er nach dem gleichen Prinzip vorgehen und entweder alle oder nur ausgewählte Clients adressieren. Das ist besonders praktisch, wenn bestimmte Informationen nur für einen Client bestimmt sind oder wenn es sich um vertrauliche Daten handelt.

6.3 Kommunikationsarchitektur

Bevor mit der Implementation begonnen wird, ist es sinnvoll, sich zu überlegen, welche Informationen Clients und der Server benötigen, um effizient miteinander zu kommunizieren.

6.3.1 Konzept

Der Server muss zwingend die Schachzüge der Spieler validieren. Falls das nicht geschehen würde, könnten Spieler ganz einfach im Spiel betrügen und illegale Züge ausführen. Deswegen liegt es nahe den Zuggenerator, der im vorherigen Kapitel erarbeitet wurde, auf dem Server laufen zu lassen. Für jeden Raum wird eine Instanz des Zuggenerators erstellt, welcher die Züge automatisch validiert.

Der Server könnte nun theoretisch die legalen Züge, die Brettaufstellung und alle weiteren nötigen Informationen im Zustand speichern und sie somit dem Client zur Verfügung zu stellen und somit den Gebrauch des Zuggenerators im Client eliminieren. Der Vorteil wäre, dass der Zuggenerator im Server und im Client nicht synchronisiert werden muss und somit Komplexität entfällt. Der Nachteil hingegen, ist es, dass der Zustand im Raum grösser wird und mehr Daten übertragen werden müssen, was die Leistung beeinträchtigen könnte, speziell bei vielen simultanen Spielen / Räumen auf dem gleichen Server. Dazu kommt, dass die Spieler eine unnötige Verzögerung bei eigenen Zügen erleben könnten, da das Brett erst aktualisiert wird, nachdem die Nachricht für den Zug gesendet und der Zustand im Server angewendet wurde.

Aus Gründen der Performance liegt die Tendenz bei der Variante den Zuggenerator auf Client und Server laufenzulassen und synchron zu halten. Um die finale Entscheidung zu machen, wird die Netzwerkkommunikation auf der Plattform lichess.org analysiert. Dieser Prozess kann Einsicht geben, wie eine etablierte Schachplattform dieses Problem gelöst hat.

6.3.1.1 Analyse Lichess

Der Programmcode der Plattform lichess.org ist vollständig öffentlich verfügbar. Allerdings ist es einfacher die Kommunikation mit einer Art reverse Engineering Methode zu analysieren, indem ein Spiel auf der Plattform gestartet und mittels Browserwerkzeugen die Netzwerkkommunikation betrachtet wird. In Chromium basierten Browsern steht in den Entwicklerwerkzeugen die Möglichkeit zur Verfügung Nachrichten zu inspizieren, die via Websocket Transport hin und her gesendet werden.

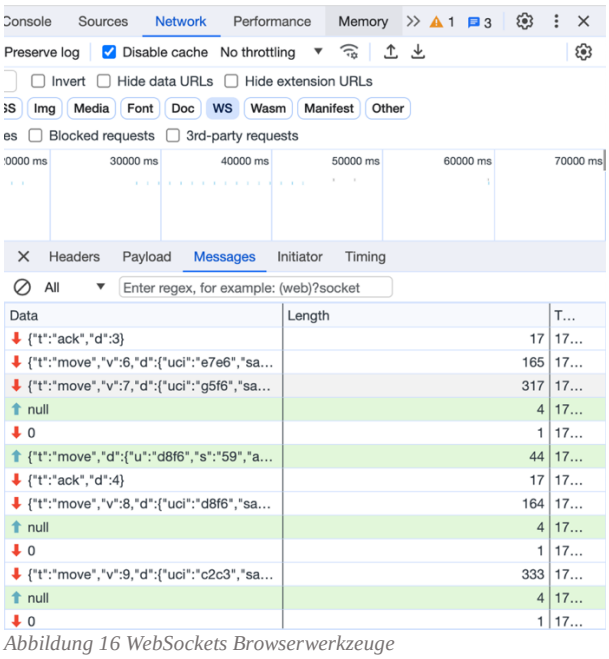

Abbildung 16 WebSockets Browserwerkzeuge

6.3.1.2 Resultate

Lichess sendet verschiedene Pakete im JSON-Format, identifiziert durch das JSON-Feld *t*, was vermutlich für "type" steht. Interessant ist hier vor allem der Typ "move". Folgend ein Beispielpaket dieser Art.

```
{
  "t": "move",
  "v": 6,
  "d": {
    "uci": "e7e6",
    "san": "e6",
    "fen": "rnbqkb1r/ppp2ppp/4pn2/3p2B1/3P4/P7/1PP1PPPP/RN1QKBNR",
    "ply": 6,
    "clock": { "white": 587.66, "black": 575.49, "lag": 8 }
  }
}
```

Das Paket enthält Informationen zum Zug in verschiedenen Formaten, sowie eine *FEN* und Informationen zur Bedenkzeit pro Spieler. Die *FEN* beinhaltet ausschliesslich die Position der Figuren. In einigen Nachrichten ist auch ein Feld "dests" vorhanden, welches vermutlich die legalen Züge beinhaltet. Dieses Feld ist nicht in jeder Nachricht vorhanden, was dem Autor merkwürdig erscheint und wofür er keine Erklärung hat.

Für eine Vollständige Brettrepräsentation fehlen Informationen wie beispielsweise Rochaderechte, *En-Passant* Feld und die Farbe, die aktuell am Zug ist. Der Autor interpretiert das so, dass Lichess auf dem Client eine Art Zuggenerator oder Schachbrett Repräsentation hat.

Interessant sind auch die Informationen zur Bedenkzeit. Die Zeit wird scheinbar auf dem Server gespeichert. Da die Uhr auf dem Client im Browser jede Sekunde heruntertickt, lässt es sich annehmen, dass die Uhr simultan auf dem Client und im Server läuft. Bei jedem Zug wird dann die Zeit vom Server auf den Client übertragen, um die Uhr synchron zu halten.

6.3.1.3 Entscheidung

Anhand der Informationen ist der Entscheid gefallen, den Zuggenerator auf dem Client und auf dem Server simultan laufen zu lassen. Der Autor hält das für sinnvoll, da die Züge auf dem Client generiert werden können, ohne Latenz. Zudem sind die *Patches* für die Zustandsaktualisierungen kleiner und der Server hat dadurch weniger Last.

Dadurch, dass der Zuggenerator auf dem Client läuft, braucht er fast keine Informationen vom Server, um das Spiel abzubilden und synchron zu halten. Einzig den Zug des Gegners und die Bedenkzeit pro Spieler muss der Server liefern. Sobald der Gegner einen Zug macht, soll der Server dem Client eine Nachricht schicken, mit den Informationen des Zugs, woraufhin dieser auf dem Client angewendet wird.

Die Bedenkzeit pro Spieler soll auf dem Server und dem Client gleichzeitig laufen gelassen werden. Um die Zeit synchron zu halten, wird die Zeit auf dem Client nach jedem Zug mit der Zeit vom Server abgeglichen.

6.3.2 Zustand

Wie bereits erwähnt sind nur wenige Informationen im Zustand notwendig. Diese sind wie folgt:

Information	Beschreibung
Spieler	Im Zustand sollen Informationen zu den Spielern gespeichert werden, damit diese vom Client angezeigt werden können.
Bedenkzeit	Die Bedenkzeit wird vom Server nach jedem Zug in den Zustand geschrieben und die Clients können somit darauf zugreifen.
Remis anbieten	Ob ein Spieler ein Remis anbietet, kann ebenfalls im Zustand gespeichert werden.

6.3.3 Nachrichten

Der Rest der Kommunikation geschieht durch die Nachrichten. Anhand der Anforderungen und der erarbeiteten Informationen wurden folgende Nachrichten eruiert.

6.3.3.1 Eingehende Nachrichten

Eingehende Nachrichten, sind Nachrichten, die der Server erwartet. Diese beinhalten nicht das Erstellen oder Verbinden zu einem Spiel, da dieser Teil Colyseus automatisch abnimmt.

Identifizierung	Zweck	Daten
make_move	Nachricht, die an den Server geschickt wird, sobald der Client einen Zug gemacht hat.	*SAN*-Repräsentation des Zuges.
resign	Nachricht, um das Spiel aufzugeben.	-
offer_draw	Nachricht, um ein Remis anzubieten	-
accept_draw	Nachricht, um ein Remisangebot anzunehmen	-

Es ist keine Nachricht notwendig, um ein Remis abzulehnen, da das Angebot einfach beim nächsten Zug des Gegners ungültig gemacht werden kann. Um ein Remis abzulehnen, kann der Spieler also einfach ziehen.

6.3.3.2 Ausgehende Nachrichten

Ausgehende Nachrichten, sind Nachrichten, die der Server verschickt.

Identifizierung	Zweck	Daten
opponent_move	Notifiziert den Client über einen Zug des Gegners	*SAN*-Repräsentation des Zuges.
game_concluded	Notifiziert Clients, wenn das Spiel vorüber ist, beispielsweise wenn die Zeit abgelaufen ist, schachmatt entsteht oder Remis vereinbart wird.	Information zum Grund des Endes des Spiels

6.4 Skalierbarkeit

Eine Anforderung an den Server ist, dass er in Zukunft skaliert werden kann. Für den Fall, dass die Plattform zu einem gewissen Punkt eine sehr hohe Last ertragen muss, sollen mehrere Instanzen des Servers nebeneinander laufen gelassen werden können. Dies ist für Gameserver generell von enormer Wichtigkeit. Bei einem Spiel wie Schach, wo es in schnellen Spielmodi wie beispielsweise Bullet-Schach (eine Minute Denkzeit pro Spieler) auf Millisekunden ankommt ist es wichtig, dass der Server nicht unter der Last von gleichzeitigen Verbindungen leidet.

Auf Linux Servern, die am verbreitetsten im Einsatz sind, ist der Standardwert für die maximale Anzahl an offenen Verbindungen 1024 pro Prozess. [53]
Dieses Limit kann via Konfiguration erhöht werden. Daher kann man annehmen, dass auf dem billigsten Cloud-Server 1024 Verbindungen stabil laufen. In der Theorie würde dies für die Schachwebsite bedeuten, dass über 500 Spiele ohne Probleme parallel laufengelassen werden können. Selbstverständlich kommt es dabei auch darauf an, was für Aufgaben der Server erfüllen muss und ob diese rechenintensiv sind. Nichtsdestotrotz macht es bei der Entwicklung einer Schachwebseite Sinn vorauszuplanen, für den Fall, dass der Server an einem gewissen Zeitpunkt eine höhere Last stemmen muss, als ein einziger Prozess das erlaubt.

Glücklicherweise hat Colyseus integriert die Fähigkeit horizontal zu skalieren. Colyseus hat die Möglichkeit zwischen Prozessen zu kommunizieren. Diese Prozesse können auch auf verschiedene Server verteilt werden. Prozesse kommunizieren miteinander via Redis, einer weit etablierten Key-Value Datenbank, die oft für Caching und Nachrichtenaustausch zwischen Prozessen verwendet wird.

Laut Colyseus Dokumentation funktioniert die Lastverteilung so:

- Räume gehören jeweils zu nur einem Prozess
- Räume werden gleichmässig auf alle verfügbaren Prozesse verteilt
- Jeder Prozess hat eine maximale Anzahl von verbundenen Clients
- Client Verbindungen sind direkt assoziiert mit dem Prozess, der den Raum erstellt hat

```
const gameServer = new Server({
  // ...
  presence: new RedisPresence(),
  driver: new RedisDriver(),
});
```

Beim Erstellen eines Colyseus Servers wird eine Verbindung mit Redis etabliert. In der Redis Datenbank werden alle Spiele zwischengespeichert. Wenn nun ein Client einem Spiel beitreten möchte, leitet ihn der empfangende Server automatisch zum richtigen Prozess, auf welchem der jeweilige Raum läuft, weiter.

Beim Deployment kann dann vor die Prozesse des Servers ein Loadbalancer geschaltet werden, ein Programm, das Verbindungen automatisch gleichmässig an alle Prozesse dahinter verteilt. Dafür stellt Colyseus keine eigene Technologie zur Verfügung, denn es kann jeder beliebige Loadbalancer gewählt werden.

Mit dieser Funktionalität kann Colyseus fast endlos skaliert werden, indem einfach mehr Prozesse gestartet werden. Loadbalancer können auch teilweise automatisch neue Prozesse starten, wenn die bestehenden zu sehr ausgelastet sind.

Da die Skalierung auf mehrere Prozesse aber für den MVP nicht nötig ist, wird in dieser Arbeit davon abgesehen. Wichtig ist nur, dass die Möglichkeit besteht, für die Zukunft.

6.4.1 Colyseus Cloud

Eine weitere, sehr einfache Möglichkeit, die Applikation zu skalieren ist den Server auf Colyseus' Cloud Infrastruktur laufen zu lassen. Damit lässt sich die Applikation automatisch skalieren, ohne sich selbst Gedanken über Loadbalancing und Redis machen zu müssen. Colyseus Cloud läuft auf AWS-Infrastruktur und ist optimieret für Gameserver Anwendungen mit Colyseus.

6.5 Implementation

Im folgenden Kapitel wird die eigentliche Implementation des Gameservers behandelt und die gesammelten Informationen aus dem letzten Kapitel angewendet.

6.5.1 Datenbank Setup

Bevor die eigentliche Implementation angegangen wird, macht es Sinn den Client für die Datenbank aufzusetzen. Dafür wird die Bibliothek *Prisma* verwendet. Grund dafür ist, dass der Autor die Bibliothek bereits bestens kennt und schon oft eingesetzt hat.

Prisma ist ein fortschrittliches ORM (Object Relational Mapping) Tool, das Entwicklern ermöglicht, Datenbankabfragen effizient und sicher durchzuführen, ohne direkt SQL-Code zu schreiben. Im Kern von Prisma steht das Schema, welches die Struktur der Datenbank und die Beziehungen zwischen den Tabellen beschreibt. Aus diesem Schema wird ein spezialisierter Client generiert, der sich nahtlos in den Anwendungscode integrieren lässt und optimierte Abfragen ermöglicht. Wichtig zu betonen ist, dass dieser Generierungsvorgang nicht während der Laufzeit der Anwendung stattfindet. Stattdessen wird der Client jedes Mal neu generiert, wenn Änderungen am Datenbank-Schema vorgenommen werden. Dies gewährleistet, dass der generierte Client immer auf dem neuesten Stand ist und mit der aktuellen Struktur der Datenbank synchronisiert ist. [54]

Durch die Generierung kennt der Client das gesamte Schema, wodurch direkte und intuitive Abfragen möglich werden, ohne dass der Entwickler die spezifischen Details und Strukturen der Datenbank im Kopf behalten muss. Ein praktisches Beispiel hierfür wäre: Angenommen, wir haben eine Datenbank mit einer Tabelle namens "User". Mit einem durch Prisma generierten Client könnten wir einfach einen neuen Benutzer hinzufügen oder vorhandene Benutzerdaten abrufen, indem wir folgenden Code verwenden:

```
await prisma.user.create({
    data: {
        name: "Max Mustermann",
        email: "max@example.com"
    }
});

const user = await prisma.user.findUnique({
    where: {
        email: "max@example.com"
    }
});
```

Dabei generiert Prisma auch Typen in TypeScript, sodass der Entwickler Intellisense für alle Attribute des Benutzers hat, wie im folgenden Screenshot zu sehen ist.

Abbildung 17 Prisma Intellisense

Prisma wird im Monorepository in einem eigenen Package generiert, sodass der Prisma Client von allen anderen Teilen der Applikation auch genutzt werden kann, beispielsweise der Applikation für das Benutzerinterface.

Um das Schema zu definieren, beitet Prisma eine eigene Sprache an. Folgendes Schema wurde definiert, anhand der Anforderungen und dem Design Kapitel in dieser Arbeit.

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgres"
  url      = env("DATABASE_URL")
}

model User {
  id            String    @id @default(cuid())
  name          String?
  email         String?   @unique
  gamesAsWhite  Game[]    @relation(name: "user_games_white")
  gamesAsBlack  Game[]    @relation(name: "user_games_black")

  createdAt DateTime @default(now())
  updatedAt DateTime @default(now()) @updatedAt
}

model Game {
  id            String  @id @default(cuid())
  whitePlayerId String?
  whitePlayer   User?   @relation(fields: [whitePlayerId], references: [id], name:
"user_games_white")
  blackPlayerId String?
  blackPlayer   User?   @relation(fields: [blackPlayerId], references: [id], name:
"user_games_black")
  result     String?
  resultType String?
```

```
  baseTime  Int?
  increment Int?
  moves     Json
  moveCount Int
  createdAt DateTime @default(now())
  updatedAt DateTime @default(now()) @updatedAt
}
```

Im Schema wird eine User-Tabelle definiert, mit Relationen zur Game-Tabelle. Ein Spiel hat immer zwei Benutzer assoziiert, einer für Weiss und einer für Schwarz. Die Relation zu Benutzern ist optional, da Benutzer auch anonym sein können, und somit keinen Benutzeraccount haben. Die Züge werden als JSON-Array gespeichert, in der *SAN*-Notation.

Mittels dieses Schemas kann nun per Befehlszeile ein Befehl ausgeführt werden, um den Client zu generieren. Prisma generiert dafür Migrationen, also Dateien, die SQL-Code enthalten. Bei jeder Änderung am Schema, wird eine neue Migration generiert, die die Änderungen widerspiegelt.

Es muss angefügt werden, dass hier etwas vorgegriffen wird. Benutzer werden nicht im Game Server erstellt, sondern von der Bibliothek, die in der Applikation für das Benutzerinterface für die Authentifizierung verwendet wird. Dazu mehr in einem späteren Kapitel.

6.5.2 Server und Transport

Um eine Colyseus Instanz zu starten, muss zuerst ein Server erstellt werden. Colyseus basiert standardmässig auf Express, der am breitesten etablierten Webserver Bibliothek für NodeJS. Der zugrundeliegende Webserver kann allerdings geändert werden. Express hat ein grosses Ökosystem ist allerdings nicht so schnell wie Alternativen. Der Autor hat sich für Fastify entschieden, eine alternative Webserver Bibliothek, die um einiges schneller ist als Express. Da Performance hier im Vordergrund steht, macht es Sinn die schnellere Variante zu wählen. [55]

```
let server: HttpServer;
const serverFactory: FastifyServerFactory = (handler) => {
  server = createServer((req, res) => {
    handler(req, res);
  });
  return server;
};

const bootstrap = async () => {
  const fastify = Fastify({ serverFactory });
  await fastify.register(fastifyExpress);

  fastify.ready(() => {
    const gameServer = new Server({
      transport: new WebSocketTransport({
        server,
      }),
    });

    gameServer.listen(8080);
  });
};
bootstrap();
```

Obiger Programmcode erstellt eine Colyseusinstanz auf dem Port 8080, angetrieben von Fastify. Um Fastify mit Colyseus kompatibel zu machen, muss eine Helferfunktion *fastifyExpress*, die Fastify selbst zur Verfügung stellt, angewendet werden.

Ein weiterer nennenswerter Punkt ist der Transport. Hier kann bei Colyseus entschieden werden zwischen dem Standard Websocket Transport, der auf der Bibliothek *ws* basiert, sowie einer weiteren Bibliothek namens *uWebsockets.js*, die bessere Leistung verspricht. Der Autor hat versucht *uWebsockets.js* zu verwenden, hatte damit aber Probleme in Kombination mit Fastify und sich deshalb für den Standard Websockets Transport entschieden, um mehr Zeit in wichtigere Anliegen investieren zu können.

6.5.3 Raum Setup

Wie bereits erwähnt, basiert Colyseus auf Räumen. Colyseus stellt eine Klasse zur Verfügung, die als Blaupause für einen Raum fungiert. Für jeden Raumtyp wird so eine Klasse erstellt und im Server registriert. Ohne Logik sieht die Klasse für das Schachspiel so aus:

```
export class ChessRoom extends Room<ChessRoomSchema, RoomOptions> {
  maxClients: number = 2; // Anzahl an maximalen Clients pro Raum

  onCreate(options: RoomOptions) {
    // Wird aufgerufen, wenn der Raum erstellt wird
  }

  async onAuth(client: Client, options: any, request: http.IncomingMessage) {
    // Wird vor onJoin aufgerufen, damit Zugriffsrechte kontrolliert werden können
  }

  onJoin(client: Client, options: any, auth: any) {
    // Wird aufgerufen, wenn ein Client sich zum Raum verbindet
  }

  onLeave(client: Client, consented: boolean) {
    // Wird aufgerufen, wenn ein Client die Verbindung trennt
  }

  onDispose() {
    // Wird aufgerufen, wenn ein Raum entsorgt wird
    // beispielsweise, wenn alle Clients die Verbindung trennen
  }
}
```

Dieser Raum kann dann beim Server registriert werden.

```
gameServer.define("chess", ChessRoom);
```

6.5.3.1 Zustand

Um den Zustand eines Raumes zu definieren, stellt Colyseus Schema Hilfsfunktionen zur Verfügung. Das Schema für den Schachraum sieht wie folgt aus, anhand der erarbeiteten Informationen im vorherigen Kapitel:

```
export class PlayerSchema extends Schema implements PlayerState {
  @type("string") userId: string;
  @type("string") name: string;
  @type("boolean") anonymous: boolean;
  @type("string") sessionId: string;
  @type("number") color: Color;
}

export class TimeControlSchema extends Schema implements TimeControlState {
  @type("number") white: number;
  @type("number") black: number;
}

export class ChessRoomSchema extends Schema implements ChessRoomState {
  @type({ map: PlayerSchema }) players = new MapSchema<PlayerSchema>();
  @type(TimeControlSchema) timeControl: TimeControlSchema;
  @type("boolean") ready: boolean = false;
  @type("string") drawOfferBy: string | null;
}
```

Die einzelnen Attribute werden mit *@type* annotiert. Dies teilt Colyseus mit, dass das jeweilige Attribut beim nächsten *Patch* auf Änderungen überprüft werden soll.

Der Raumzustand besteht aus Informationen zu den Spielern, sowie der Farbe der Figuren, die der Spieler kontrolliert. Dank des Monorepositories können im Gameserver Typen vom Zuggenerator verwendet werden, zum Beispiel das Enum *Color*.

Clients werden in Colyseus jeweils mit einer eindeutigen Zeichenkette identifiziert - diese ist im Spielerschema als *sessionId* gespeichert. Weiterhin sind auch die Informationen zur verbleibenden Zeit pro Farbe im Zustand gespeichert.

Dieser Zustand kann nun von der Raum Klasse aus manipuliert werden, beispielsweise so:

```
this.state.players.get('<<sessionId>>').color = Color.White;
```

Damit ist nun die Grundlage gelegt, um die Spiellogik mittels der Nachrichten einzubauen.

6.5.4 Nachrichten registrieren

Um nun Nachrichten von Clients zu empfangen können Colyseus Nachrichten-Listeners hinzugefügt werden. Diese Listeners werden aufgerufen, wenn ein Client, der mit dem jeweiligen Raum verbunden ist, eine Nachricht sendet. Nachrichten-Listeners werden registriert, wenn ein Raum erstellt wird, also in der *onCreate* Methode, die dafür gedacht ist. Da die eingehenden Nachrichten in einem vorherigen Kapitel bereits definiert wurden, können sie nun registriert werden.

