Hochschule für
Technik und Wirtschaft
Dresden
University of Applied Sciences

Fakultät Informatik/Mathematik

Diplomarbeit

im Studiengang Allgemeine Informatik

Thema:

Webentwicklung mittels Kotlin am Beispiel eines RESTful-Schachservers

Eingereicht von:	Felix Dimmel
Eingreicht am:	22. Mai 2018
Betreuer:	Prof. Dr.-Ing. Jörg Vogt
2. Gutachter:	Prof. Dr.-Ing. Arnold Beck

Inhaltsverzeichnis

KAPITEL 1

Einleitung

Laut einer Onlinestudie der ARD/ZDF-Medienkommission von 2017 [ARD17a; ARD17b] ist die Nutzung des Internets im letzten Jahr um 6% in Deutschland gestiegen. Dieser große Bedeutungszuwachs allein in Deutschland zeigt auch die immer mehr zunehmende Nachfrage nach Webinhalten. Was wiederum auch ein Indiz für die große Beliebheit der Skriptsprache JavaScript sein könnte, denn diese ist laut der letzten jährlichen Onlineumfrage der Stack Exchange Inc. [Inc17] auf Platz 1 der beliebtesten Programmiersprachen. Dennoch kann es mit unter schwierig werden große Applikationen mit JavaScript zu entwickeln. An dieser Stelle kommt die noch sehr junge Programmiersprache Kotlin ins Spiel.

Kotlin ist eine statisch typisierte objektorientierte Programmiersprache, welche in Java-Bytecode, in JavaScript und in Nativen Code kompiliert werden kann. Desweiteren ist Kotlin neben Java seit Mai 2017 offizielle Programmiersprache für Android [Sha17]. Dadurch ist Kotlin sehr vielseitig und lässt sich für viele Anwendungsfälle einsetzen.

Da aber Kotlin wie schon oben erwähnt noch sehr jung ist, soll diese Arbeit dazu genutzt werden zu erforschen wie sich diese Programmiersprache für Server- bzw. Client-Anwendungen eignet bzw. welche Probleme während der Umsetzung aufgetreten sind.

TODO:
- Kapitelverlauf erläutern
- Vorraussetzungen

KAPITEL 2

Grundlagen

In diesem Kapitel sollen die Grundlagen, welche zum besseren Verständnis notwendig sind, behandelt werden. Dabei dient der erste Abschnitt zur Erläuterung von Representational State Transfer (REST)-APIs. Wobei auf die Definition, die Vor- und Nachteile einer API im allgemeinen eingegangen wird. Im Sinne der REST-APIs werden weiterhin deren Qualitätsmerkmale und das Design-Konzept HATEOAS genauer erläutert. Der zweite Abschnitt dient zur näheren Betrachtung des Build-Tools Gradle, welches für die Umsetzung des Praktischen Teils dieser Arbeit verwendet wurde. Eine genauere Betrachtung der Schachnotationen Forsyth-Edwards-Notation (FEN) und Standard Algebraic Notation (SAN) sollen Inhalt des dritten Abschnittes sein. Im letzten Kapitel wird kurz auf die Schachregeln verwiesen.

2.1 REST-API

REST ist ein von Roy Fielding entwickelter Architekturstil, welchen er in seiner Dissertation [Fie00] beschrieb. Dabei geht er ebenfalls auf eine Reihe von Leitsätzen und Praktiken ein, welche sich in System auf Basis von Netzwerken bewährt haben.

Laut [Spi16, S. 143] unterstützt der REST Architekturstil eine Reihe von Protokollen, mit welchen solcher umgesetzt werden kann. Der bekannteste bzw. am häufigsten verwendete Vertreter ist dabei das Protokoll Hypertext Transfer Protocol (HTTP). Es wird dabei REST im Zusammenhang mit HTTP als RESTful HTTP bezeichnet.

Da in dieser Arbeit die Webentwicklung im Vordergrund steht und HTTP einer der wichtigsten Standards im Web ist, soll im nachfolgenden Verlauf REST immer im Sinne von RESTful HTTP verstanden werden.

In den nachfolgenden Abschnitten sollen die allgemeine Definition, die Vor- und Nachteile und die Qualitätsmerkmale einer API näher beleuchtet werden. Anschließend wird auf die Grundprinzipien von REST detaillierter eingegangen. Als letztes wird das Konzept HATEOAS, welches laut Roy Fielding [Fie08] ein Muss für jede RESTful API ist, genauer erläutert.

Die nachfolgenden Unterkapitel basieren auf dem Buch „API-Design“ [Spi16, S. 7–10, 13–14, 144–148, 189] von Kai Spichale.

2.1.1 Allgemeine Definition einer API

Laut [Spi16, S. 7] definiert Kai Spichale eine API mit den Worten von Joshua Bloch wie folgt: „Eine API spezifiziert die Operationen sowie die Ein- und Ausgaben einer Softwarekomponente. Ihr Hauptzweck besteht darin, eine Menge an Funktionen unabhängig von ihrer Implementierung zu definieren, sodass die Implementierung variieren kann, ohne die Benutzer der Softwarekomponente zu beeinträchtigen“. Des weiteren unterteilt dieser die APIs in zwei Kategorien ein, in Programmiersprachen- und Remote-APIs. Des weiteren definiert er die APIs der Programmiersprachen als abhängig und die Remote als unabhängig gegenüber der Sprache und der Plattform.