```
this.onMessage<{ move: string }>(ClientMessage.Move, (client, payload) => {});
this.onMessage(ClientMessage.Resign, (client, payload) => {});
this.onMessage(ClientMessage.OfferDraw, (client, payload) => {});
this.onMessage(ClientMessage.AcceptDraw, (client, payload) => {});
```

Die Funktion, die als zweiter Parameter mitgegeben wird, wird aufgerufen, wenn der Server eine Nachricht vom jeweiligen Typ erhält. Die Funktion erhält als Parameter eine Repräsentation des Clients, der die Nachricht gesendet hat und ein Objekt, mit den Daten, die mitgeschickt wurden.

6.5.5 Commands

Die Spiellogik könnte direkt in den Callback-Funktionen der Nachrichten-Listeners implementiert werden. Dies würde aber dazu führen, dass der Programmcode in der Raum Klasse gross wird und nicht sehr gut wartbar ist. Deswegen bietet Colyseus *Commands* an. Ein Command ist ein Stück Programmcode, das vom Raum oder von einem anderen Command *dispatched*, also ausgelöst werden kann. Colyseus gibt nicht vor, wie die Gamelogik gestaltet werden soll, empfiehlt aber dieses Pattern zu verwenden. [56]

Commands können wie folgt dispatched werden:

```
this.dispatcher.dispatch(new OnMoveCommand(), {
    sessionId: client.sessionId,
    move: payload.move,
});
```

Alle Commands haben eine Referenz zur Raum Klasse und können somit jederzeit auf alle Informationen, die darin gespeichert sind, zugreifen.

Ein Command sieht so aus:

```
type Payload = {
  sessionId: string;
  move: string;
};

export class OnMoveCommand extends Command<ChessRoom, Payload> {
  async execute(payload: Payload) {
  }
}
```

Der Command hat nur eine einzige Funktion, die läuft, wenn der Command dispatched wird. Zudem hat der Command eine *Payload*, also Daten, die ihm mitgegeben werden können.

Folgend wird die Gamelogik, aufgeteilt in Commands dokumentiert.

6.5.6 Authentication

Bevor ein Client sich mit dem Raum verbinden kann, müssen seine Zugriffsrechte überprüft werden. Dies geschieht mittels der *onAuth* Methode in der Raum Klasse. Hier wird etwas vorgegriffen. Das Thema wird im Kapitel, das die Frontend Applikation behandelt noch einmal genauer behandelt.

Der Server erhält vom Client bei dem Aufbau der Websockets Verbindung ein Access Token. Darin ist, falls der Benutzer ein Konto hat, die ID des jeweiligen Benutzers gespeichert. Im Falle eines anonymen Spielers erhält der Gameserver trotzdem ein Token, aber statt der ID des Benutzers ist darin ein Flag, das den Client als Anonym identifiziert. Dieses Token wird mit dem Passwort, mit welchem es signiert wurde, auf Echtheit und Gültigkeit verifiziert.

```
async onAuth(client: Client, options: any, request: http.IncomingMessage) {
    if (!options.accessToken) return false;
    const secret = process.env.TOKEN_SECRET!;

    const verified = jwt.verify(options.accessToken, secret, {
```

```
      ignoreExpiration: false,
    });

    if (!verified || typeof verified !== "object") {
      return false;
    }

    if (verified.anonymous === true) {
      return { anonymous: true };
    } else if ("userId" in verified && typeof verified.userId === "string") {
      const user = await prismaClient.user.findFirstOrThrow({
        where: { id: verified.userId },
      });

      return user;
    }

    return false;
}
```

Falls der Client nicht anonym ist, wird der Benutzer aus der Datenbank geholt und von der Methode zurückgegeben. Der Rückgabewert der *onAuth* Methode wird anschliessend von Colyseus der *onJoin* Methode übermittelt.

6.5.7 Spiel beitreten

Um Spieler einem Raum hinzuzufügen, wird ein Command verwendet, welcher von der *onJoin* Methode dispatched wird. Er erhält als Parameter die Benutzerinformationen, die die *onAuth* Funktion zurückgegeben hat.

Der Command dient hauptsächlich dazu, die Informationen des Spielers in den Zustand zu schreiben. Dieser Command wird zwei Mal pro Spiel aufgerufen, für jeden Client einmal. Der Command erhält als Parameter die Informationen des Spielers, sowie die *sessionId*.

```
export class OnJoinCommand extends Command<ChessRoom, Payload> {
  execute(payload: Payload) {
    const player = new PlayerSchema().assign({
      sessionId: payload.sessionId,
      ...this.getUserState(payload.user),
    });
    this.state.players.set(payload.sessionId, player);

    if (this.state.players.size === 2) {
      logger.debug("Room populated, locking...", { id: this.room.roomId });
      this.room.lock();
      this.room.dispatcher.dispatch(new InitGameCommand());
    }
  }
}
```

Wenn sich der zweite Spieler verbindet, wird der Raum geschlossen, sodass sich keine weiteren Clients mehr verbinden können. Daraufhin wird der *InitGameCommand* dispatched.

6.5.8 Spiel initialisieren

Der *InitGameCommand* dient dazu alle Vorbereitungen zu machen, damit das Spiel losgehen kann. Er teilt beiden Spielern eine zufällige Farbe zu, erstellt eine Instanz des Zuggenerators von der Startposition aus, schreibt die Zeit pro Spieler in den Zustand und setzt dann den Booleanwert *ready* im Zustand auf wahr. Dies signalisiert den Clients, dass das Spiel nun gestartet werden kann. Wenn mindestens einer der Spieler nicht anonym ist, wird zudem ein neues Spiel in die Datenbank geschrieben. Einige Stellen wurden mit Kommentaren ersetzt aus Gründen der Ausführlichkeit

```
export class InitGameCommand extends Command<ChessRoom> {
  async execute() {
    let firstPlayerColor: Color;
    this.room.state.players.forEach((player) => {
      if (typeof firstPlayerColor !== "undefined") {
        player.color = firstPlayerColor ^ 1;
      } else {
        firstPlayerColor = Number(Math.random() < 0.5);
        player.color = firstPlayerColor;
      }
    });

    this.room.board = ChessBoard.fromInitialPosition();

    this.room.board.moveCallback = ({ gameOutcome }) => {
      if (gameOutcome) {
        // Wenn das Spiel endet, wird ein Command ausgelöst
        this.room.dispatcher.dispatch(new ConcludeGameCommand());
      }
    };

    // Hier wird die Zeit pro Spieler gesetzt

    const playerArray = Array.from(this.state.players.values());

    if (!playerArray.every((p) => p.anonymous)) {
      // Hier wird das Spiel in der Datenbank erstellt

      this.room.gameId = game.id;
    }

    this.room.state.ready = true;
  }
}
```

Sobald *ready* auf wahr ist, können Spieler nun Züge machen.

6.5.9 Zug machen

In einem vorherigen Abschnitt wurde der Nachrichten-Listener für die Nachricht *make_move* initialisiert. Der Listener dispatched nun den Command *OnMoveCommand*, welcher Züge von Spielern entgegennimmt.

Zuerst wird überprüft, ob der jeweilige Spieler überhaupt am Zug ist und ob das Spiel bereits gestartet hat. Dann werden Remis-Angebote zurückgesetzt, wenn der Zug vom Spieler stammt, der das Angebot erhält. Dies bedeutet, dass das Remis-Angebot abgelehnt wird.

Der Zug wird dem Zuggenerator übergeben zur Synchronisation:

```
this.room.board.makeMove(payload.move);
```

Anschliessend wird der Gegner über den Zug informiert, indem die ausgehende Nachricht *opponent_move* an den gegnerischen Client gesendet wird.

```
const opponentClient = this.room.clients.find(
  (c) => c.sessionId !== payload.sessionId
);
opponentClient?.send(ServerMessage.OpponentMove, {
  move: payload.move,
});
```

Weiterhin werden Commands dispatched, die jeweils die Uhr des Spielers stoppen und die des Gegners starten. Diese Commands werden in einem späteren Abschnitt abgedeckt.

```
this.room.dispatcher.dispatch(new StopClockCommand(), {
  color: player.color,
});

this.room.dispatcher.dispatch(new StartClockCommand(), {
  color: player.color ^ 1,
});
```

Zuletzt, falls einer der beiden Spieler nicht anonym ist und somit ein Spiel in der Datenbank existiert, wird die Game Tabelle mit dem neuen Zug aktualisiert.

Da der Programmcode ausführlich ist, wurden nur einzelne Teile aufgeführt.

6.5.10 Uhr starten und stoppen

Im vorherigen Abschnitt wurden die Commands zum Kontrollieren der Schachuhr bereits erwähnt. Es gibt jeweils einen Command um die Uhr zu starten und um sie zu stoppen.

6.5.10.1 Uhr starten

Der Command, um die Uhr zu starten ist simpel. Dabei wird, sofern das Spiel mit Zeitformat gespielt wird, eine Funktion aufgerufen, die den Timer startet mit der verbleibenden Zeit des jeweiligen Spielers. Bei Ablauf des Timeout wird der *ConcludeGameCommand* ausgeführt, der das Spiel beendet und im nächsten Abschnitt behandelt wird.

```
const remainingTime = payload.color
  ? this.state.timeControl.black
  : this.state.timeControl.white;

this.room.timer = this.clock.setTimeout(() => {
  this.room.board.setGameOutcome({
    result: payload.color ? GameResult.WhiteWins : GameResult.BlackWins,
```

```
    type: EndGameType.Timeout,
  });
  this.room.dispatcher.dispatch(new ConcludeGameCommand());
}, remainingTime);
```

Erwähnenswert ist hier, dass nicht die JavaScript native Funktion *setTimeout* verwendet wird, um den Timer zu starten. Die *setTimeout* Funktion wird üblicherweise dafür verwendet, eine Callback Funktion nach einer bestimmten Verzögerung auszuführen. Allerdings kann die native Version sehr ungenau sein. [57] Da die Genauigkeit von Zeitmessungen bei Spielen, die in Echtzeit laufen wichtig ist, bietet Colyseus eine sehr genaue, eigene Implementation an.

6.5.10.2 Uhr stoppen

Das Gegenstück des *StartClockCommand* ist der *StopClockCommand*. Falls die Uhr eines Spielers während seines Zuges nicht abgelaufen ist, wird sie hier gestoppt.

```
const remainingTime = payload.color
  ? this.state.timeControl.black
  : this.state.timeControl.white;

this.room.state.timeControl.isTicking = false;
this.room.timer.pause();
this.room.timer.clear();

const incrementMs = (this.room.metadata.timeControl?.increment ?? 0) * 1000;
const newTime = remainingTime - this.room.timer.elapsedTime + incrementMs;

this.state.timeControl.assign({
  [payload.color ? "black" : "white"]: newTime,
});
```

Zudem wird die neue, verbleibende Zeit des Spielers errechnet und in den Zustand geschrieben. Falls mit einem Zeitinkrement gespielt wird, wird dieses zur verbleibenden Zeit dazugerechnet.

6.5.11 Spielende

Wenn das Spiel, aus irgendeinem Grund zu Ende kommt, sei es Schachmatt, Remis, durch Aufgeben oder Verlieren auf Zeit, wird der *ConcludeGameCommand* ausgeführt. Dieser notifiziert die Clients, dass das Spiel beendet wurde.

```
this.room.broadcast(ServerMessage.GameConcluded, {
  gameOutcome,
});
```

Ab diesem Zeitpunkt können keine Züge mehr gemacht werden. Der Command stoppt zudem die laufende Uhr und speichert das Resultat des Spiels im jeweiligen Eintrag in der Datenbank.

6.5.12 Weitere Commands

Weiterhin gibt es Commands für das Resignieren, für das Remis-Angebot und für Remis akzeptieren. Diese werden hier nicht speziell aufgeführt, da keine nennenswerte Logik enthalten ist.

6.6 Evaluation

6.6.1 Lasttest

Um sicherzustellen, dass der Gameserver eine hohe Last ertragen kann, ohne abzustürzen, wird ein Lasttest durchgeführt. Colyseus stellt glücklicherweise ein Programm zur Verfügung, das für genau diesen Zweck programmiert wurde. Dem Programm kann ein Skript mitgegeben werden, das als Anleitung für die Lasttests dient und eine Anzahl an Clients, mit der getestet werden soll.
Das Skript dient als Client für Colyseus und simuliert jeweils ein einzelnes Spiel.

Testfall	Lasttests
Beschreibung	Es wird eine grosse Anzahl an Spielen gestartet, die gleichzeitig laufen.
Erfolgskriterium	Die Spiele sollen fehlerfrei durchlaufen, ohne dass der Server in die Knie geht, bis alle Spiele abgeschlossen sind.
Parameter	Der Test soll mit 1000 verbundenen Clients durchgeführt werden. Zwei Clients pro Spiel. Spiele werden mit einem Zeitformat von fünf Minuten und zwei Sekunden Inkrement durchgeführt.
Messungen	Während die Spiele laufen, werden alle fünf Sekunden folgende Werte gemessen in Bezug auf den Gameserver Prozess: - CPU-Auslastung in % - Arbeitsspeicher Auslastung in % - Arbeitsspeicher Auslastung in MB
Infrastruktur	Der Test wird in einem Docker Container auf dem Rechner des Autors laufengelassen. Der Gameserver Prozess läuft in einem Docker Container, der folgende Ressourcen zur Verfügung hat: - 2 CPU-Kerne - 8GB Memory Gerätespezifikation: Apple Macbook Pro, 14-Zoll, 2021 - 32GB Memory - Apple Silicon M1 Prozessor

6.6.1.1 Implementation

Folgendes Skript wurde geschrieben, um Spiele zu simulieren. Beide Clients, die mit einem Raum verbunden sind, machen abwechslungsweise zufällige Züge und nehmen sich eine Zufällige Bedenkzeit zwischen einer und sechs Sekunden.

```
async function main(options: Options) {
  let firstMoveMade = false;

  const board = ChessBoard.fromInitialPosition();
  const client = new Client(options.endpoint);
  const room = await client.joinOrCreate<ChessRoomSchema>(options.roomName, {
    timeControl: {
      time: 300,
      increment: 2,
    },
  });

  room.onMessage("*", (type, message) => {
    if (type === ServerMessage.OpponentMove) {
      board.makeMove(message.move);
```

```
        makeRandomMove();
      }

      if (type === ServerMessage.GameConcluded) {
          room.leave();
      }
    });

    room.onStateChange((state) => {
      if (state.ready && !firstMoveMade) {
        const [_, player] = Array.from(state.players).find(
          ([clientId]) => clientId === room.sessionId
        )!;

        if (player.color === Color.White) {
          makeRandomMove();
          firstMoveMade = true;
        }
      }
    });
}
```

Um die Nutzungswerte zu messen, wird ein Bash-Skript eingesetzt. Das Skript sucht nach der Prozess-ID des Gameservers und schreibt dann alle 5 Sekunden die vorhin definierten Werte in eine CSV-Datei.

```
echo "Timestamp,CPU%,MEM%,RSS (MB)" > process-stats.csv
while true; do
    PID=$(lsof -t -i :8080)
    if [ ! -z "$PID" ]; then
        TIMESTAMP=$(date +%Y-%m-%d_%H:%M:%S)
        CPU_MEM_PERCENT=$(ps u -p $PID | awk 'NR>1 {print $3 "," $4}')
        RSS_KB=$(cat /proc/$PID/status | grep 'VmRSS' | awk '{print $2}')
        RSS_MB=$(awk "BEGIN {printf \"%.2f\",${RSS_KB}/1024}")
        if [ ! -z "$CPU_MEM_PERCENT" ] && [ ! -z "$RSS_MB" ]; then
            echo "$TIMESTAMP,$CPU_MEM_PERCENT,$RSS_MB" >> process-stats.csv
        fi
    else
        echo "No process found on port 8080 at $(date +%Y-%m-%d_%H:%M:%S)" >>
process-stats.csv
    fi
    sleep 5
done
```

Folgend ein Screenshot des Colyseus Lasttest Programms:

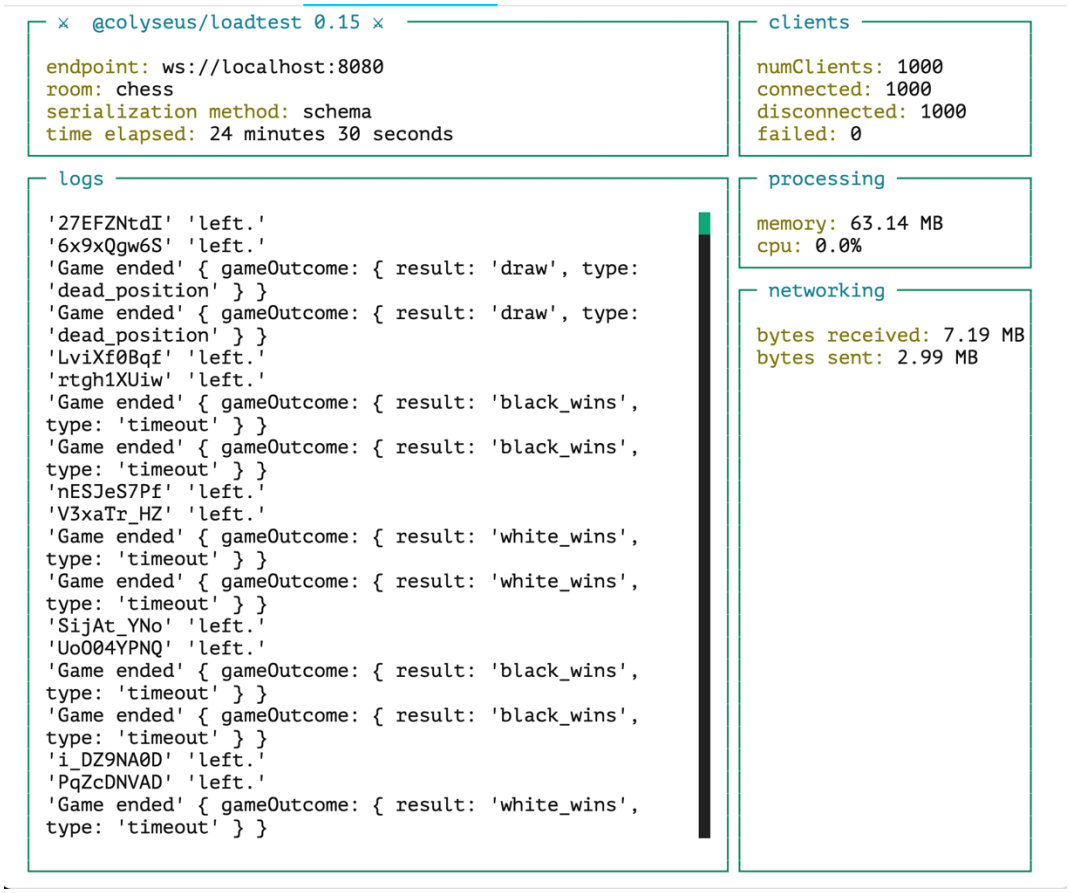

Abbildung 18 Colyseus Lasttest Programm

6.6.1.2 Auswertung

Der Lasttest ist erfolgreich durchgelaufen und alle Spiele konnten abgeschlossen werden. Die insgesamte Dauer, bis alle Spiele abgeschlossen wurden, betrug 24 Minuten und 25 Sekunden.

Folgende Auslastungswerte des Prozesses wurden gemessen:

Wert	**Minimum**	**Maximum**
CPU-Auslastung in %	0.7	24.2
Memory-Auslastung in %	1.3	6.9
Memory-Aulastung in MB	109.48	555.37

Es kann gesagt werden, dass der Lasttest ein Erfolg war. Die CPU-Auslastung war minimal angesichts der Belastung. Die Arbeitsspeicher Werte sind ebenfalls vertretbar. Der Gameserver könnte vermutlich eine sehr viel höhere Last ertragen. Mittels horizontaler Skalierung können potenziell zehntausende Spiele gleichzeitig gehandhabt werden. Aus den Daten wurden Graphen generiert, um die Ressourcennutzung zu visualisieren.

6.6.1.2.1 CPU-Auslastung Chart

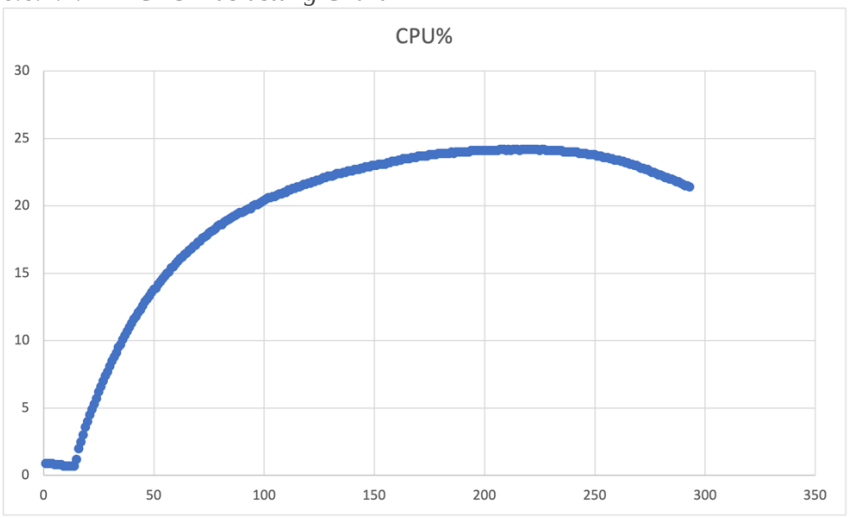

Abbildung 19 Lasttest Resultat Diagramm 1

6.6.1.2.2 Memory-Auslastung in % Chart

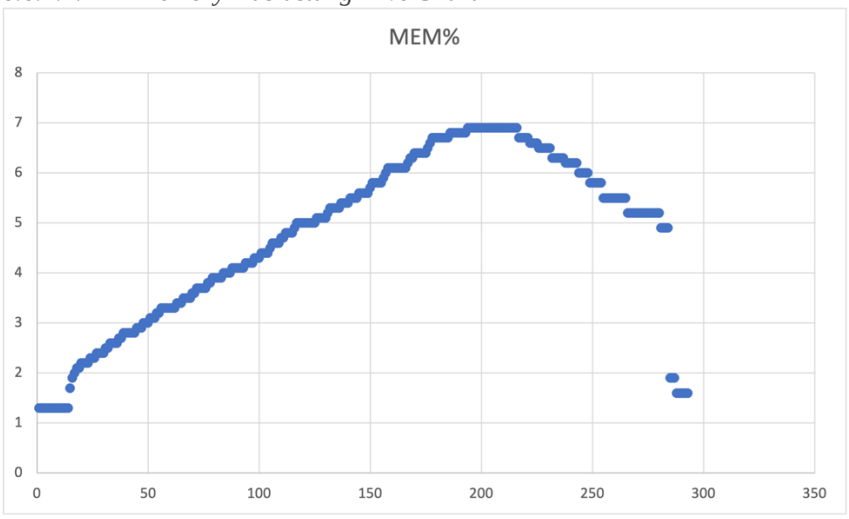

Abbildung 20 Lasttest Resultat Diagramm 2

6.6.1.2.3 Memory-Auslastung in Megabytes Chart

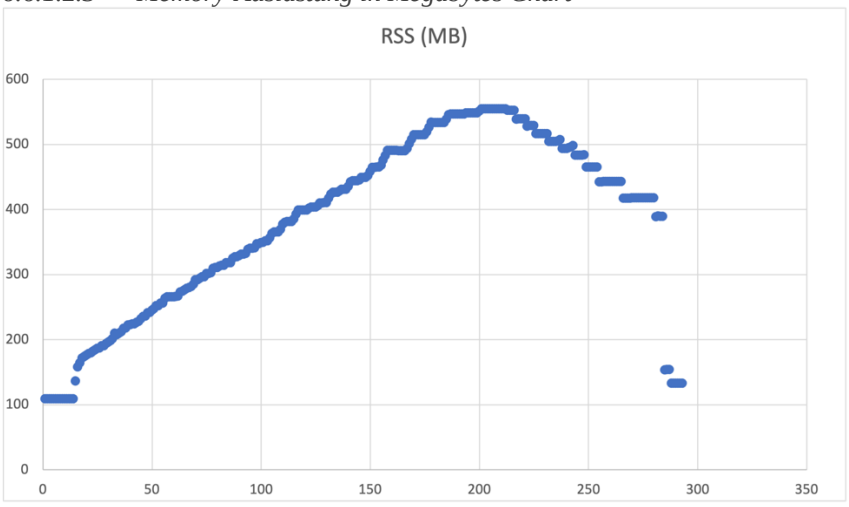

Abbildung 21 Lasttest Resultat Diagramm

7 Implementation Benutzerinterface

In diesem Kapitel wird die Next.js Applikation, also das Benutzerinterface umgesetzt. Folgend wird beschrieben, wie dabei vorgegangen ist, welche Gedanken gemacht und welche Technologien verwendet wurden.

7.1 Einführung in React und Next.js

Um dem Leser das Verständnis zu vereinfachen, wird eine Einführung in React und Next.js gemacht. Da Next.js selbst zahlreiche Funktionen anbietet und auf React aufbaut, ist es unmöglich hier sehr tief ins Detail zu gehen. Nichtsdestotrotz macht es Sinn einige Konzepte kurz zu erläutern.

7.1.1 React und CSR

React ist eine beliebte JavaScript-Bibliothek für die Erstellung von Benutzeroberflächen, insbesondere für Single-Page-Anwendungen (SPAs), die ein schnelles, interaktives Benutzererlebnis bieten sollen. React wird von Facebook entwickelt und gepflegt und ermöglicht es Entwicklern, interaktive und dynamische Webanwendungen zu erstellen, die als Reaktion auf Datenänderungen effizient aktualisiert und gerendert werden können.

Ein Vorteil von React liegt in seiner komponentenbasierten Architektur. Komponenten sind wiederverwendbare, isolierte Codestücke, die einen Teil der Benutzeroberfläche darstellen. Sie können ihren eigenen Zustand und ihre eigenen Parameter verwalten, wodurch React-Anwendungen modular und leicht zu warten sind. [58]

React läuft standardmässig ausschliesslich im Browser, was man clientseitiges Rendering nennt (CSR). Folgend werden ein paar Konzepte kurz erklärt, die für das Verständnis notwendig sind (für eine detaillierte Erklärung wird die offizielle Dokumentation von React empfohlen):

7.1.1.1 Zustand

```
const [counter, setCounter] = useState(0);
```

Um Zustand innerhalb von React Komponenten handzuhaben, gibt es den *setState* Hook. Sie gibt den Wert und eine Funktion, um den Wert zu verändern zurück. Wenn der Wert sich verändert, wird der jeweilige Komponent neu gerendert.

7.1.1.2 Komponentenlifecycle und Side Effects

useEffect ist ein Hook in React, der es ermöglicht, Nebeneffekte in funktionalen Komponenten auszuführen. Diese Nebeneffekte können Datenabrufe, manuelle DOM-Manipulationen oder andere Seiteneffekte sein. Der Hook wird nach dem Rendern der Komponente ausgeführt und kann auf Änderungen von bestimmten Werten (Abhängigkeiten) reagieren, die in einem Abhängigkeitsarray übergeben werden. Wenn bestimmte Abhängigkeiten sich ändern, wird der Code innerhalb von useEffect erneut ausgeführt. Um Ressourcen freizugeben oder Aufräumarbeiten vorzunehmen, kann useEffect eine Aufräumfunktion zurückgeben, die ausgeführt wird, bevor die Komponente unmountet oder bevor der Hook erneut aufgrund geänderter Abhängigkeiten ausgeführt wird.

```
useEffect(() => {
    // Funktion, die ausgeführt wird, wenn sich die Abhängigkeiten ändern
    return () => {
      // Funktion zum Aufräumen
    };
  }, [dependencies]);
```

7.1.1.3 Kontext

React Context bietet eine Möglichkeit, Daten durch den Komponentenbaum zu leiten, ohne sie als Props durch jede einzelne Komponente durchreichen zu müssen. Ein Context wird durch *React.createContext()* erstellt und hat zwei Hauptkomponenten: Provider und Consumer.

Der Provider wird verwendet, um den aktuellen Kontextwert für einen Komponentenbaum bereitzustellen. Alle Komponenten unterhalb des Providers können den Wert des Kontexts ohne explizite Übergabe nutzen.

Der Consumer wird in einer Komponente verwendet, die den Wert des Kontexts lesen möchte. In funktionalen Komponenten ist das häufig nicht notwendig, da der Hook useContext() denselben Zweck erfüllt und oft einfacher zu verwenden ist.

```
function App() {
  // Schritt 2: Einen Zustand für unser Thema erstellen
  const [theme, setTheme] = useState('dark');

  return (
    // Schritt 3: Den Provider verwenden, um den Wert für untergeordnete
Komponenten bereitzustellen
    <ThemeContext.Provider value={{ theme, setTheme }}>
      <Child />
    </ThemeContext.Provider>
  );
}

function Child() {
  // Schritt 4: useContext verwenden, um den Wert des Kontexts in einer
Kindkomponente zu lesen
  const { theme, setTheme } = useContext(ThemeContext);

  return (
    <div>
      Aktuelles Theme: {theme}
      <button onClick={() => setTheme(theme === 'light' ? 'dark' : 'light')}>
        Theme wechseln
      </button>
    </div>
  );
}
```

7.1.2 Next.js

7.1.2.1 Renderparadigmen

Next.js baut auf React auf, rendert die Applikation aber zuerst auf dem Server zu HTML und schickt dieses an den Benutzer, sodass er beim initialen Laden der Seite nicht lange warten muss, bis grosse JavaScript Dateien heruntergeladen wurden. Zu diesem Zeitpunkt ist die Applikation allerdings noch nicht interaktiv. Während der Benutzer bereits die Seite sieht, werden alle nötigen Scripts heruntergeladen und die Applikation wird im Hintergrund gestartet. Das Resultat ist ein äusserst schneller erster Ladevorgang im Vergleich zu rein clientseitigen Applikationen. Serverseitiges Rendering ist die Standardeinstellung von Next.js, aber es gibt noch mehr Renderingparadigmen. [59]

In diesem Projekt wurde nur eine dieser zusätzlichen Möglichkeit genutzt, nämlich *Servercomponents.* Servercomponents werden **ausschliesslich** auf dem Server gerendert. Das HTML, das der Komponent produziert, wird dann an den Client geschickt. Dies hat zum Vorteil, dass für Servercomponents kein JavaScript geladen werden muss und dass in Servercomponents beispielsweise direkt Datenbankabfragen gemacht werden können. Der Nachteil ist, dass keine interaktiven Elemente in Servercomponents vorkommen dürfen, weil dafür JavaScript benötigt wird. [60]

7.1.2.2 Routing

Das zweite Konzept, das für die Verständlichkeit erklärt werden muss, ist das Routing. Next.js verwendet ein Dateisystem-basiertes Routing. Seiten können definiert werden, indem im Ordner *app* Dateien erstellt werden. Der Pfad im Ordner entspricht der URL, über welche die Seite erreichbar ist. Für jede Seite muss eine *page.ts* Datei erstellt werden, die einen React Komponenten beinhaltet. Beispiel Ordnerstruktur:

/app
/layout.tsx
/page.tsx (/)
/profile
/page.tsx
/blog
/[id].tsx
/page.tsx

Im Beispiel hat es drei Seiten: Die Root-Seite, eine Profilseite und eine Blogseite mit einem dynamischen Parameter für die ID des Blogs. Die folgenden URLs werden generiert:

- /
- /profile
- /blog/<id>

Auf jeder Ebene können Layouts definiert werden (im Beispiel *layout.tsx*). Layouts gelten für alle Seiten das entsprechende Routensegment und können genutzt werden, um Komponenten anzuzeigen, die für jede Seite gleich sind, beispielsweise eine Navigation oder ein Footer. Bei Navigationen zwischen Seiten werden Layouts nicht neu gerendert und können auch einen Zustand zwischenspeichern, beispielsweise mit *useState* und React Kontext.

7.2 Setup

7.2.1 TailwindCSS

Um die Applikation zu stylen, wird TailwindCSS verwendet. Bei der Wahl dieser Technologie spielt vor allem die Präferenz des Autors eine Rolle. TailwindCSS ist eine Ansammlung von CSS-Hilfsklassen, die zusammengesetzt werden können, um jedes beliebige Design umzusetzen. Dafür muss der Autor selbst keine Zeile CSS schreiben, sondern nur die Klassen im Markup anwenden. TailwindCSS ist dazu hoch konfigurierbar und inkludiert nur die Hilfsklassen, die wirklich gebraucht wurden, wenn das Projekt gebaut wird. [61]

Ein Beispiel: `<div className="bg-red-500 p-4"></div>`
Dieses Element hat einen roten Hintergrund und ein *padding* von 4 Einheiten (eine Einheit ist standardmässig 16 Pixel).

7.2.2 RadixUI & Shadcn

Um die Entwicklung des Benutzerinterfaces etwas einfacher zu gestalten, wird eine React Komponentenbibliothek namens *RadixUI* verwendet. RadixUI stellt eine breite Palette von Komponenten zur Verfügung, die in Webprojekte üblicherweise gebraucht werden, beispielsweise ein Dialog, ein Navigationsmenü oder ein Formular. Solche Komponenten selbst umzusetzen kann äusserst aufwändig sein. RadixUI's Komponenten sind nicht gestyled, sondern stellen nur die Funktionalität dahinter zur Verfügung. Das Styling muss vom Entwickler selbst übernommen werden. Hier kommt *Shadcn* ins Spiel – Shadcn basiert auf RadixUI Komponenten, hat aber Standardstyles bereits angewendet. Die Shadcn Komponenten sind mit TailwindCSS gestyled, was wunderbar in das Ökosystem des Projekts passt.

Hier ein Beispiel wie Shadcn aussieht, übernommen von der offiziellen Dokumentation:

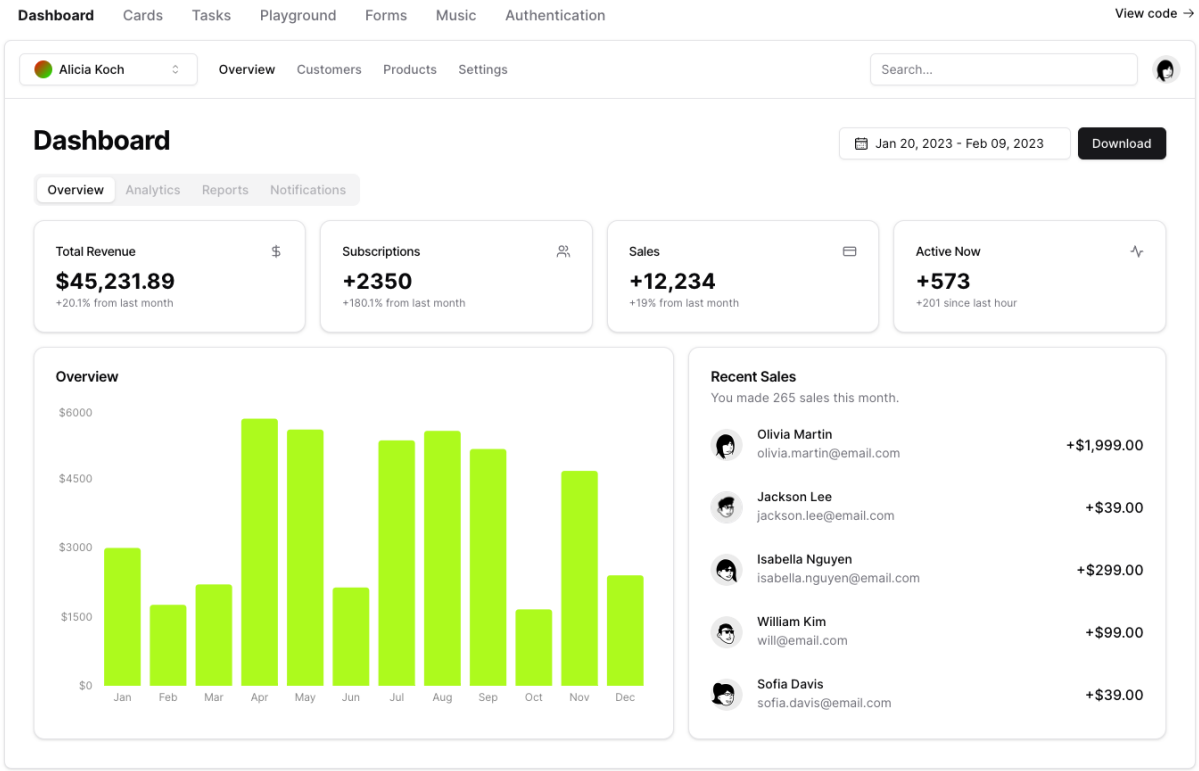

Abbildung 22 Shadcn UI Demonstration

Die Komponenten sehen professionell aus, sind aber trotzdem unvoreingenommen. Das Verwenden dieser Bibliothek ermöglicht es dem Autor, sich auf die wesentliche Funktionalität zu konzentrieren,

anstatt sich in stilistischen Verfangen zu verlieren. Die Komponenten sind ausserdem auch sehr einfach anpassbar.

7.2.3 Prisma Integration

Da der Prisma Client bereits generiert wurde, kann dieser im Projekt nun genutzt werden, um Datenbank Abfragen zu machen. Allerdings muss dabei bei Next.js Projekten folgendes beachtet werden: Next.js nutzt im Entwicklungsmodus Hot-Module-Reloading (HMR), was dafür dient, dass Änderungen am Code direkt reflektiert werden, ohne dass der Next.js Server neugestartet werden muss und ohne dass die Seite neugeladen werden muss. Das kann mit Prisma zusammen Probleme machen, denn es wird jedes Mal ein neuer Prisma Client instanziiert, wird eine neue Verbindung zur Datenbank erstellt. Somit kann die Datenbank schnell mit zu vielen Verbindungen überlastet werden. [62]

Um dies zu umgehen, wird der Prisma Client im globalen Kontext von Node.js zwischengespeichert:

```
import { PrismaClient } from "ks-database";

const globalForPrisma = globalThis as unknown as {
  prisma: PrismaClient | undefined;
};

export const prisma = globalForPrisma.prisma ?? new PrismaClient();

if (process.env.NODE_ENV !== "production") globalForPrisma.prisma = prisma;
```

Durch diesen Code wird der Prisma Client nun auch bei HMR-Aktualisierungen wiederverwendet.

7.2.4 NextAuth

Um Authentifikation nach der Definition im Kapitel «Analyse» zu gewährleisten, wird die Bibliothek *NextAuth* eingesetzt. NextAuth implementiert den OpenIDConnect Standard und ist somit für die Plattform optimal geeignet. Zudem wird die Bibliothek vom Techgiganten *Vercel* unterstützt und bietet dadurch eine hohe Zuverlässigkeit. Im folgenden Abschnitt wird die Bibliothek und die Authentifikation via GitHub integriert. NextAuth bietet sogar eine direkte Integration mit Prisma an, was für den Entwickler sehr komfortabel ist.

7.2.4.1 Setup

Das Setup ist simpel. Die Bibliothek muss zuerst installiert werden. Anschliessend kann eine Konfigurationsdatei erstellt werden:

```
export const authOptions: AuthOptions = {
  providers: [
    GitHubProvider({
      clientId: "<<github client id>>",
      clientSecret: "<<github client secret>>",
    }),
  ],
  adapter: PrismaAdapter(prisma) as any,
  session: { strategy: "jwt" },
};
```

In der Konfiguration wird ein GithubProvider erstellt, der von NextAuth zur Verfügung gestellt wird. Darin muss eine Client ID und ein Client Secret definiert sein, welche auf GitHub im Benutzerprofil

generiert werden können. Zusätzlich wird ein *PrismaAdapter* benötigt, der die Prisma Client Instanz entgegennimmt und somit die Datenbank automatisch verknüpft.

Als nächstes müssen API-Routen definiert werden. Diese werden dafür verwendet die Login-Seite darzustellen und weitere OpenIDConnect spezifische Endpunkte zur Verfügung zu stellen.

Wie bereits erwähnt werden Routen über das Dateisystem definiert. Im Pfad */app/api/auth/[…nextauth]* wird eine Datei route.ts erstellt, mit folgendem Inhalt:

```
import NextAuth from "next-auth";
import { authOptions } from "@/lib/auth";

const handler = NextAuth(authOptions as any);
export { handler as GET, handler as POST };
```

Der NextAuth Route Handler nimmt nun das Objekt, das im vorherigen Schritt definiert wurde als Parameter entgegen.

Als letzter Schritt müssen Tabellen für NextAuth in der Datenbank generiert werden. Glücklicherweise bietet NextAuth in der Dokumentation ein fertiges Prisma Schema an, das in das bestehende Schema integriert werden kann. Das Schema wird um drei Tabellen erweitert:

```
model Account {
  id                       String  @id @default(cuid())
  userId                   String
  type                     String
  provider                 String
  providerAccountId        String
  refresh_token            String? @db.Text
  refresh_token_expires_in Int
  access_token             String? @db.Text
  expires_at               Int?
  token_type               String?
  scope                    String?
  id_token                 String? @db.Text
  session_state            String?

  user User @relation(fields: [userId], references: [id], onDelete: Cascade)

  @@unique([provider, providerAccountId])
}

model Session {
  id           String   @id @default(cuid())
  sessionToken String   @unique
  userId       String
  expires      DateTime
  user         User     @relation(fields: [userId], references: [id], onDelete:
Cascade)
}
```

```
model VerificationToken {
  identifier String
  token      String   @unique
  expires    DateTime