2.1.2 Vorteile einer API

Stabilität durch lose Kopplung

Mithilfe von APIs soll die Abhängigkeit zum Benutzer minimiert werden, der dadurch nicht mehr stark an die Implementierung gekoppelt ist. Das ermöglicht eine Veränderung der eigentlichen Implementation einer Softwarekomponente, ohne das der Benutzer davon etwas bemerkt.

Portabilität

Es ist möglich für unterschiedliche Plattformen eine einheitliche Implementierung einer API bereitzustellen, obwohl diese im inneren verschieden implementiert sind. Ein bekanntest Beispiel ist dabei die Java Runtime Enviroment (JRE), welches diese Funktionalität für Java-Programme bereitstellt.

Komplexitätsreduktion durch Modularisierung

Der API-Benutzer besitzt in erster Linie keine genauen Informationen über die Komplexität der Implementierung. Diese Tatsache folgt dem Geheimprinzip und soll der Beherrschung großer Projekte in Hinsicht ihrer Komplexität dienen. Zusätzlich bringt dieser Aspekt auch einen Wirtschaftlichen Vorteil, denn durch die Modularisierung ist eine bessere Arbeitsteilung möglich, was wiederum Entwicklungskosten sparen kann.

Softwarewiederverwendung und Integration

Neben dem verbergen von Details zur Implementierung sollten APIs Funktionen einer Komponente leicht verständlich bereitstellen, um API-Nutzern eine einfache Integration bzw. Verwendung zu

ermöglichen. Aus diesen sollten APIs auch dahingehend optimiert werden.

2.1.3 Nachteile einer API

Interoperabilität

Ein Nachteil, der allerdings nur Programmiersprachen-APIs betrifft, ist die Interoperabilität zu anderen Programmiersprachen. Beispielsweise kann ein Programm, welches in Go geschrieben wurde, nicht auf die Java-API zugreifen. Als Problemlösung stehen hierbei aber die Remote-API bereit. Diese arbeiten mit Protokollen wie HTTP oder Advanced Message Queuing Protocol (AMQP), welche sprach- und plattformunabhängig sind[1].

Änderbarkeit

Dadurch das geschlossene API-Verträge mit den Benutzern nicht gebrochen werden sollten, kann es hinsichtlich der Änderbarkeit zu Problemen kommen. Das ist aber nur der Fall sofern die Benutzer nicht bekannt sind oder nicht kontrolliert werden können. In so einem Fall spricht man von veröffentlichten APIs. Als Gegenstück dazu können interne APIs betrachtet werden, denn bei diesen wäre eine Kontrolle der Benutzer möglich.

2.1.4 Qualitätsmerkmale

Benutzbarkeit

Als Zentrale Anforderung an einem guten Design für API stehen die leichte Verständlichkeit, Erlernbarkeit und Benutzbarkeit für andere Nutzer. Um diese Anforderung umzusetzen haben sich ein Reihe von allgemeinen Eigenschaften/Zielen etabliert. Eine Auflistung, inklusive einer kurzen Beschreibung der jeweiligen Eigenschaft, werden in der Tabelle 2.1 aufgeführt. Für detaillierte Informationen zu die einzelnen Eigenschaft, können diese in [Spi16, S. 14–23] nachgeschlagen werden.

Effizienz

Unter dem Qualitätsmerkmal Effizienz kann zum Beispiel der geringe Verbrauch von Akku oder dem Datenvolumen bei Mobilen Geräten verstanden werden. Oder aber die Skalierbarkeit einer API, welches bei einem großen Zuwachs von Aufrufen durchaus entscheidend sein kann.

1 siehe Kapitel 2.1.1

Tabelle 2.1: Eigenschaften/Ziele des Qualitätsmerkmals „Benutzbarkeit“
(verändert nach [Spi16, S. 14–23])