  @@unique([identifier, token])
}
```

Nun ist das Setup schon komplett. Wenn alles geklappt hat, kann man im Browser auf die Seite /api/auth/signin navigieren, wo folgende Seite dargestellt wird:

Abbildung 23 Login Seite mit NextAuth und GitHub

Da nur GitHub als Anbieter konfiguriert wurde, wird nur ein Button angezeigt (es könnten aber mehrere Anbieter konfiguriert werden). Per Klick auf den Button wird der Benutzer zu GitHub weitergeleitet, wo er sich anmelden muss, um dann zurück zur Applikation geleitet zu werden.

7.2.4.2 Ablaufdiagramm NextAuth

Hier ist visualisiert, wie NextAuth Authentifizierung handhabt. Dieses Diagramm stammt von der offiziellen NextAuth Dokumentation. Wichtig ist dabei, dass NextAuth nachdem der User sich erfolgreich bei GitHub authentifiziert, selbst eine Session generiert und ein Access Token ausstellt und nicht das Access Token von GitHub verwendet für den Gebrauch innerhalb der Applikation. Gerade wenn mehrere Anbieter konfiguriert sind, macht dieses Vorgehen Sinn. Zudem besteht Möglichkeit zusätzliche, applikationsspezifische Informationen im Token zu speichern und die Gültigkeit des Tokens selbst zu kontrollieren.

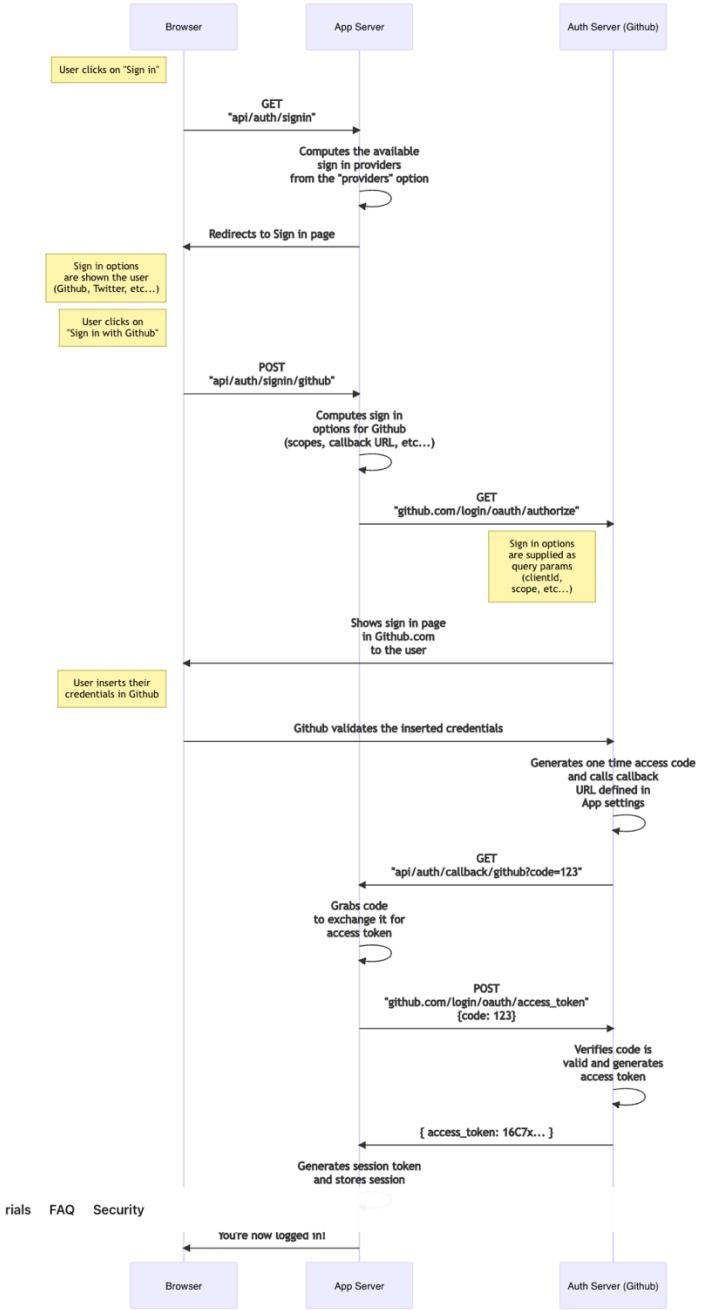

Abbildung 24 NextAuth Ablaufdiagramm [63]

7.3 Implementation

Zur Implementation muss gesagt werden, dass Programmcodeabschnitte in diesem Dokument jeweils stark vereinfacht und auf das Wesentliche beschränkt wurden. Die vollständige Implementation ist im Quellcode ersichtlich.

7.3.1 Komponentenbaum

Anhand der Wireframes und mittels des Routings von Next.js können Seiten und Komponenten grob definiert werden. Dies wird in folgender Grafik visualisiert:

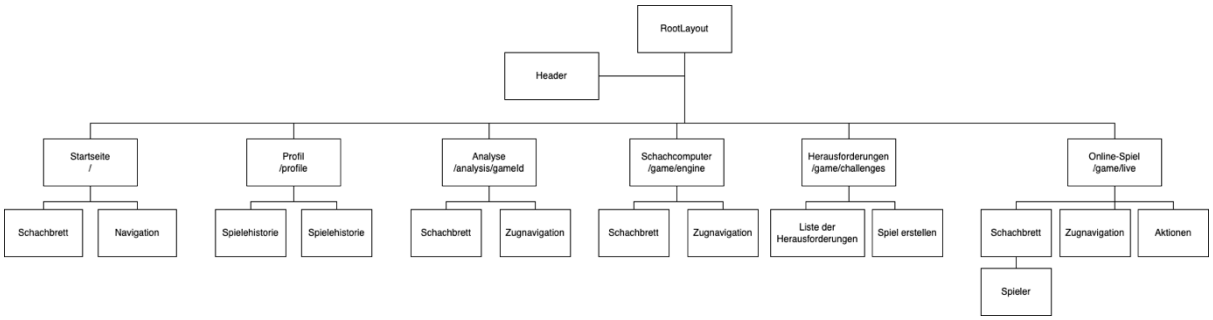

Abbildung 25 Benutzerinterface Komponentenbaum

Die Grafik dient als Anhaltspunkt für die Entwicklung und ist nicht unbedingt vollständig oder hundert Prozent korrekt.

7.3.2 Allgemeine Elemente

In Next.js ist ein Rootlayout zwingend nötig. Das Rootlayout ist ein Servercomponent und rendert nur den *Header* und die jeweilige Seite (*children*), auf der sich der Benutzer aktuell befindet.

```
export default function RootLayout({
  children,
}: {
  children: React.ReactNode;
}) {
  return (
    <html lang="en">
      <body className="bg-neutral-50">
        <Header />
        <div className="container mx-auto">{children}</div>
      </body>
    </html>
  );
}
```

Ebenso simpel ist die *Header* Komponente. Diese wird aus Gründen der Ausführlichkeit nicht aufgeführt. Die Komponente zeigt einen Login Button an, falls der Benutzer nicht eingeloggt ist und den Avatar (GitHub Avatar, wird in der Session bereitgestellt und kann ausgelesen werden) falls doch. Zusätzlich wird mittels Komponenten von Shadcn eine Navigation angezeigt für schnellen Zugriff auf die anderen Seiten.

7.3.3 BoardContext

Um die Zuggenerator Logik in die Next.js Applikation einzubauen, wird eine Komponente *BoardContext* erstellt. Diese Komponente dient dazu den Zustand des Zuggenerators (Position der Figuren, aktive Farbe, legale Züge) in React Zustand zu speichern und für Kinderkomponenten zur Verfügung zu stellen.

Zunächst wird ein Kontext erstellt:

```
export const BoardContext = React.createContext<BoardStateValues>({});
```

Anschliessend kann die *BoardContext* Komponente erstellt werden:

```
export default function BoardState({ children }: BoardStateProps) {
  const board = useMemo(() => ChessBoard.fromInitialPosition(), []);

  return (
    <BoardContext.Provider
      value={
        {
          // TODO
        }
      }
    >
      {children}
    </BoardContext.Provider>
  );
}
```

Die Komponente instanziiert den Zuggenerator von der Anfangsposition an. Nun müssen die Werte, die der Zuggenerator zur Verfügung stellt im React Zustand gespeichert und jedes Mal, wenn ein Zug gemacht wird, aktualisiert werden:

```
const [activeColor, setActiveColor] = useState<Color>(board.activeColor);
const [legalMoves, setLegalMoves] = useState<Array<Move>>(board.legalMoves);
const [boardSquares, setBoardSquares] = useState(board.boardSquares);
const [capturedPieces, setCapturedPieces] = useState(board.capturedPieces);

useEffect(() => {
  board.moveCallback = (boardState) => {
    setLegalMoves(boardState.legalMoves);
    setActiveColor(boardState.activeColor);
    setBoardSquares(boardState.boardSquares);
    setCapturedPieces(boardState.capturedPieces);
  };
}, [board]);

const makeMove = (move: Move | string) => {
  board.makeMove(move);
};
```

Es wird zusätzlich eine Funktion *makeMove* definiert, die dem Zuggenerator einen Zug übergibt. Alle Werte werden nun dem Kontext-Provider mitgegeben, sodass Kinderkomponenten diese verwenden können.

```
return (
  <BoardContext.Provider
    value={{
      activeColor,
      boardSquares,
      makeMove,
      legalMoves,
    }}
  >
    {children}
  </BoardContext.Provider>
);
```

Kinderkomponenten vom BoardContext können nun ganz einfach auf die Informationen zugreifen.

7.3.4 Brett

Um die Spiellogik im Benutzerinterface abzubilden, wird ein Brett benötigt. Dazu gehören folgende Komponenten:

- Eine *Board* Komponente, die das Brett darstellt
- Eine *Square* Komponente, die ein Feld repräsentiert
- Komponenten für Figuren aller Art und für beide Farben

7.3.4.1 Figuren

Um die Figuren abzubilden, werden Grafiken benötigt. *Wikimedia Commons*, eine Ansammlung von frei nutzbaren Medien, bietet eine Palette von allen Schachfiguren in Form von SVG-Dateien an [28]. Das ist für diesen Anwendungsfall praktisch.

```
const Bishop = forwardRef<HTMLDivElement, PieceProps>(
  ({ pieceColor: color, ...rest }, ref) => {
    return (
      <div {...rest} ref={ref}>
        {color === Color.White ? (
          <svg>{/* Figur Weiss */}</svg>
        ) : (
          <svg>{/* Figur Schwarz */}</svg>
        )}
      </div>
    );
  }
);
```

Für jede Figur wurde eine Komponente erstellt, die abhängig von der Farbe das SVG für Schwarz oder Weiss darstellt.
Zusätzlich wurde eine Komponente erstellt, die anhand der Repräsentation der Figur in der Mailbox, welche der Zuggenerator zur Verfügung stellt, die korrekte Figur in der richtigen Farbe anzeigt.

```
const resolvePiece = (piece: string) => {
  switch (piece.toUpperCase()) {
    case "K":
      return King;
    case "Q":
```

```
      return Queen;
    case "R":
      return Rook;
    case "P":
      return Pawn;
    case "N":
      return Knight;
    case "B":
      return Bishop;
    default:
      return null;
  }
};

export const Piece = (({ piece }) => {
  const color = getPieceColor(piece);

  const InternalPiece = resolvePiece(piece);

  return InternalPiece && color !== null ? (
    <InternalPiece pieceColor={color} />
  ) : null;
});
```

7.3.4.2 Feld

Um die Figuren anzuzeigen, muss eine Komponente, die ein Feld repräsentiert, erstellt werden. Ein Feld erhält als Parameter den Index von 0-63 und die Repräsentation der Figur, die darauf steht. Wenn eine Figur auf dem Feld steht, wird die *Piece* Komponente gerendert. Anhand des Index des Felds kann die Farbe bestimmt werden.

```
function Square({ index, piece }: Props) {
  const color = getSquareColor(index);
  return (
    <div
      className={clsx({
        "bg-indigo-500": color === Color.Black,
        "bg-indigo-100": color === Color.White,
      })}
    >
      <div className="relative w-full h-full">
        {typeof piece === "string" && (
          <Piece
            squareIndex={index}
            piece={piece}
            className="absolute inset-0 z-10"
          />
        )}
      </div>
    </div>
  );
}
```

7.3.4.3 Brett

Um nun alles zusammenzufügen, kann eine *Board* Komponente erstellt werden. Diese erhält die Mailbox aus dem BoardContext als Parameter und rendert 64 Felder.

```
export default function Board({
  flipped,
  boardSquares,
}: Props) {
  return (
    <div>
      {boardSquares.map((_, i) => {
        const index = flipped ? 63 - i : i;
        return (
          <InteractiveSquare
            piece={boardSquares[index]}
            key={index}
            index={index}
          />
        );
      })}
    </div>
  );
}
```

Damit stellt das Brett nun die korrekte Aufstellung dar, aber ist noch nicht interaktiv.

7.3.4.4 Interaktivität

Um Züge zu machen, müssen Figuren auf dem Brett verschoben werden können. Dafür wird die Bibliothek *dndkit* verwendet, die einfache Drag & Drop Funktionalität für React zur Verfügung stellt.

Die Bibliothek teilt Komponenten auf in *Draggables* und *Droppables*. Draggables sind in diesem Fall die Figuren und Droppables die Felder.

Die *Square* Komponente wird erweitert mit dem *useDroppable* Hook von dndkit:

```
const id = useId();
const { setNodeRef, isOver } = useDroppable({
  id,
  data: {
    squareIndex: index,
  },
});
```

Mit der Funktion *setNodeRef* wird das Element ausgewählt, das als Droppable fungieren soll.

Nun können die Figuren mit dem *useDraggable* Hook zu Draggables gemacht werden. Dafür wird eine neue Komponente gemacht, die die *Piece* Komponente verpackt:

```
function DraggablePiece({ squareIndex, piece, className }: PieceProps) {
  const id = useId();

  const { attributes, listeners, setNodeRef, isDragging } = useDraggable({
```

```
    id,
    data: {
      fromSquare: squareIndex,
      piece,
    },
  });

  return (
    <Piece
      ref={setNodeRef}
      piece={piece}
      className={cn(className, "cursor-grab", {
        "opacity-0": isDragging,
      })}
      {...attributes}
      {...listeners}
    ></Piece>
  );
}
```

Als letztes muss die *Board* Komponente mit einem *DndContext* verpackt werden. Dort können Eventhandler definiert werden für wenn ein Draggable gezogen oder abgelegt wird.

```
<DndContext onDragEnd={handleDragEnd}>
    // Inhalt Board Component
</DndContext>
```

Der *handleDragEnd* Eventhandler überprüft, auf welches Droppable das Draggable gezogen wurde. Mit diesen Informationen kann dann aus der Liste der legalen Züge der korrespondierende Zug gefunden werden.

```
function handleDragEnd(event: DragEndEvent) {
  const fromSquare: number = event.active.data.current?.fromSquare;
  const targetSquare: number = event.over?.data.current?.squareIndex;
  if (
    typeof fromSquare !== "undefined" &&
    typeof targetSquare !== "undefined" &&
    fromSquare !== targetSquare
  ) {
    const currentSquareLegalMoves = legalMoves.filter(
      (move) =>
        move.sourceSquare === fromSquare && move.targetSquare === targetSquare
    );

    if (currentSquareLegalMoves.length === 1) {
      onMove(currentSquareLegalMoves[0]);
    }
  }
}
```

Wird ein gültiger Zug gefunden, wird dieser an den Zuggenerator übergeben, welcher ihn ausführt, dann die Mailbox aktualisiert, was wiederum dazu führt, dass die neue Position in der *Board* Komponente widerspiegelt, wird. Damit ist das Brett im Benutzerinterface nun interaktiv.

7.3.4.5 Weitere Funktionalitäten und Komponenten

In den letzten Abschnitten wurde nur beschrieben, wie das Brett grundsätzlich aufgebaut wurde. In Realität wurden noch mehr Arbeiten gemacht, die hier nicht speziell ausgewiesen werden. Diese Arbeiten beinhalten:

- Es wird ein Koordinatensystem angezeigt, das die Reihen von 1-8 und die Linien von A-H beschriftet.
- Das Brett kann umgedreht werden, sodass der schwarze Spieler unten ist.
- Wenn beim Loslassen einer Figur über einem Feld mehrere legale Züge mit dem gleichen Ursprungs- und Zielfeld gefunden werden, handelt es sich um eine Beförderung. In diesem Fall wird ein Dialog angezeigt, der den Benutzer die Figur auswählen lässt, zu welcher er den Bauern befördern möchte.
- Legale Züge werden auf dem Brett markiert. Felder, auf welche gezogen werden kann, zeigen einen Indikator an, sobald man die jeweilige Figur anklickt.

7.3.4.6 Anwendung

Die erstellten Komponenten können zusammengesetzt werden, um das Brett auf einer Seite anzuzeigen. Das sieht in etwa so aus:

```
<BoardProvider>
  <BoardContext.Consumer>
    {({ boardSquares, legalMoves, makeMove, activeColor }) => {
      return (
        <Board
          activeColor={activeColor}
          boardSquares={boardSquares}
          legalMoves={legalMoves}
          onMove={makeMove}
        />
      );
    }}
  </BoardContext.Consumer>
</BoardProvider>
```

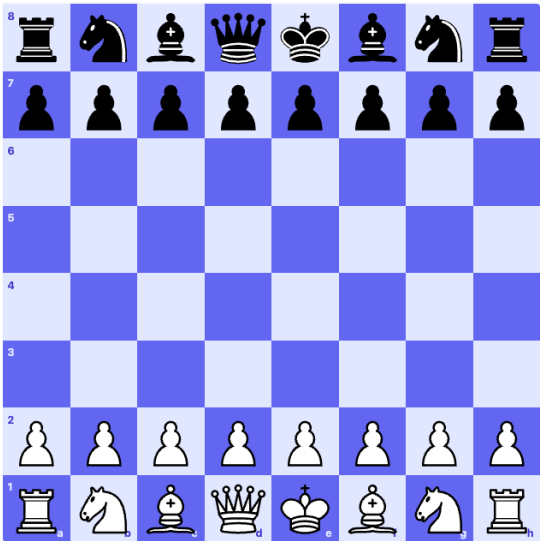

Abbildung 26 Board Komponente

7.3.5 Zugnavigation

Damit man in Spielen über die gemachten Züge navigieren kann, wird eine Komponente benötigt, die ein Interface dafür zur Verfügung stellt. Dafür muss zuerst der *BoardContext* erweitert werden. Neu werden die gemachten Züge und die Position, die daraus entstand (der Zuggenerator stellt diese Information zur Verfügung), sowie der Zug, den der Benutzer in der Zugnavigation auswählt im Zustand gespeichert.

```
const [moveHistory, setMoveHistory] = useState(board.moveHistory);
const [positionHistory, setPositionHistory] = useState(board.positionHistory);
const [selectedMove, setSelectedMove] = useState<Move | null>(null);
```

Anstatt die Mailbox vom Kontext direkt für die Kinderkomponenten zur Verfügung zu stellen, wird zuerst überprüft, ob der Benutzer die aktuelle Position sehen möchte oder eine vergangene, anhand davon, ob der *selectedMove* gesetzt ist.

```
const currentPosition = useMemo(() => {
  if (!selectedMove) return boardSquares;

  const positionIndex = moveHistory.indexOf(selectedMove);
  return parseFEN(positionHistory[positionIndex]).boardSquares;
}, [selectedMove, boardSquares, moveHistory, positionHistory]);

const viewPositionAfterMove = (move: Move) => {
  if (moveHistory.indexOf(move) === moveHistory.length - 1) {
    setSelectedMove(null);
  } else {
    setSelectedMove(move);
  }
};
```

Wenn ein Zug ausgewählt wurde, wird die Position zurückgegeben, die aus dem Zug resultierte und wenn nicht die aktuelle Position. Zusätzlich wurden Funktionen hinzugefügt, um zum nächsten oder zum vorherigen Zug zu springen. Diese Informationen werden nun zusammen, mit den bisherigen Informationen über den Kontext verteilt.

```
<BoardContext.Provider
  value={{
    // ...
    boardSquares: currentPosition,
    selectedMove: selectedMove ?? moveHistory[moveHistory.length - 1],
    viewPositionAfterMove,
    viewPreviousPosition,
    viewNextPosition,
  }}
>
  {children}
</BoardContext.Provider>
```

Damit kann die Zugnavigation Komponente gebaut werden. Diese wird hier nicht speziell ausgeführt. Sie zeigt alle Züge des aktuellen Spiels an und Benutzer können auf einen Zug klicken, um die dazugehörige Position anzuzeigen. Zusätzlich gibt es Buttons um zum ersten, zum letzten, zum vorherigen und zum nächsten Zug zu springen.

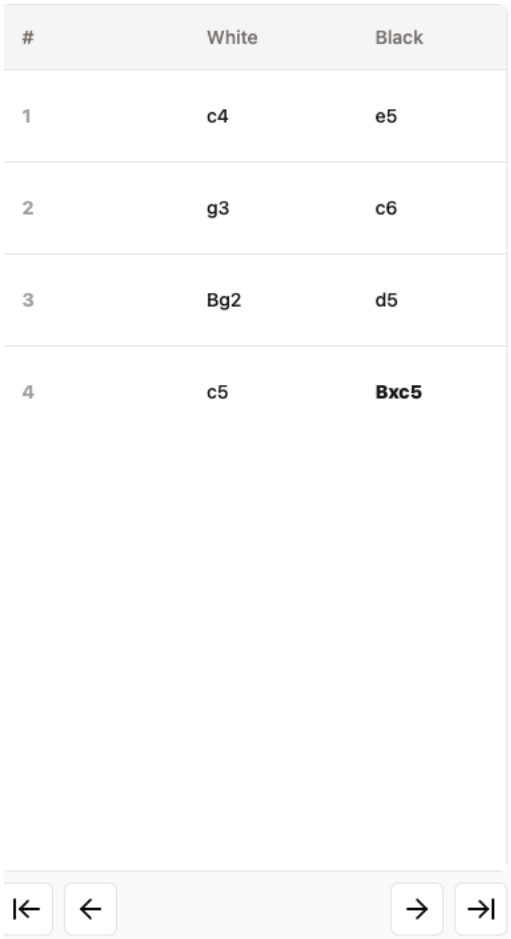

Abbildung 27 Zugnavigation Komponente

7.3.6 Colyseus Verbindung

Da alle Komponenten, die es für ein Spiel braucht, erstellt sind, kann nun Colyseus angehängt werden. Colyseus stellt eine Bibliothek zur Verfügung, um die Verbindung zum Server zu vereinfachen. Um die Bibliothek einzusetzen wird ein neuer Kontext erstellt. Dieser instanziiert die Bibliothek und macht sie für die React Komponenten verfügbar.

```
export const ColyseusContext = React.createContext<ColyseusContextState>({});
```

Dazu gehört die Provider Komponente:

```
export default function ColyseusProvider({ children }) {
  const colyseus = useRef(new Client("ws://localhost:8080"));
  const [room, setRoom] = useState<Room<ChessRoomSchema> | null>(null);

  return (
    <ColyseusContext.Provider
      value={{
        colyseusClient: colyseus.current,
      }}
    >
      {children}
    </ColyseusContext.Provider>
  );
```

```
}
```

Wie man sieht, wird ein Colyseus Client instanziiert mit der URL zum Server über das Websockets Protokoll. Mit dem Client können nun Funktionen erstellt werden, um einen Raum zu erstellen, beizutreten oder offene Räume abzufragen:

```
const joinRoom = async (roomId: string) => {
  const connectedRoom = await colyseus.current.joinById<ChessRoomState>(
    roomId,
  );
  setRoom(connectedRoom);
};

const createRoom = async (args?: { time: number; increment?: number }) => {
  const createdRoom = await colyseus.current.create<ChessRoomState>("chess", {
    timeControl: args ? { ...args } : undefined,
  });
  setRoom(createdRoom);
};

const getAvailableRooms = () => {
  return colyseus.current.getAvailableRooms("chess");
};
```

7.3.6.1 Colyseus Authentication

Initial hatte der Autor geplant, das Access Token, das NextAuth ausstellt zu verwenden, um sich an Colyseus zu authentifizieren. Es hat sich aber herausgestellt, dass die Autoren von NextAuth dieses Vorgehen aus Sicherheitsgründen nicht empfehlen. [64] Deshalb hat der Autor folgende Lösung implementiert: Wenn ein Client sich verbinden möchte, also ein Spiel erstellt oder einem beitritt, wird eine HTTP-Abfrage an einen Next.js API-Endpunkt gesendet. Dieser Endpunkt signiert ein Access Token mit einem Schlüssel. Das Token enthält Informationen zum Benutzer und wird beim Verbindungsaufbau mit dem Colyseus Server mitgeschickt, welcher es mit demselben Schlüssel wieder entschlüsseln und verifizieren kann. Die Implementation davon befindet sich in der Dokumentation des Gameservers. Da es sich um eine Verbindung mit Websockets handelt, welche nach dem Öffnen bestehen bleibt, hat das Token eine Gültigkeit von nur zwei Minuten (es wird nur beim Verbindungsaufbau gebraucht). Dieser Ansatz funktioniert gut und ist simpel.

Die Implementation des Next.js API-Endpunkt, der das Token generiert ist wie folgt.

```
export async function POST() {
  const session = await getServerSession(authOptions);
  const secret = process.env.TOKEN_SECRET!;

  if (!session) {
    const payload = { anonymous: true };
    const accessToken = jwt.sign(payload, secret, {
      expiresIn: EXPIRES_IN,
    });

    return NextResponse.json({ accessToken });
  }
```

```
  const user = await prisma.user.findFirst({
    where: {
      email: session?.user?.email,
    },
  });

  if (!user) {
    return new Response("Forbidden", {
      status: 403,
    });
  }

  const payload = { userId: user.id };
  const accessToken = jwt.sign(payload, process.env.TOKEN_SECRET as string, {
    expiresIn: EXPIRES_IN,
  });

  return NextResponse.json({ accessToken });
}
```

Wenn der Benutzer angemeldet, also nicht anonym ist, wird der entsprechende Datensatz aus der Datenbank gelesen. Anschliessend wird das Token mit dem Schlüssel und der ID des Benutzers als Inhalt signiert und vom Endpunkt zurückgegeben. Im Falle eines anonymen Benutzers wird auch ein Token signiert, aber der Inhalt ist nur *anonymous: true*. Der Gameserver, der das Token erhält, weiss somit welcher Benutzer, dass die Verbindung aufbauen möchte, oder ob er anonym ist.

7.3.7 Herausforderungen

Um ein neues Spiel zu starten, wird nun die Herausforderungen Seite erstellt. Über diese können Benutzer ein Spiel erstellen, oder einem bestehenden beitreten. Damit Zugriff auf den Colyseus Kontext möglich ist, muss die Seite im Kontext Provider verpackt sein.

Die Herausforderungsseite wird eingeteilt in zwei Komponenten:

- Liste der Herausforderungen
- Spiel erstellen

7.3.7.1 Spiel erstellen

Die *CreateChallenge* Komponente, greift via Kontext auf den Colyseus Client zu und zeigt eine Liste von Buttons an, die jeweils ein Spiel mit einem bestimmten Zeitformat erstellen. Dafür wird die im Kontext definierte Funktion *createRoom* verwendet. Der Code wird hier nicht ausgewiesen, da die Komponente nur ein paar Buttons anzeigt, die die Funktion aufrufen.

7.3.7.2 Spiel beitreten

Die Komponente, um Spielen beizutreten ist ebenso simpel. Die Räume werden abgefragt mit der im Kontext definierten *getAvailableRooms* Funktion und die Resultate in einer Liste angezeigt. Für jeden Listeneintrag wird auch ein Button angezeigt, welcher die *joinRoom* Funktion aus dem Kontext aufruft.

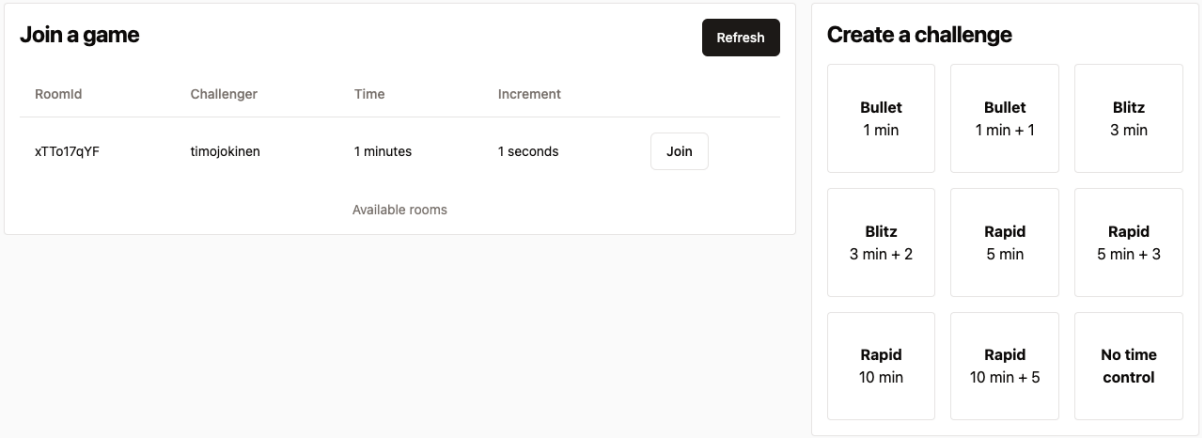

Abbildung 28 Herausforderungsseite

7.3.8 Online-Spiel

Da die Spieler nun gepaart werden können (Spieler 1 erstellt ein Spiel und Spieler 2 tritt bei) kann nun die Seite für Online-Spiele erstellt werden.

Die Seite wird unter dem URL-Pfad /game/live/<raumId > erstellt und ist vorerst leer.

7.3.8.1 Weiterleitung zum Spiel

Um die Spieler auf die Seite weiterzuleiten, wenn das Spiel beginnt, kann der Colyseus Kontext erweitert werden:

```
useEffect(() => {
  let unlisten = room?.state.listen("ready", (ready) => {
    if (ready) {
      router.push("/game/live/" + room.roomId);
      unlisten?.();
    }
  });
  return () => {
    unlisten?.();
  };
}, [room, router]);
```

Die Client Bibliothek von Colyseus bietet Listeners an, wenn Attribute im Colyseus Zustand ändern. In der Colyseus Dokumentation in dieser Arbeit wurde beschrieben, dass das *ready* Attribut im Zustand auf wahr gesetzt wird, sobald das Spiel initialisiert wurde. Im Code wird deshalb ein Eventhandler auf dieses Attribut erstellt. Die Benutzer werden somit, sobald das Spiel von Colyseus initialisiert wurde, via Next.js Router zur Online-Spiel Seite weitergeleitet.

7.3.8.2 Online-Spiel Seite

Nun kann die Seite implementiert werden. Dafür wird eine Komponente *LiveMatch*, verpackt im *BoardContext* erstellt. Diese Komponente dient dazu, mit Colyseus zu kommunizieren, Nachrichten zu senden und auf Nachrichten zu hören.

Implementation der Online-Spiel-Seite:

```
export default async function Game() {
  return (
    <div>
      <BoardContext>
        <LiveMatch />
      </BoardContext>
    </div>
  );
}
```

Die *LiveMatch* Komponente stellt das Brett und die Zugnavigation dar und erhält vom *BoardContext* und *ColyseusContext* die nötigen Informationen:

```
const { room, player, opponent } = useContext(ColyseusContext);
const { legalMoves, makeMove, boardSquares, activeColor } =
    useContext(BoardContext);
```

Die Komponente reagiert auf Züge des Gegners, indem auf die Nachricht *opponent_move*, die im Gameserver als ausgehende Nachricht definiert wurde und widerspiegelt den Zug im lokalen Zuggenerator:

```
useEffect(() => {
  let unlistenOpponentMove = room?.onMessage<{ move: string }>(
    ServerMessage.OpponentMove,
    ({ move }) => {
      makeMove(move);
    }
  );

  return () => {
    unlistenOpponentMove?.();
  };
}, [makeMove, room]);
```

Wenn der Benutzer einen Zug macht, wird validiert, ob er am Zug ist und sendet anschliessend die Nachricht *make_move*, die als eingehende Nachricht im Gameserver definiert wurde. Der Zug wird auch im Zuggenerator angewendet. Somit ist das Brett mit Colyseus synchronisiert.

```
const handleMove = (move: Move): void => {
  if (getPieceColor(move.piece) !== activeColor) return;
  if (player?.color !== activeColor) return;
  makeMove(move);
  room?.send(ClientMessage.Move, { move: move.toString() });
};
```

7.3.8.3 Weitere Arbeiten

Erneut, um die Dokumentation verständlich und übersichtlich zu halten, wurde nicht der gesamte Programmcode ausgeführt. Weitere Arbeiten beinhalteten:

- Wenn der Server die Nachricht *conclude_game* sendet und das Spiel somit beendet wurde, wird das den Spielern angezeigt.
- Das Brett wird für den Spieler, der die schwarzen Figuren kontrolliert, umgedreht.

- Informationen zu den Spielern werden angezeigt. Diese Information befindet sich im Zustand des Gameservers und wird im *ColyseusContext* gespeichert.
- Es werden Buttons zum resignieren und Remis anbieten angezeigt, die die jeweiligen Nachrichten an den Server schicken
- Wenn ein Remis angeboten wurde, wird dem jeweiligen Spieler ein Button angezeigt, über welchen er annehmen kann.
- Es wird angezeigt, welcher Spieler welche Figuren geschlagen hat. Diese Information stellt der Zuggenerator zur Verfügung.

Somit können Online-Spiele nun abgewickelt werden, mitsamt allen Funktionalitäten.

7.3.9 Schachuhr

Die Schachuhr fehlt noch, welche in Online-Spielen angezeigt werden muss. Der Gameserver aktualisiert die verbleibende Zeit pro Spieler nach jedem Zug im Zustand. Währenddessen muss aber im Browser eine Uhr laufen, die optimalerweise synchron mit der Uhr auf dem Server ist.

Dafür wurde eine neue Komponente *Clock* erstellt.

```
export default function Clock({ timeRemaining, color }: Props) {
  return (
    <div
      className={clsx("border rounded text-4xl font-bold py-2 px-4", {
        "bg-neutral-900 text-white": color === Color.Black,
        "bg-white text-black border-neutral-300": color === Color.White,
      })}
    >
      {formatTimeRemaining(timeRemaining)}
    </div>
  );
}
```

Die Komponente zeigt zunächst nur die verbleibende Zeit vom Server formatiert an. Wenn der Spieler am Zug ist, soll die Uhr starten und die verbleibende Zeit herunterzählen. Hierfür bietet es sich an *setInterval* oder *setTimeout* zu nutzen, um jede Sekunde den Wert um eine Sekunde zu reduzieren.

```
setInterval(() => {
  // Zeit aktualisieren
}, 1000);
```

Dieser Ansatz führt aber sehr schnell dazu, dass die Uhr nicht mehr synchron mit der Server Uhr ist. Der Grund dafür ist, dass diese Funktionen im Browser äusserst ungenau sind. Die Genauigkeit hängt davon ab, was der Hauptthread vom Browser sonst noch zu tun hat. Wenn der Thread blockiert ist, wird die Ausführung der Callback-Funktion verzögert. [57] Anhand von Debugging des Autors ist die Uhr mit diesem Ansatz nach nur schon 5 Sekunden stark desynchronisiert von der Server Uhr.

Als Abhilfe können Webworkers genutzt werden. [15] Webworkers erlauben es Funktionalität in Hintergrundthreads, anstatt des Hauptthreads des Browsers laufen zu lassen. Zur Folge sind Funktionen wir *setTimeout* und *setInterval* um einiges präziser. Der Hintergrundthread kann mit dem Hauptthread via Nachrichten kommunizieren. Folgend die Implementation des Webworkers:

```
let count = 0;
let interval: ReturnType<typeof setInterval>;

self.onmessage = function (event: any) {
  switch (event.data.command) {
    case "start":
      clearInterval(interval);
      count = event.data.timeRemaining;
      tick();
      interval = setInterval(tick, 500);
      break;
    case "stop":
      clearInterval(interval);
      break;
  }
};

function tick() {
  if (count <= 0) {
    clearInterval(interval);
    return;
  }
  count -= 500;
  if (count < 0) count = 0;
  postMessage(count);
}
```

Wenn der Worker die Nachricht *start* mit der verbleibenden Zeit erhält, wird eine Schlaufe gestartet, die alle 500 Millisekunden die verbleibende Zeit aktualisiert und eine Nachricht an den Hauptthread mit der aktualisierten Zeit schickt. Auf die Nachricht *stop* reagiert der Worker, indem er die Schlaufe beendet.

Der Webworker kann nun in der *Clock* Komponenten genutzt werden.

```
const [localTimeRemaining, setLocalTimeRemaining] = useState<number | null>(
  timeRemaining
);

useEffect(() => {
  setLocalTimeRemaining(timeRemaining);
  if (isTicking && timeRemaining !== null) {
    workerRef.current?.postMessage({ command: "start", timeRemaining });
  } else {
    workerRef.current?.postMessage({ command: "stop" });
  }
}, [isTicking, timeRemaining]);
```

Auf Nachrichten vom Webworker wird wie folgt reagiert:

```
workerRef.current.onmessage = (event) => setLocalTimeRemaining(event.data);
```

Diese Lösung funktioniert zuverlässig und bleibt über längere Zeit mit dem Server synchron. Es wird immer Abweichungen geben, aber sie sind minimal. Zudem wird die Zeit bei jedem Zug mit dem Server abgeglichen und die Abweichungen sind somit für Menschen nicht bemerkbar.

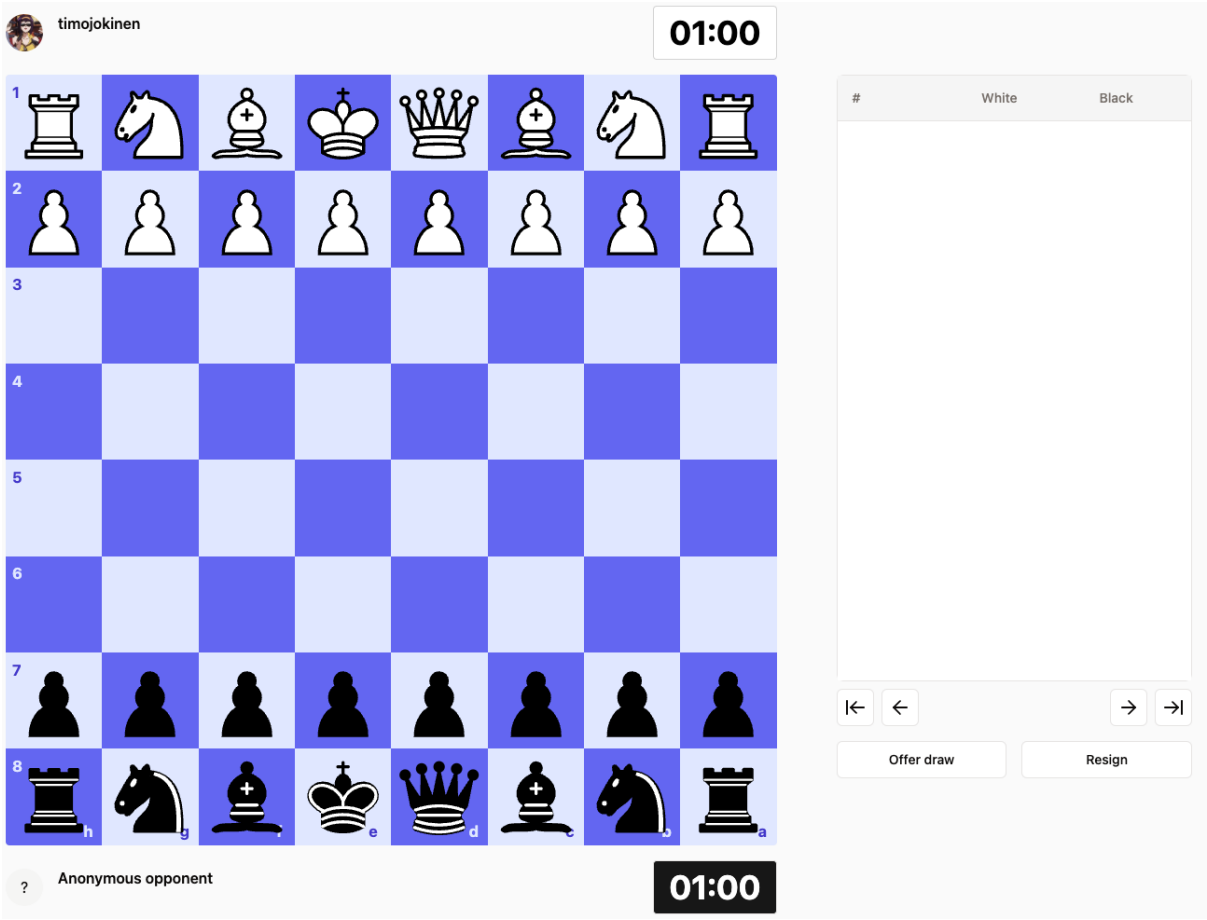

Abbildung 29 Online-Spiel mit Schachuhr

7.3.10 Stockfish

Als nächstes wird Stockfish integriert. Hier wurde bereits entschieden, dass die WebAssembly Version von Stockfish 15 eingesetzt wird. Diese steht öffentlich zum Download zur Verfügung.

Um Stockfish zu integrieren, wird ein neuer Kontext *StockfishContext* erstellt.

```
export const StockfishContext = React.createContext<StockfishContextState>({});
```

Und der dazugehörige Provider:

```
export default function StockfishProvider({ children }: Props) {
  return (
    <>
      <StockfishContext.Provider
        value={
          {
            // TODO
          }
        }
      >
        {children}
      </StockfishContext.Provider>
      <Script src="/stockfish.js" onReady={() => {}}></Script>
```

```
    </>
  );
}
```

Der Provider rendert zudem ein *Script*, welches Stockfish in den Browser lädt. Die *onReady* Funktion wird ausgeführt, sobald das Script geladen wurde. Um Stockfish zu initialisieren, wird via *onReady* eine Funktion ausgeführt:

```
const initStockfish = async () => {
  if (!Stockfish) throw new Error("Stockfish.js was not loaded yet.");
  const sf = await Stockfish();
  await sf.ready;
};
```

Ab diesem Zeitpunkt ist Stockfish global im Browserkontext verfügbar.

7.3.10.1 UCI

Das UCI-Protokoll (Universal Chess Interface) ermöglicht die Kommunikation zwischen Schachsoftware und Schachcomputer wie Stockfish. Dabei basiert UCI auf einem Nachrichtensystem, bei dem Befehle und Daten zwischen dem Benutzerinterface und der Schach-Engine ausgetauscht werden. Durch dieses Protokoll können Anwender Befehle an die Schach-Engine senden und Ergebnisse erhalten. Einige gängige Befehle im UCI-Protokoll sind zum Beispiel *position*, um eine Schachstellung zu setzen, und *go*, um die Engine zum Berechnen des besten Zuges anzuregen. Die Kommunikation erfolgt in der Regel in einem textbasierten Format, was die Implementierung und Nutzung des Protokolls relativ einfach und effizient macht. Mit dem UCI-Protokoll können Entwickler auf eine standardisierte Weise mit verschiedenen Schach-Engines interagieren, was die Interoperabilität und die Flexibilität des Systems erhöht. [65]

Der Ablauf ist folgendermassen:

1. Das Benutzerinterface sendet die Nachricht *uci*, um die Kommunikation zu starten.
2. Die Engine antwortet mit *uciok*.
3. Anschliessend kann das Benutzerinterface die Engine mit Optionen konfigurieren, indem eine oder mehrere Nachrichten im Format *setoption <name> value <value>* gesendet werden. Die Optionen geben vor wie die Engine sich verhält (beispielsweise wie viele Threads verwendet werden dürfen, wie stark die Engine ist oder bis zu welcher Tiefe sie Positionen evaluieren soll).
4. Wenn alle Optionen gesetzt sind, sendet das Benutzerinterface die Nachricht *isready*.
5. Die Engine antwortet mit der Nachricht *readyok*.
6. Das Benutzerinterface sendet die Nachricht *ucinewgame*, um der Schach-Engine zu sagen, dass sie ein neues Spiel vorbereiten soll.
7. Das Benutzerinterface sendet die Nachricht *isready*, um zu signalisieren, dass nun Positionen evaluiert werden sollen.
8. Die Engine antwortet erneut mit der Nachricht *readyok*.
9. Das Benutzerinterface kann nun eine Position definieren, welche als Ausgangslage dient. Dafür wird die Nachricht *position fen <FEN>* gesendet.
10. Anschliessend kann die Position evaluiert werden, indem die Nachricht *go depth <depth>* gesendet wird. Die Tiefe bestimmt, wie viele Züge vorwärts die Engine berechnet.
11. Ab diesem Punkt beginnt die Engine zu rechnen und sendet fortlaufend Nachrichten mit Informationen, bis die gewählte Tiefe erreicht ist.
12. Wenn die gewählte Tiefe erreicht ist, hört die Engine auf zu Rechnen und sendet die Nachricht *bestmove <Zugnotation>*, welche den besten Zug in der jeweiligen Position definiert.

Es muss dazu gesagt werden, dass das UCI-Protokoll noch einige weitere Funktionen zur Verfügung stellt. Hier wurde nur dokumentiert, was für die Umsetzung im Projekt relevant ist.

Die Nachrichte, die die Engine sendet, während sie rechnet, enthält wichtige Informationen zur Stellung, die verwendet werden können, um die Bewertungsleiste anzuzeigen. Beispiel:

info depth 17 seldepth 27 multipv 1 score cp 101 nodes 4079925 nps 9783992 time 417 pv e2e4 e7e5 g1f3 b8c6 f1b5 g8f6 e1g1 f6e4 d2d4 f8e7 d1e2 e4d6 b5c6 b7c6 d4e5 d6b7 c2c4 e8g8 f1d1 b7c5 c1e3 c8a6 e3c5 e7c5 b1c3

Die wichtigste Information für dieses Projekt ist der *cp* Wert (in diesem Fall 101). CP steht für Centipawn und ist eine Masseinheit, die Engines verwenden, um den Vorteil der Farbe, die am Zug ist, darzustellen. Ein Centipawn ist ein Hundertstel eines Bauern. Somit sind 100 Centipawns = 1 Bauer. Wenn der Wert beispielsweise 400 ist, bedeutet das, dass der Vorteil der Seite, die am Zug ist, als hätte diese Seite vier Bauern mehr als die gegnerische. Dieser Wert bezieht sich nicht nur auf die Figuren auf dem Brett, sondern rechnet auch strategische Vorteile mit ein. Diese Information kann verwendet werden, um die Bewertungsleiste anzuzeigen. [66]

7.3.10.2 Implementation

Der Ablauf kann nun in der Applikation implementiert werden. Da es sich dabei um eine grosse Menge von Programmcode handelt, werden hier nur die wichtigsten Punkte aufgeführt.

Es wird eine Klasse *EngineWrapper* erstellt, die eine Stockfish Instanz entgegennimmt. Die Klasse initialisiert ein Spiel gemäss dem beschriebenen Ablauf und veröffentlicht eine Methode, welche eine *FEN* entgegennimmt und die Engine rechnen lässt. Damit die Nachrichten der Engine in der korrekten Reihenfolge abgearbeitet werden, wird ein *Queue* Datentyp implementiert:

```
class Queue {
  private getter: null | ((value: string | PromiseLike<string>) => void);
  private list: string[];

  constructor() {
    this.getter = null;
    this.list = [];
  }

  async get() {
    if (this.list.length > 0) {
      return this.list.shift()!;
    }

    return new Promise<string>((resolve) => (this.getter = resolve));
  }

  put(x: string) {
    if (this.getter) {
      this.getter(x);
      this.getter = null;
    } else {
      this.list.push(x);
    }
  }
}
```

Es handelt sich hierbei um eine FIFO-Queue, also first-in first-out. Die erste Nachricht wird auch als erstes abgearbeitet.

Der Konstruktor der *EngineWrapper* Klasse fügt alle Nachrichten der Engine in die Queue hinzu.

```
constructor(stockfishInstance: StockfishInstance) {
  this.stockfishInstance = stockfishInstance;
  this.queue = new Queue();
  this.stockfishInstance.addMessageListener((line) => this.queue.put(line));
}
```

Die Methode in der *EngineWrapper* Klasse, um Positionen zu analysieren sieht wie folgt aus:

```
public async evaluatePosition(fen: string, depth: number) {
  const { activeColor } = parseFEN(fen);
  this.activeColor = activeColor;

  this.send("stop");
  this.send(`position fen ${fen}`);
  this.send(`go depth ${depth}`);

  await this.receiveUntil((line) => line.startsWith("bestmove"));
}
```

Zuerst werden bisherige Rechnungen gestoppt mit der *stop* Nachricht. Dann wird die Position gesetzt und anhand der gegebenen Tiefe berechnet, bis die Nachricht *bestmove* von der Engine kommt.

Weiterhin implementiert die Klasse zwei Methoden, damit Konsumenten über Informationen benachrichtigt werden können:

```
public subscribe(subscriber: Subscriber): () => void {
  this.subscribers.push(subscriber);
  return () => this.unsubscribe(subscriber);
}

public unsubscribe(subscriber: Subscriber) {
  const index = this.subscribers.indexOf(subscriber);
  if (index !== -1) {
    this.subscribers.splice(index, 1);
  }
}
```

Jede Zeile, die die Engine schickt, wird dann abgearbeitet. Für jede *info* Nachricht wird der Centipawn Wert in einen Wert von 10 (Weiss hat eine gewinnende Position) bis -10 (Schwarz hat

eine gewinnende Position) übersetzt und an alle Subscriber geschickt (zusammen mit anderen Werten wie dem besten Zug in der Position). Im Stockfish Kontext kann nun eine *EngineWrapper* Klasse instanziiert werden:

```
const engineWrapper = new EngineWrapper(sf);
await engineWrapper.initialize({
  Threads: navigator?.hardwareConcurrency ?? 1, // Ein Thread pro CPU Kern
});
await engineWrapper.initializeGame();
```

Anschliessend kann eine Subscription erstellt werden:

```
useEffect(() => {
  const unsubscribe = engine?.subscribe((data) => {
    if (data.type === "evaluation") {
      setEvaluation(data.value);
      setMateIn(data.mateIn ?? null);
      setBestMove(null);
    } else {
      setBestMove(data.value);
    }
  });

  return () => {
    unsubscribe?.();
  };
}, [engine]);
```

Nun wird jedes Mal, wenn die Position ändert, die *evaluatePosition* Methode der *EngineWrapper* Klasse aufgerufen. Als Resultat erhält der Stockfish Kontext die nötigen Informationen.

7.3.10.3 Bewertungsleiste

Mit den Informationen zur Stellung, die nun vom Stockfish Kontext zur Verfügung gestellt werden, kann eine Komponente für die Bewertungsleiste erstellt werden. Die Bewertungsleiste nutzt die Bibliothek *motion*, um den Wert zu animieren.

```
export default function EvalBar({ flipped }: Props) {
  const { evaluation, mateIn } = useContext(StockfishContext);
  const percent = 100 - ((evaluation + 10) / 20) * 100;
  return (
    <div>
      <motion.div
        initial={{ height: "50%" }}
        animate={{ height: percent + "%" }}
        className="bg-black w-full absolute bottom-0 flex justify-center"
      ></motion.div>
    </div>
  );
}
```

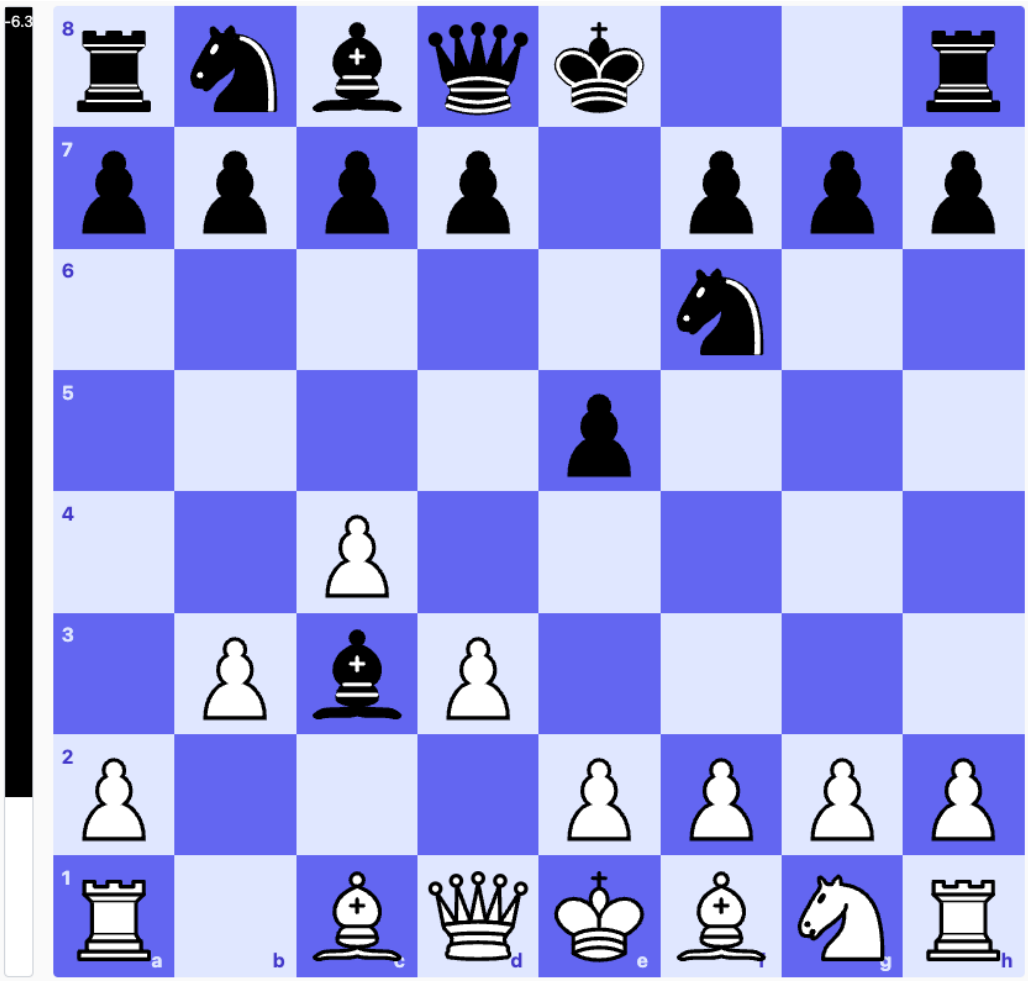

Abbildung 30 Brett mit Bewertungsleiste

7.3.10.4 Engine-Spiel

Mit den Informationen zur Stellung und dem besten Zug kann schlussendlich auch noch das Spiel gegen Stockfish implementiert werden. Dazu wird eine neue Seite erstellt und es werden einfach alle Komponenten, die bisher erarbeitet wurden, zusammengefügt. Der Spieler kann Züge machen, und wenn Stockfish am Zug ist, wird einfach der beste Zug aus dem Stockfish Kontext angewendet. Folgender Programmcodeabschnitt ist aus der neu erstellten Seite *game/engine* und macht genau das:

```
useEffect(() => {
  if (activeColor === playerColor || !bestMove) {
    return;
  }

  const origin = bestMove.slice(0, 2);
  const target = bestMove.slice(2, 4);
  const promotion = bestMove.slice(4, 5);

  const move = legalMoves.find(
    (move) =>
      move.sourceSquare === squareToIndex(origin) &&
      move.targetSquare === squareToIndex(target) &&
      (move.promotionPiece?.toLowerCase() ?? "" === promotion.toLowerCase())
  );

  if (move) {
    makeMove(move);
  }
}, [activeColor, playerColor, bestMove, legalMoves, makeMove]);
```

7.3.11 Profil

Als letztes muss das Benutzerprofil, mit der Historie aller Spiele des Benutzers erstellt werden. Das ist glücklicherweise sehr simpel. Als Seite wird ein Servercomponent verwendet, der eine Abfrage an die Datenbank macht und alle Spiele des jeweiligen Benutzers abruft.

```
const session = await getServerSession(authOptions);
if (!session || !session.user) {
  redirect("/api/auth/signin");
}

const games = await prisma.game.findMany({
  where: {
    AND: {
      NOT: {
        result: null,
      },
      OR: [
        { blackPlayerId: session?.user.id },
        { whitePlayerId: session?.user.id },
      ],
    },
```

```
  },
  orderBy: {
    createdAt: "desc",
  },
  include: {
    blackPlayer: true,
    whitePlayer: true,
  },
});
```

Die Spiele werden dann in einer Liste angezeigt, sortiert nach Datum. Per Klick auf einen Eintrag wird der Benutzer auf eine weitere Seite weitergeleitet, auf welcher er das Spiel über die Zugnavigation nochmal von vorne bis zum Schluss durchgehen kann. Dazu wird die Bewertungsleiste angezeigt, damit der Benutzer seine Fehler erkennen kann. Der Autor hält es nicht für nötig, in die Implementation einzugehen.

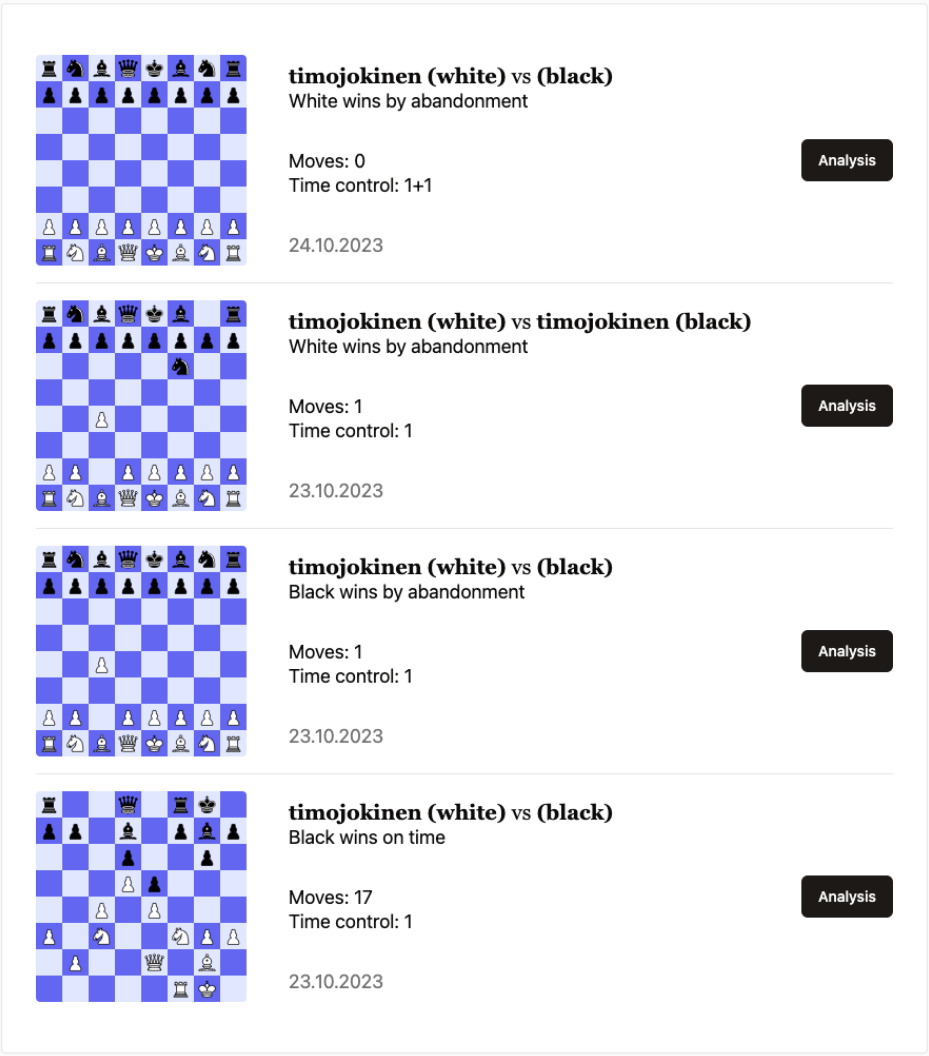

Abbildung 31 Spielehistorie im Profil

7.3.12 Abschluss und Endprodukt

Mit den dokumentierten Komponenten und dank der modularen Architektur kann jede Variation der Anforderungen (Startseite, Online-Spiel, Engine-Spiel, Analyse) abgedeckt werden, indem die

Komponenten zusammengefügt werden. Ein Grossteil davon wurde abgedeckt, aber nicht alles, da sich der Inhalt nur wiederholen würde. Mit dem Benutzerinterface und Stockfish kommen nun alle bisher erarbeiteten Komponenten zusammen, was den Abschluss der Implementation im Umfang der Arbeit bedeutet.

8 Endresultat

Für die Vollständigkeit werden folgend Screenshots von allen erstellten Seiten noch einmal aufgeführt.

Startseite

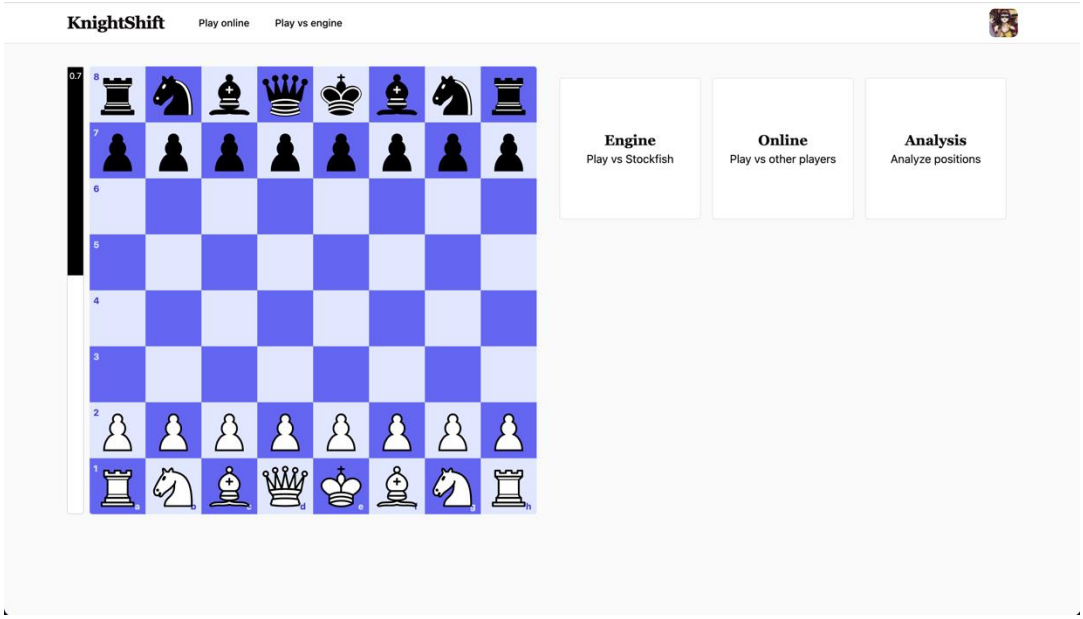

Abbildung 32 Screenshot Startseite

Herausforderungen

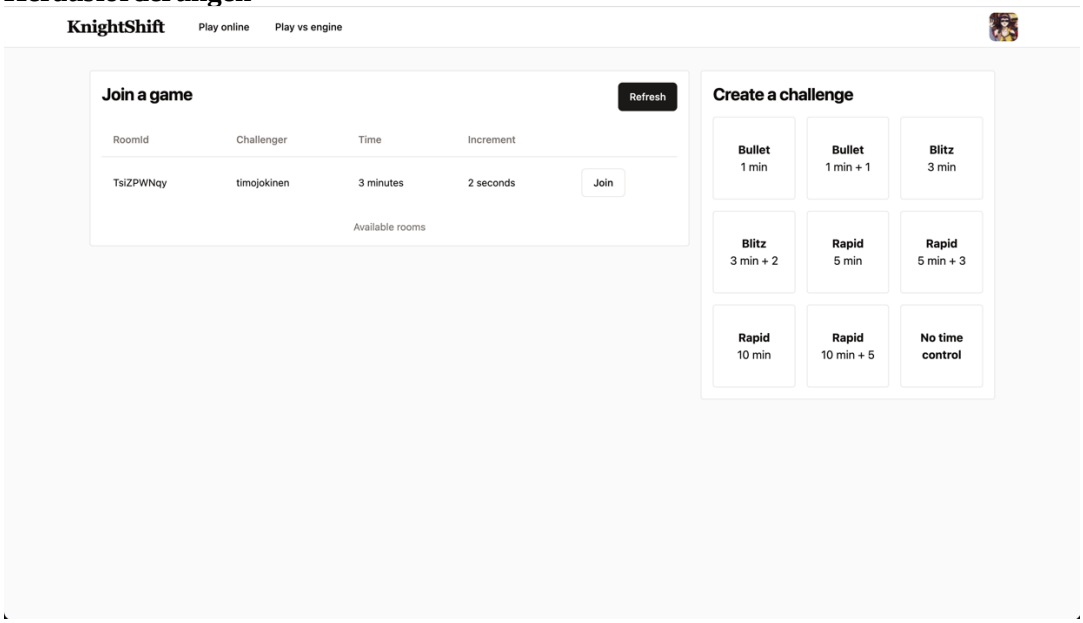

Abbildung 33 Screenshot Herausforderungen Seite

Spiel gegen Stockfish

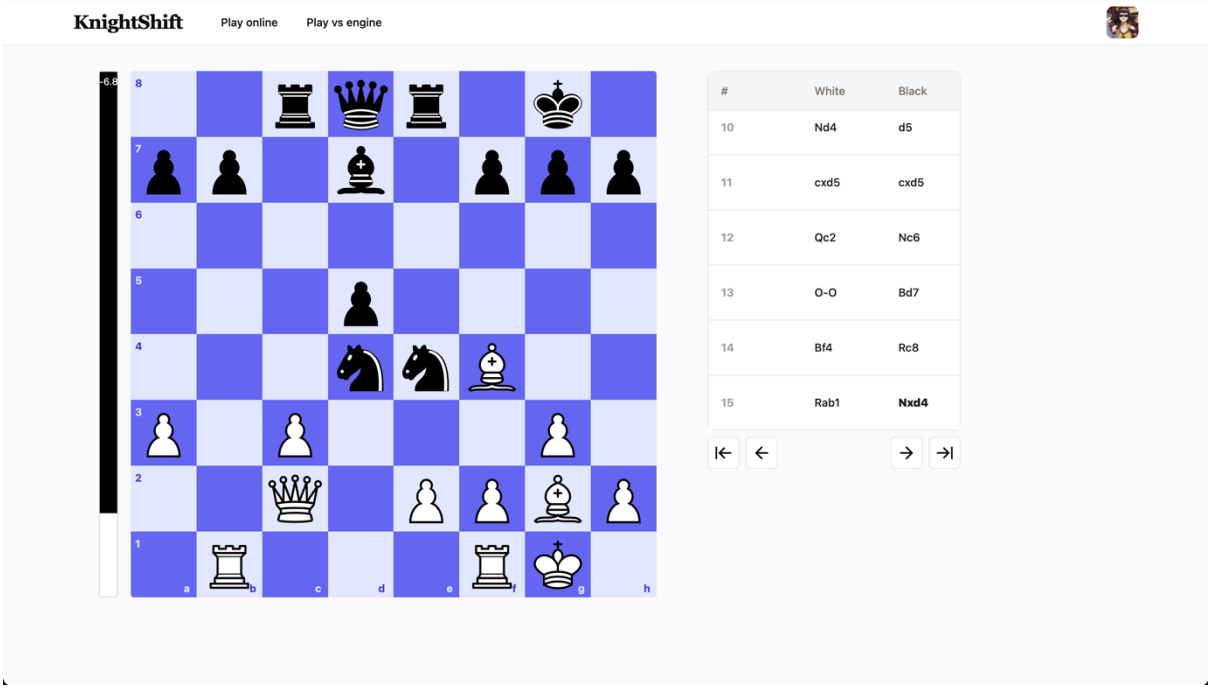

Abbildung 34 Screenshot Spiel gegen Stockfish

Online-Spiel

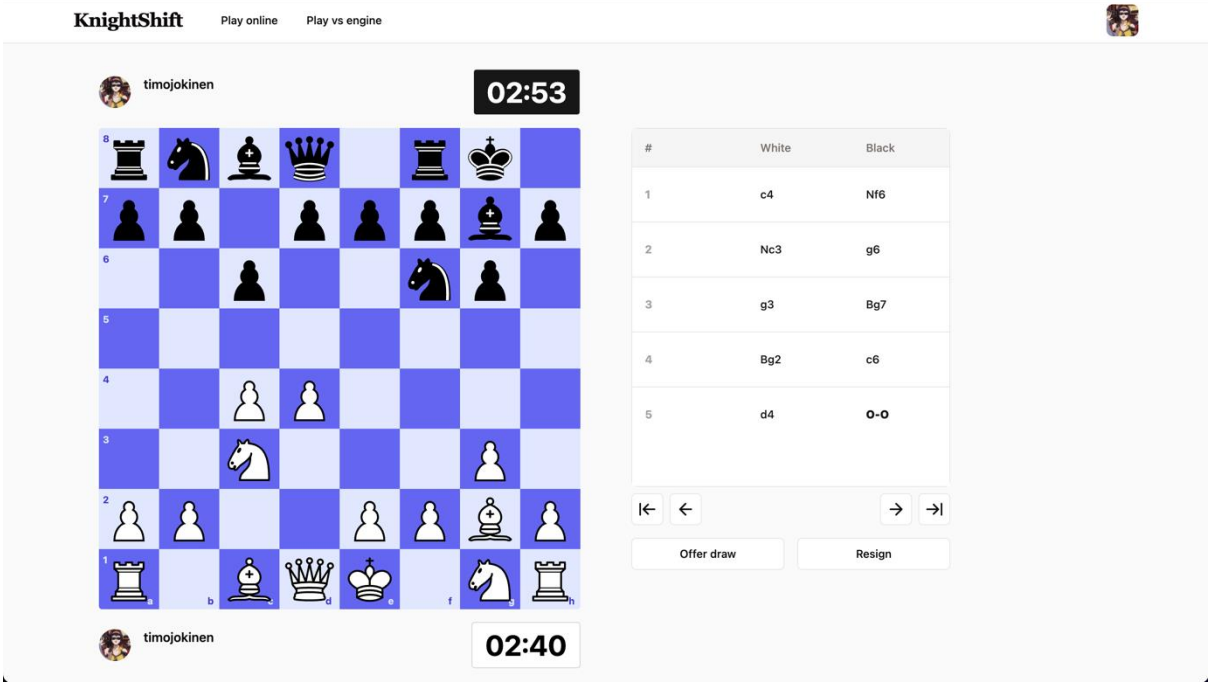

Abbildung 35 Screenshot Online-Spiel

Profil

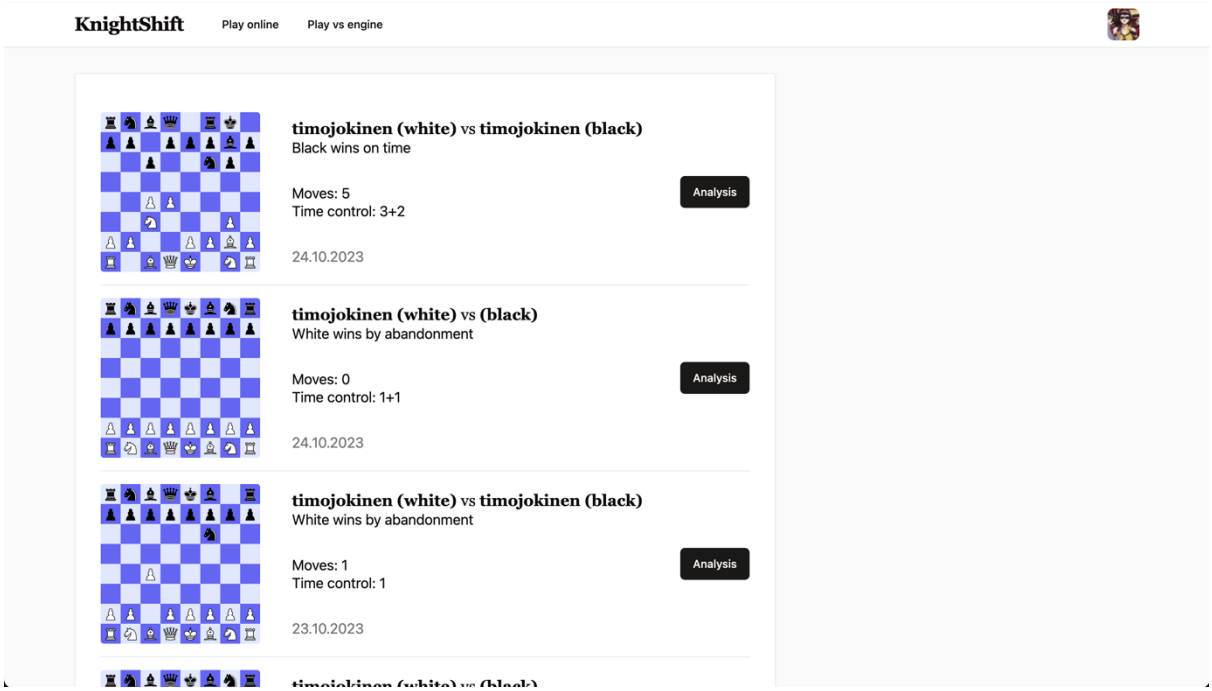

Abbildung 36 Screenshot Profil

Analyse

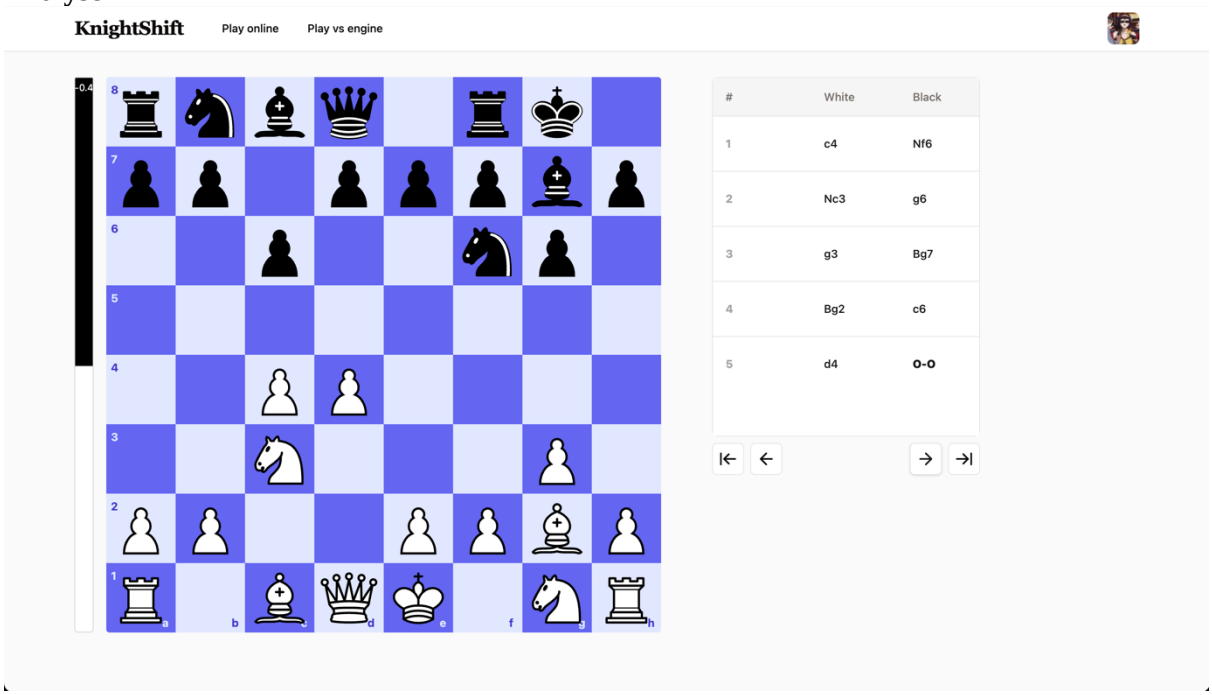

Abbildung 37 Screenshot Analyse

9 Evaluation

Mit dem Benutzerinterface kommen nun alle Komponenten des Projekts zusammen und die Implementation für den MVP ist vollendet. Um das Projekt zu evaluieren, werden in diesem Kapitel zuerst die User Storys durchgegangen. Dabei wird überprüft, ob die definierten Akzeptanzkriterien erreicht sind und in einem zweiten Schritt werden auch noch die Erfolgskriterien analysiert und ausgewertet.

9.1 User Storys

Name	Konto erstellen	
Akzeptanzkriterien	Möglichkeit ein Konto zu erstellen	Erfüllt
	Möglichkeit sich mit dem erstellten Konto anzumelden	Erfüllt
Relevanz	Erforderlich	
Status	Abgeschlossen	

Name	Profil	
Akzeptanzkriterien	Möglichkeit eigene Spiele auf einer Seite anzusehen	Erfüllt
Relevanz	Erforderlich	
Status	Abgeschlossen	

Name	Spiel erstellen	
Akzeptanzkriterien	Möglichkeit Spiele zu erstellen	Erfüllt
	Möglichkeit Spiele mit Zeitformat zu konfigurieren (Bedenkzeit und Inkrement)	Erfüllt
	Funktioniert auch für Benutzer ohne Benutzerkonto	Erfüllt
Relevanz	Erforderlich	
Status	Abgeschlossen	

Name	Herausforderungen sehen	
Akzeptanzkriterien	Möglichkeit eine Liste der Herausforderungen zu sehen. In der Liste wird aufgeführt, was für ein Zeitformat das Spiel hat und wer die Herausforderung erstellt hat.	Erfüllt
	Funktioniert auch für Benutzer ohne Benutzerkonto	Erfüllt
Relevanz	Erforderlich	
Status	Abgeschlossen	

Name	Herausforderung annehmen	
Akzeptanzkriterien	Möglichkeit Herausforderungen anzunehmen	Erfüllt
	Funktioniert auch für Benutzer ohne Benutzerkonto	Erfüllt
Relevanz	Erforderlich	
Status	Abgeschlossen	

Name	Elo Bewertung	
Akzeptanzkriterien	Benutzer erhalten nach einer bestimmten Anzahl von Spielen eine Bewertung, die ihren Fähigkeiten entspricht.	Nicht erfüllt
Relevanz	Erwünschenswert	
Status	Offen – zu hohe Komplexität	

Name	Paarung	
Akzeptanzkriterien	Möglichkeit in eine Paarungs-Warteschlaufe einzutreten	Nicht erfüllt
	Sobald ein Partner gefunden wurde, beginnt das Spiel	Nicht erfüllt
Relevanz	Erwünschenswert	
Status	Offen – zu hohe Komplexität	

Name	Schachcomputer herausfordern	
Akzeptanzkriterien	Möglichkeit gegen einen Schachcomputer spielen zu können	Erfüllt
	Es wird eine Bewertung der Position mittels einer Bewertungsleiste angezeigt.	Erfüllt
Relevanz	Erforderlich	
Status	Abgeschlossen	

Name	Analyse eigener Spiele	
Akzeptanzkriterien	Möglichkeit eigene vergangene Spiele Zug für Zug durchzuspielen	Erfüllt
	Es wird eine Bewertung der Position mittels einer Bewertungsleiste angezeigt.	Erfüllt
Relevanz	Erforderlich	
Status	Abgeschlossen	

Name	Schachregeln	
Akzeptanzkriterien	Möglichkeit Züge auszuführen, die in der Position erlaubt sind.	Erfüllt
Relevanz	Erforderlich	
Status	Abgeschlossen	

Name	Spielende	
Akzeptanzkriterien	Schachmatt wird erkannt	Erfüllt
	Remis wird erkannt	Erfüllt
Relevanz	Erforderlich	
Status	Abgeschlossen	

Name	Schachuhr	
Akzeptanzkriterien	Bedenkzeit wird pro Spieler angezeigt	Erfüllt
	Wenn die Bedenkzeit abläuft, wird das Spiel beendet.	Erfüllt
Relevanz	Erforderlich	
Status	Abgeschlossen	

Name	Resignieren	
Akzeptanzkriterien	Möglichkeit ein Spiel aufzugeben	Erfüllt
Relevanz	Erwünschenswert	
Status	Abgeschlossen	

Name	Remis anbieten	
Akzeptanzkriterien	Möglichkeit Remis anzubieten	Erfüllt
	Möglichkeit Remis anzunehmen	Erfüllt
	Möglichkeit Remis abzulehnen	Erfüllt
Relevanz	Erwünschenswert	
Status	Abgeschlossen	

Name	Spielübersicht	
Akzeptanzkriterien	Möglichkeit eine Historie aller gemachten Züge anzusehen	Erfüllt
	Möglichkeit über Züge zu navigieren, um eine vergangene Position anzusehen	Erfüllt
Relevanz	Erforderlich	
Status	Abgeschlossen	

Name	Zug zurücknehmen	
Akzeptanzkriterien	Möglichkeit Züge zurücknehmen zu können.	Nicht erfüllt
	In Online-Spielen muss der Gegner den Vorschlag zur Rücknahme akzeptieren oder ablehnen können.	Nicht erfüllt
Relevanz	Erwünschenswert	
Status	Offen - Zeitmangel	

Name	Notizen	
Akzeptanzkriterien	Möglichkeit Notizen machen zu können.	Nicht erfüllt
	Notizen können zu einem späteren Zeitpunkt wieder angesehen werden.	Nicht erfüllt
Relevanz	Erwünschenswert	
Status	Offen - Zeitmangel	

Name	Chatfunktion	
Akzeptanzkriterien	Möglichkeit Nachrichten im Chat zu schreiben.	Nicht erfüllt
Relevanz	Erwünschenswert	
Status	Offen - Zeitmangel	

9.1.1 Evaluation

Erfreulicherweise sind alle erforderlichen User Storys implementiert. Einige Funktionen, die als erwünschenswert aber optional markiert sind, konnten nicht berücksichtigt werden. Es ist normal, dass bei einem grossen Projekt nicht alle Anforderungen in den Umfang passen. Die implementierten User Storys bilden eine gute Grundlage für weitere Entwicklung ausserhalb des Umfangs des MVPs.

9.2 Erfolgskriterien

Folgend werden die Erfolgskriterien evaluiert. Bei einigen Zielen wurde ein Kommentar hinzugefügt.

9.2.1 Funktionale Erfolgskriterien

9.2.1.1 Registrierung

Kriterium	Benutzer können ein Benutzerkonto erstellen.
Messung	Funktionalität ist vorhanden
Relevanz	Erforderlich
Status	Ziel erreicht

9.2.1.2 Login

Kriterium	Benutzer können sich mit ihrem Konto bei der Webseite anmelden.
Messung	Funktionalität ist vorhanden
Relevanz	Erforderlich
Status	Ziel erreicht

9.2.1.3 Online-Spiel

Kriterium	Benutzer können mit oder ohne Anmeldung Schach gegen andere Benutzer spielen.
Messung	Funktionalität ist vorhanden
Relevanz	Erforderlich
Status	Ziel erreicht

9.2.1.4 Herausforderung

Kriterium	Benutzer können eine Herausforderung erstellen und andere Benutzer können diese annehmen.
Messung	Funktionalität ist vorhanden
Relevanz	Erforderlich
Status	Ziel erreicht

9.2.1.5 Matchmaking

Kriterium	Benutzer können nach sich in die Matchmaking-Warteschlaufe begeben und werden automatisch mit einem passenden Kontrahenten gepaart.
Messung	Funktionalität ist vorhanden
Relevanz	Optional
Status	Ziel nicht erreicht
Kommentar	Matchmaking hat sich als ein äusserst kompliziertes Thema herausgestellt, weswegen der Autor erforderliche Ziele priorisiert hat, mit der Intention das Thema am Ende der Arbeit noch einmal anzugehen. Dafür wurde die Zeit zu knapp und die Funktion kann für die Zukunft, ausserhalb des Projekts angegangen werden.

9.2.1.6 Schachcomputer-Spiel

Kriterium	Benutzer können gegen einen starken Schachcomputer spielen. (Angedacht ist Stockfish 15)
Messung	Funktionalität ist vorhanden
Relevanz	Erforderlich
Status	Ziel erreicht

9.2.1.7 Spieloptionen

Kriterium	Es stehen verschiedene Zeitformate zur Verfügung in Form von Bedenkzeit pro Spieler und ein optionales Inkrement (Zeit, die nach jedem Zug zur Bedenkzeit dazugerechnet wird).
Messung	Mindestens folgende Zeitformate stehen zur Verfügung: - 3 Minuten - 3 Minuten + 2 Sekunden Inkrement - 5 Minuten - 5 Minuten + 3 Sekunden Inkrement - 10 Minuten
Relevanz	Erforderlich
Status	Ziel erreicht
Kommentar	Die Applikation wurde so aufgebaut, dass Zeitformate flexibel sind und somit jede mögliche Kombination von Bedenkzeit und Inkrement möglich ist.

9.2.1.8 Benutzerprofil

Kriterium	Benutzer sehen eine Liste der vergangenen Spiele in ihrem Profil und können diese Zug für Zug noch einmal durchspielen.
Messung	Funktionalität ist vorhanden
Relevanz	Erforderlich
Status	Ziel erreicht

9.2.1.9 Analyse

Kriterium	Für vergangene Spiele wird Benutzern eine Bewertungsleiste angezeigt, die von einem Schachcomputer angetrieben wird, um die Positionsstärke für Weiss oder Schwarz zu veranschaulichen.
Messung	Funktionalität ist vorhanden
Relevanz	Erforderlich
Status	Ziel erreicht

9.2.2 Technische Erfolgskriterien

9.2.2.1 Leistungsfähigkeit

Kriterium	Die Plattform kann mehrere Spiele gleichzeitig abwickeln.
Messung	Es wird ein Last-Test durchgeführt, wobei die Plattform 500 Spiele gleichzeitig stemmen soll.
Relevanz	Erforderlich
Status	Ziel erreicht
Kommentar	Die Plattform hat problemlos 500 Spiele gestemmt.

9.2.2.2 Skalierbarkeit

Kriterium	Die Plattform soll horizontal skalierbar sein (bedeutet mehrere Instanzen der Applikation können nebeneinander laufen).
Messung	Plattform ist so aufgebaut, dass weitere Instanzen dazugeschaltet werden könnten.
Relevanz	Erforderlich
Status	Ziel erreicht
Kommentar	Die Skalierung wurde zwar nicht implementiert, aber es wurde sichergestellt, dass die Möglichkeit besteht zu skalieren. Der Autor befindet dieses Ziel somit als erreicht.

9.2.2.3 Echtzeit

Kriterium	Spiele auf der Plattform zwischen zwei menschlichen Benutzern laufen in Echtzeit.
Messung	Züge von Kontrahenten werden sofort angezeigt, ohne dass die Seite aktualisiert werden muss.
Relevanz	Erforderlich
Status	Ziel erreicht

9.2.2.4 Legale Züge errechnen

Kriterium	Die Ausrechnung der legalen Züge wird programmiert, anstatt eine Bibliothek zu verwenden und integriert fortgeschrittene Techniken aus der Schachprogrammierung.
Messung	- Funktionalität ist vorhanden. - Code ist selbst geschrieben und verwendet Schachprogrammierung Theorie. - Performance wird evaluiert und die legalen Züge werden für eine Position in unter 10 Millisekunden generiert.
Relevanz	Erforderlich
Status	Ziel erreicht
Kommentar	Das Ziel wurde übertroffen, indem die Zuggeneration im Durchschnitt weniger als eine halbe Millisekunde dauerte.

9.2.2.5 Stockfish

Kriterium	Als Schachcomputer und für die Evaluation von Positionen wird Stockfish eingesetzt.
Messung	Stockfish wird integriert.
Relevanz	Erforderlich
Status	Ziel erreicht

9.2.2.6 Automatisierte Tests

Kriterium	Kritische Stellen der Applikation werden mittels automatisierten Tests getestet.
Messung	Automatisierte Tests sind vorhanden.
Relevanz	Erforderlich
Status	Ziel erreicht
Kommentar	Es wurde nur ein Testfall mit Unit-Tests abgedeckt, nämlich dass der Zuggenerator Spiele korrekt abwickeln kann. Da die Schachregeln der Kern der gesamten Plattform darstellen, ist das wohl die kritischste Stelle. Man kann die Lasttests für den Colyseus Server auch als automatisierten Test zählen, denn diese testen neben der Last auch, ob die Spiellogik korrekt abläuft. Während dem Durchlauf ist kein Spiel gescheitert - alle Spiele wurden mit einem Resultat (Weiss gewinnt, Schwarz gewinnt oder Remis) abgeschlossen, was ein Testament für die korrekte Implementation und Stabilität des Gameservers ist. Nichtsdestotrotz wäre es sinnvoll gewesen für das Benutzerinterface Unit-Tests zu schreiben. Das hat leider nicht in den Umfang der Arbeit gepasst.

9.2.2.7 Sicherheit

Kriterium	Die Methode der Authentifizierung entspricht heutigen Standards.
Messung	Die Authentifizierung folgt einer etablierten Methode.
Relevanz	Erforderlich
Status	Ziel erreicht
Kommentar	Ziel wurde erreicht, indem der OpenIDConnect Standard mit GitHub als Anbieter eingebaut wurde. Da dieser Standard weit etabliert ist und eine Bibliothek verwendet wurde, die aktiv gewartet wird und von einem Techgiganten (Vercel) unterstützt wird, geht der Autor davon aus, dass keine Sicherheitslücken bei der Authentifizierung bestehen. Da der Autor allerdings kein Penetrationstestexperte ist und keine jahrelange Ausbildung in IT Security hat, können Sicherheitslücken nicht komplett ausgeschlossen werden. Falls die Applikation zu einem bestimmten Zeitpunkt für die Öffentlichkeit freigegeben werden sollte, wäre es sinnvoll ein professionelles Security Testing zu veranstalten. Dies liegt allerdings weit ausserhalb des Umfangs des Projekts.

9.3 Bekannte Fehler und Unschönheiten

Im Verlaufe der Entwicklung wurden ein paar unkritische Fehler und Unschönheiten entdeckt, die nicht im Rahmen der Arbeit gelöst wurden.

- Spieler können aktuell ihren eigenen Spielen beitreten.
- Spieler können sich durch schnelles Klicken zu mehreren Räumen / Spielen verbinden, was zu unerwartetem Verhalten führt.
- Wenn ein Spieler ein Spiel erstellt, erhält er keine Rückmeldung / Ladeanzeige von der Applikation und wird plötzlich weitergeleitet, wenn ein anderer Spieler beitritt.
- Einem Spiel beitreten kann manchmal 2-3 Sekunden dauern. Währenddessen weiss der Benutzer nicht, was los ist, da keine Ladeanzeige vorhanden ist.
- In spezifischen Situationen schlägt die Bewertungsleiste fehl und zeigt offensichtlich falsche Werte an.
- Da die meisten Seiten sehr ähnlich aussehen, weiss man als Benutzer manchmal nicht, wo man sich befindet.
- Wenn man in einem Online-Spiel die Zugnavigation benützt, werden die geschlagenen Figuren pro Spieler nicht auf die angezeigte Position angepasst.

10 Reflexion

Mit der Implementation aller geplanten Komponenten kommt das Projekt im Rahmen dieser Arbeit nun zu Ende. Dieses umfassende Projekt hat mich, wie auch erwartet, stark herausgefordert. Allen Themen voran, haben die Recherche und die Implementation des Zuggenerators meine Fähigkeiten am meisten getestet. Schachprogrammierung ist ein äusserst tiefgründiges und komplexes Thema und da ich bis anhin keine Erfahrung damit hatte, investierte ich zahllose Stunden in das Erlernen der Theorie. Die Implementation hat somit auch länger gebraucht als erwartet, was den ganzen Zeitplan etwas nach rechts verschoben hat.

Auch das Benutzerinterface hat mehr Aufwand generiert, als initial erwartet. Um alle Funktionen fehlerfrei abzudecken, musste einige Male der Programmcode des Zuggenerators und des Gameservers aktualisiert werden. Glücklicherweise war ich mir von Anfang an bewusst, dass das geschehen kann und habe genug früh mit dem Projekt begonnen. Schlussendlich konnten alle Themen unter einen Hut gebracht werden. Wenn ich allerdings jetzt von vorne beginnen müsste, würde ich mir für die Diplomarbeit ein Thema auswählen, das etwas mehr spezialisiert in eine bestimmte Richtung ist, anstatt ein so umfangreiches Projekt mit vier Teilprojekten (Zuggenerator, Gameserver, Benutzerinterface, Stockfish), um mehr Zeit und Aufwand in Details investieren zu können. Einige Stellen im jetzigen Programmcode, vor allem im Benutzerinterface, sind etwas unpoliert oder nicht besonders schlau durchdacht. Normalerweise würde ich über den Code iterieren, bis ich damit zufrieden bin aber da der Umfang des Projekts gross war, hatte ich keine Zeit dafür.

Ein Thema, bei welchem ich mich schwergetan habe, war das Dokumentieren von verwendeten Technologien, speziell beim Benutzerinterface. Abzuschätzen, welche Punkte erklärt werden müssen und welche Punkte überflüssig sind, war nicht einfach und ich habe viele Stunden investiert, darüber zu grübeln, wie ich das Dokument so verständlich wie möglich gestalten kann. Auch die Dokumentation des Zuggenerators und von den Konzepten der Schachprogrammierung war nicht einfach. Um ein kompliziertes Konzept verständlich zu erklären, musste man es zuerst selbst bis ins letzte Detail verstehen. Da hatte ich allerdings auch den grössten Lerneffekt. Die Arbeit hat mir gezeigt, dass «technical writing» geübt sein muss und ich verstehe, warum Firmen Leute anstellen, nur für diesen Zweck.

Die Analysephase hätte noch umfassender stattfinden können, aber es ist alles so aufgegangen so wie geplant. Da ich als einzige Person am Projekt gearbeitet habe, habe ich die Techniken verwendet, die mir am meisten Klarheit verschaffen und mich zum Ziel bringen, anstatt Diagramme zu skizzieren, die dann doch nicht verwendet werden. Wenn ein ähnliches Projekt in einem Team umgesetzt würde, wäre das Vorgehen wohl etwas anders.

Abschliessend kann gesagt werden, dass ich seit der Themeneingabe nur mit wenigen Pausen fast täglich an der Arbeit gearbeitet habe und deswegen umso mehr stolz auf das Resultat bin. Glücklicherweise sind meine Pläne aufgegangen und auf dem Weg sind mir keine blockierenden Limitationen in den Weg gekommen. Das Projekt hat meine Fähigkeiten in verschiedensten Bereichen herausgefordert, mich dazu gezwungen mir neue Konzepte beizubringen und meine Programmierkenntnisse zu vertiefen. Ich fühle mich als Entwickler einen Schritt weiter als zum Zeitpunkt der Projekteingabe. Ich hoffe auch, dass mich die unzähligen Male, die ich im Verlaufe des Projekts gegen mich selbst Schach gespielt oder kläglich gegen Stockfish verloren habe, als Schachspieler etwas weitergebracht haben.

11 Anhang

11.1 Glossar

Minimum Viable Product / MVP	Ein Produkt, das die minimalen Anforderungen erfüllt, um funktionsfähig zu sein. Wird oft für das Austesten von Ideen verwendet.
UI	User Interface, Benutzerinterface
SCRUM	Eine Methodologie für agile Teams, basierend auf Iterationen
User Story	Eine kurz und prägnant formulierte Anforderung aus der Sicht eines Benutzers.
LLM	Large-Language-Model, ein Werkzeug das Text generieren kann, beispielsweise ChatGPT.
JIT-Compiler	Just-In-Time Kompilierung, Programmcode wird während der Laufzeit fort zu kompiliert.
Subscriber	Ein Subjekt, das in irgendeiner Art über etwas benachrichtigt wird.

11.2 Quellenverzeichnis

[1] «National Museums Liverpool,» [Online]. Available: https://www.liverpoolmuseums.org.uk/stories/which-greater-number-of-atoms-universe-or-number-of-chess-moves. [Zugriff am 8 August 2023].

[2] «Ziele formulieren: Mit der SMART-Formel klare Ziele formulieren,» [Online]. Available: https://www.weka.ch/themen/fuehrung-kompetenzen/mitarbeiterfuehrung/qualifikation-und-ziele/article/ziele-formulieren-mit-der-smart-formel-klare-ziele-formulieren/. [Zugriff am 4 September 2023].

[3] «Agile Manifesto,» [Online]. Available: https://agilemanifesto.org/. [Zugriff am 4 September 2023].

[4] «Atlassian: Microservices vs Monolith,» [Online]. Available: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith. [Zugriff am 8 August 2023].

[5] «Nakama Dokumentation,» [Online]. Available: https://heroiclabs.com/docs/nakama/getting-started/. [Zugriff am 10 September 2023].

[6] «Photon Dokumentation,» [Online]. Available: https://photonengine.com. [Zugriff am 10 September 2023].

[7] «Colyseus Dokumentation,» [Online]. Available: https://colyseus.io/. [Zugriff am 10 September 2023].

[8] «Next.js,» [Online]. Available: https://nextjs.org/docs. [Zugriff am 11 September 2023].

[9] «Next.js serverside Rendering,» [Online]. Available: https://nextjs.org/docs/app/building-your-application/rendering/client-components#full-page-load. [Zugriff am 11 September 2023].

[10] «Relational Databases Scaling,» [Online]. Available: https://www.marklogic.com/blog/relational-databases-scale/. [Zugriff am 21 September 2023].

[11] «SQL vs NoSQL,» [Online]. Available: https://www.coursera.org/articles/nosql-vs-sql. [Zugriff am 17 September 2023].

[12] «Neon Serverless Datenbank,» [Online]. Available: https://neon.tech/. [Zugriff am 21 September 2023].

[13] «PostgreSQL vs MySQL,» [Online]. Available: https://developer.okta.com/blog/2019/07/19/mysql-vs-postgres#postgres-advantages-over-mysql. [Zugriff am 21 September 2023].

[14] «JavaScript Single-Threaded,» [Online]. Available: https://www.geeksforgeeks.org/why-javascript-is-a-single-thread-language-that-can-be-non-blocking/. [Zugriff am 18 September 2023].

[15] «JavaScript WebWorkers,» [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers. [Zugriff am 18 September 2023].

[16] «Memory Management JavaScript,» [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_management. [Zugriff am 18 September 2023].

[17] «JavaScript JIT Compiler,» [Online]. Available: https://blog.bitsrc.io/the-jit-in-javascript-just-in-time-compiler-798b66e44143. [Zugriff am 18 September 2023].

[18] «Chess.js GitHub,» [Online]. Available: https://github.com/jhlywa/chess.js/blob/master/README.md. [Zugriff am 18 September 2023].

[19] «Chess Computers Ranking,» [Online]. Available: https://ccrl.chessdom.com/ccrl/4040/. [Zugriff am 18 September 2023].

[20] «Webassembly,» [Online]. Available: https://webassembly.org/. [Zugriff am 2 Oktober 2023].

[21] «Lichess Stockfish WebAssembly,» [Online]. Available: https://github.com/lichess-org/stockfish.wasm. [Zugriff am 18 September 2023].

[22] «Single-Sign-On,» [Online]. Available: https://auth0.com/docs/authenticate/single-sign-on. [Zugriff am 24 September 2023].

[23] «OpenIDConnect Flows,» [Online]. Available: https://darutk.medium.com/diagrams-of-all-the-openid-connect-flows-6968e3990660. [Zugriff am 24 September 2023].

[24] «Turborepo,» [Online]. Available: https://turbo.build/repo. [Zugriff am 23 September 2023].

[25] «VSCode DevContainers,» [Online]. Available: https://code.visualstudio.com/docs/devcontainers/containers. [Zugriff am 18 Oktober 2023].

[26] «NodeJS DevContainer Docker Image,» [Online]. Available: https://hub.docker.com/_/microsoft-devcontainers-javascript-node. [Zugriff am 18 Oktober 2023].

[27] «FIDE Laws of Chess,» [Online]. Available: https://www.fide.com/FIDE/handbook/LawsOfChess.pdf. [Zugriff am 16 September 2023].

[28] «Wikimedia Commons Chess Pieces,» [Online]. Available: https://commons.wikimedia.org/wiki/Category:SVG_chess_pieces. [Zugriff am 20 Oktober 2023].

[29] «API-First Design,» [Online]. Available: https://blog.developer.adobe.com/three-principles-of-api-first-design-fa6666d9f694. [Zugriff am 12 September 2023].

[30] «Forsyth-Edwards Notation,» [Online]. Available: https://en.wikipedia.org/wiki/Forsyth%E2%80%93Edwards_Notation. [Zugriff am 16 September 2023].

[31] «UML Spezifikation,» [Online]. Available: https://www.omg.org/spec/UML/2.5.1/About-UML. [Zugriff am 24 September 2023].

[32] «Board Representation,» [Online]. Available: https://www.chessprogramming.org/Board_Representation. [Zugriff am 4 Oktober 2023].

[33] «Mailbox Repräsentation,» [Online]. Available: https://www.chessprogramming.org/Mailbox. [Zugriff am 4 Oktober 2023].

[34] «Bitboard Representation,» [Online]. Available: https://www.chessprogramming.org/Bitboards. [Zugriff am 4 Oktober 2023].

[35] «How to build your own chess engine,» [Online]. Available: https://nkarve.github.io/programming/2022/06/08/chessposition.html. [Zugriff am 5 Oktober 2023].

[36] «Bitboard und Mailbox Hybrid,» [Online]. Available: https://chess.stackexchange.com/questions/39956/is-it-worth-it-to-use-a-redundant-mailbox-representation-in-addition-to-bitboard. [Zugriff am 5 Oktober 2023].

[37] «Datenstruktur Rochaderechte,» [Online]. Available: https://www.chessprogramming.org/Castling_Rights. [Zugriff am 5 Oktober 2023].

[38] «Private class fields JavaScript,» [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Private_class_fields. [Zugriff am 5 Oktober 2023].

[39] «Didactic Bitboard Chess Engine in C,» [Online]. Available: https://github.com/maksimKorzh/chess_programming/blob/master/src/bbc/bbc_1.0/bbc.c. [Zugriff am 4 Oktober 2023].

[40] «Encoding Moves,» [Online]. Available: https://www.chessprogramming.org/Encoding_Moves. [Zugriff am 5 Oktober 2023].

[41] «SAN Notation,» [Online]. Available: https://www.chessprogramming.org/Algebraic_Chess_Notation. [Zugriff am 6 Oktober 2023].

[42] «TC39 BigInt,» [Online]. Available: https://github.com/tc39/proposal-bigint. [Zugriff am 6 Oktober 2023].

[43] «JavaScript 64 Bit,» [Online]. Available: https://www.w3schools.com/js/js_bitwise.asp. [Zugriff am 7 Oktober 2023].

[44] «Chess pieces,» [Online]. Available: https://www.chessprogramming.org/Pieces. [Zugriff am 7 Oktober 2023].

[45] «Magical Bitboards and how to find them,» [Online]. Available: https://analog-hors.github.io/site/magic-bitboards/. [Zugriff am 7 Oktober 2023].

[46] «Attack and defend maps,» [Online]. Available: https://www.chessprogramming.org/Attack_and_Defend_Maps. [Zugriff am 8 Oktober 2023].

[47] «Magic Bitboards,» [Online]. Available: https://www.chessprogramming.org/Magic_Bitboards. [Zugriff am 7 Oktober 2023].

[48] «Time vs Space complexity,» [Online]. Available: https://www.geeksforgeeks.org/time-complexity-and-space-complexity/. [Zugriff am 7 Oktober 2023].

[49] «Generating magic numbers,» [Online]. Available: https://www.youtube.com/watch?v=UnEu5GOiSEs. [Zugriff am 17 September 2023].

[50] «Generating slider piece attacks,» [Online]. Available: https://m.youtube.com/watch?v=1lAM8ffBg0A. [Zugriff am 18 August 2023].

[51] «Zobrist Hashing,» [Online]. Available: https://www.chessprogramming.org/Zobrist_Hashing. [Zugriff am 2 September 2023].

[52] «PGN Files,» [Online]. Available: https://www.pgnmentor.com/files.html. [Zugriff am 18 Oktober 2023].

[53] «Colyseus Docs Rooms,» [Online]. Available: https://docs.colyseus.io/server/room. [Zugriff am 9 Oktober 2023].

[54] «Linux File Descriptor Requirements,» [Online]. Available: https://docs.oracle.com/cd/E19476-01/821-0505/file-descriptor-requirements.html. [Zugriff am 13 Oktober 2023].

[55] «Prisma Dokumentation,» [Online]. Available: https://www.prisma.io/. [Zugriff am 20 Oktober 2023].

[56] «Fastify vs Express,» [Online]. Available: https://www.inkoop.io/blog/express-vs-fastify-in-depth-comparison-of-node-js-frameworks/. [Zugriff am 19 Oktober 2023].

[57] «Colyseus Command Pattern,» [Online]. Available: https://docs.colyseus.io/best-practices/#the-command-pattern. [Zugriff am 19 Oktober 2023].

[58] «Limitationen setTimeout und setInterval,» [Online]. Available: https://www.syncfusion.com/blogs/post/settimeout-setinterval-uses-limitations.aspx. [Zugriff am 21 Oktober 2023].

[59] «React Dokumentation,» [Online]. Available: https://react.dev/. [Zugriff am 21 Oktober 2023].

[60] «Renderparadigmen Next.js,» [Online]. Available: https://nextjs.org/learn/foundations/how-nextjs-works/rendering. [Zugriff am 20 Oktober 2023].

[61] «Next.js Server Components,» [Online]. Available: https://nextjs.org/docs/app/building-your-application/rendering/server-components. [Zugriff am 20 Oktober 2023].

[62] «TailwindCSS,» [Online]. Available: https://tailwindcss.com/. [Zugriff am 20 Oktober 2023].

[63] «Prisma with Next.js,» [Online]. Available: https://www.prisma.io/docs/guides/other/troubleshooting-orm/help-articles/nextjs-prisma-client-dev-practices. [Zugriff am 20 Oktober 2023].

[64] «NextAuth OAuth Flow,» [Online]. Available: https://next-auth.js.org/configuration/providers/oauth. [Zugriff am 20 Oktober 2023].

[65] «NextAuth authenticating external APIS,» [Online]. Available: https://github.com/nextauthjs/next-auth/issues/6152#issuecomment-1363548720. [Zugriff am 16 Oktober 2023].

[66] «Description of the UCI Protocol,» [Online]. Available: https://gist.github.com/DOBRO/2592c6dad754ba67e6dcaec8c90165bf. [Zugriff am 23 Oktober 2023].

[67] «Centipawns,» [Online]. Available: https://chess.fandom.com/wiki/Centipawn. [Zugriff am 23 Oktober 2023].

11.3 Abbildungsverzeichnis

11.4 Eigenständigkeitserklärung

Hiermit bestätige ich, dass ich die vorliegende Arbeit eigenständig und ohne fremde Hilfe verfasst habe. Alle verwendeten Quellen sind vollständig und korrekt im Quellenverzeichnis aufgeführt und im entsprechenden Text zitiert. Es wurden keine anderen Hilfsmittel verwendet als die gekennzeichneten. Alle wörtlichen und sinngemässen Übernahmen aus externen Quellen sind klar als solche gekennzeichnet und mit korrekten Quellenangaben versehen. Des Weiteren erkläre ich, dass diese Arbeit nicht im Rahmen einer anderen Prüfung vorgelegt wurde.

Timo Jokinen, 24.10.2023

11.5 Klassifizierung

Vorliegende Arbeit ist als öffentlich klassifiziert und darf mit Quellenangabe von Teko, sowie der Allgemeinheit publiziert und referenziert werden.

11.6 Protokolle

Während der Arbeit wurden zwei Termine mit dem Diplomcoach durchgeführt, um den Fortschritt zu besprechen und Fragen des Autors zu klären. Folgend sind Protokolle der Termine aufgeführt (abgekürzt und nicht Wort für Wort).

11.6.1 Erster Termin (15.09.2023)

Beim ersten Termin hat der Autor dem Diplomcoach vor allem Fragen gestellt. Neben den Fragen wurde der Fortschritt vom Produkt präsentiert. Zu diesem Zeitpunkt sind keine kritischen Probleme aufgetaucht, die den Erfolg des Projekts gefährden würden. Folgende Fragen wurden gestellt:

Autor: Ist es in Ordnung die Dokumentation in LaTex zu schreiben:
Diplomcoach: Kein Problem.

Autor: Das Projekt wird nicht wasserfallmässig umgesetzt. Da es sich um ein grosses Projekt handelt, ist es schwierig von Anfang an jedes Detail zu planen. Zum Start sind möglicherweise noch gar nicht alle Informationen bekannt und ergeben sich erst im Verlauf der Arbeit. Deshalb wurde das Projekt nach einem agilen Ansatz umgesetzt. Ist das in Ordnung?
Diplomcoach: Muss in der Arbeit festgehalten werden.

Autor: In den Evaluationskriterien gibt es den Punkt «Berechnung / Wirtschaftlichkeit». In meinem Projekt ist zunächst kein Geld im Spiel und die Wirtschaftlichkeit wird nicht in Betracht gezogen, da es um die technische Umsetzung geht und das Projekt nicht für einen bestimmten Kunden entwickelt wird.
Diplomcoach: Evaluationskriterien sind bereichsübergreifend und somit treffen nicht alle Punkte auf jeden Bereich zu.

Autor: Wird der Code bewertet?
Diplomcoach: Nein, nur die Dokumentation. Die Dokumentation soll die Arbeit widerspiegeln, indem beschrieben wird, was für Gedanken gemacht und was für Muster angewendet wurden.

Autor: Über ein Thema zu schreiben wie beispielsweise eine bestimmte Technologie zu schreiben ist schwierig, wenn angenommen wird, dass der Leser keine Erfahrung mit der jeweiligen Technologie hat. Wie wird die Verständlichkeit bewertet? Ich kann nicht die gesamte Dokumentation einer Bibliothek einbinden.
Diplomcoach: Wichtige Stellen müssen dokumentiert werden, sodass das Verständnis gegeben ist. Es muss nicht jede Zeile einzeln erklärt werden.

Autor: Darf ChatGPT für Research verwendet werden? Muss das ausgewiesen werden?
Diplomcoach: Stellen in der Dokumentation, zum Beispiel Text oder Programmcode, muss markiert werden, wenn es von einem Model generiert wurde. Für Research allgemein, ist es kein Problem, sollte aber irgendwo in der Arbeit notiert sein.

Autor: Muss die Arbeit deployed werden? Vielleicht reicht das zeitlich nicht und ist nicht in meinen Erfolgskriterien / Zielen.
Diplomcoach: Nein.

11.6.2 Zweiter Termin (19.10.2023)

Beim zweiten Termin hat der Autor mit dem Diplomcoach die Arbeit überflogen und der Diplomcoach hat Feedback zu dem gegeben, was er gesehen hat. Es sind keine Probleme oder Limitationen aufgetaucht, die den Projekterfolg gefährden. Der Autor war sich zu diesem Zeitpunkt sicher, dass die Arbeit erfolgreich abgeschlossen werden kann. Das Feedback des Diplomcoach war positiv und es sind keine nennenswerten Punkte aufgetaucht. Folgende Fragen wurden gestellt:

Folgende Fragen wurden gestellt:
Autor: Was muss genau in welcher Form abgegeben werden?
Diplomcoach: Qualifikationsprofil und Diplomarbeit digital.

Autor: Müssen Instruktionen, um den Programmcode laufenzulassen inkludiert werden?
Diplomcoach: Wäre vorteilhaft.

Autor: Programmcode in der Doku kann manchmal auf eine neue Seite umgebrochen werden. Sollen die Seiten so formatiert werden, dass das nicht geschieht?
Diplomcoach: Seitenumbruch sollte kein Problem sein bei grösseren Abschnitten. Wenn es sich anbietet, lieber etwas leerer Platz lassen.

Autor: Müssen die Anforderungen bei der Auswertung referenziert werden?
Diplomcoach: Ja.

Autor: Beispielsweise in einem Datenbankschema, muss jedes Attribut erklärt werden?
Diplomcoach: Nein, nur wichtige Punkte für das Verständnis.

Autor: Wenn in einem Abschnitt vorgegriffen wird, auf ein Thema, das in der Dokumentation später behandelt wird, wie soll umgegangen werden?

Diplomcoach: Referenzieren auf zukünftiges Kapitel

Autor: Für die Analyse der Plattform wurden nur Werkzeuge eingesetzt, die ich für sinnvoll hielt. Ist es ein Problem, wenn die Analyse nicht nach Schulbuch aus dem Fach «System- und Softwareengineering» abläuft?
Diplomcoach: Kein Problem.

11.7 Quellcode

Der gesamte Quellcode ist verfügbar in einem öffentlichen Git-Repository:
https://github.com/timojokinen/diplomarbeit-multiplayer-chess-website