Eigenschaft/Ziel	Beschreibung
Konsistent	Gemeint ist damit das Entscheidungen hinsichtlich des Entwurfs einheitlich im Quellcode angewandt werden. Das betrifft beispielsweise die Namensgebung von Variablen oder Funktionen.
Intuitiv verständlich	Für diese Ziel ist eine Konsistente API unter Verwendung von Namenskonventionen unabdingbar. Grundlegend kann dabei gesagt werden das gleiche Dinge einheitliche Namen, aber auch ungleiche Dinge unterschiedliche Namen haben sollten. So können bereits durch bekannte Funktionen Rückschlüsse auf andere unbekannte gezogen werden. Ein konkretes Beispiel dafür sind die *getter*- und *setter*-Methoden in der Java-Welt.
Dokumentiert	Eine ausführliche Dokumentation mit Beschreibungen zu den einzelnen Klassen, Methoden und Parametern, ist für eine einfache Benutzung dringend erforderlich. Zusätzlich sollten Beispiele die Erläuterungen unterstreichen.
Einprägsam und leicht zu lernen	Damit eine API einfach zu erlernen ist, sind die ersten drei Punkte dieser Tabelle unabdingbar. Wichtig ist die Einstiegshürden gering zu halten, um potenzielle Nutzer nicht abzuschrecken. Prinzipiell ist es von Vorteil wenn mit relativ wenig Code erste Ergebnisse erzielt werden können.
Lesbaren Code fördern	Eine API kann großen Einfluss auf die Lesbarkeit des Client-Codes haben und diesen so signifikant verbessern. Durch eine gute Lesbarkeit können zum Beispiel besser bzw. schneller Fehler entdeckt werden. Um den Client-Code möglichst schmal zu halt sollte die API Hilfsmethoden anbieten, wobei gilt das der Client nichts tun müssen sollte was ihm durch die API abgenommen werden könnte.
Schwer falsch zu benutzen	Unerwartetes Verhalten aus Sicht den Nutzers sollte vermieden werden, um unerwartete Fehler zu vermeiden und um die API intuitiv zu halten.
Minimal	Prinzipiell gilt es eine API so klein wie möglich zu halten und Funktionen welche nicht unbedingt benötigt werden im zweifel weg zu lassen. Denn nachträglich können solche Elemente nicht mehr trivial entfernt werden. Außerdem lässt sich sagen das je größer die API desto größer die Komplexität.
Stabil	Durch Stabilität wird sicher gestellt das Änderungen keine Auswirkung auf alte Benutzer hat. Wenn auftretende negative Auswirkungen unvermeidbar sind, sollten diese entweder ausführlich kommuniziert oder eine neue Version benutzt werden.
Einfach erweiterbar	Für die Erweiterbarkeit einer API ist der Aufwand von Änderungen von Clients zu beachten. Wenn der Client nach einer Erweiterung nicht angepasst werden muss, ist dabei der Idealfall. Durch Vererbung kann beispielsweise so ein Verhalten erreicht werden.

Zuverlässigkeit

Unter der Zuverlässigkeit einer API kann die Fehleranfälligkeit verstanden werden bzw. wie gut diese auf Fehler reagieren kann. Ein wichtiger Aspekt der dabei auf jeden Fall beachtet werden sollte ist die Rückgabe von standardisierten HTTP-Statuscodes. Dies ermöglicht dem Benutzer ein ordentliches Feedback, welches noch durch konkretisierte Fehlermeldungen verdeutlicht werden sollte.

2.1.5 Grundprinzipien von REST

Eindeutige Identifikation von Ressourcen

Für jede Ressourcen muss eine eindeutige Identifikation definiert werden. Im Web stehen dabei Uniform Ressource Indetifiers (URIs) für diesen Zweck bereit. Die Wichtigkeit dieses Prinzips liegt nahe. Wenn beispielsweise Produkte von einem Online-Shop nicht eindeutig identifiziert werden könnten, dann wäre das zum Beispiel für Werbezweck mehr als unpraktisch. E-Mails mit personalisierter Werbung für Produkte wäre ohne eine eindeutige Identifikation dieser nicht bzw. sehr umständlich möglich.

Wichtig ist zu beachten das mit URIs nicht nur einzelne Ressourcen identifiziert, sondern auch Ressourcenlisten werden können. Um bei dem Beispiel von oben zu bleiben, könnte es eine URI geben mit welcher ein einzelnes Produkt und eine mit welcher eine Liste von Produkten identifiziert wird. Dies ist kein Widerspruch gegen das Prinzip, welches besagt das jede Ressource eindeutig identifiziert werden muss, sondern in diesem Fall gilt eine Liste als selbstständige Ressource.

Verwendung von Hypermedia

Hypermedia als Begriff ist Zusammensetzung aus den Begriffen *Hypertest* und *Multimedia*. Dabei kann *Hypermedia* als Oberbegriff von *Hypertext* betrachtet werden, denn *Hypermedia* unterstützt nicht nur Texte, sondern auch andere Multimediale Inhalte wie zum Beispiel Dokumente, Bilder, Videos oder Links. Letzteres kann für das Ausführen von Funktionen oder zum Navigieren innerhalb des Browsers verwendet werden und ist ein bekanntes Element in der Hypertext Markup Language (HTML). Daher kann HTML auch als klassischer Vertreter von Hypermediaformaten betrachtet werden. Damit ein Client weiß welche Aktionen bzw. welchen Pfaden folgen kann, werden ihm diese vom Server mithilfe von Hypermedia zur Verfügung gestellt.

Verwendung von HTTP-Standardmethoden

Um die vom Server bereit gestellten Links ordnungsgemäß auszuführen, müssen neben den URIs auch einheitliche Schnittstellen bekannt sein. Das setzt voraus, dass alle Clients über Verwen-

dung und Semantik der Schnittstellen Bescheid wissen. An dieser Stelle kommen die HTTP-Standardmethoden zum Einsatz. Die Schnittstellen von HTTP bestehen dabei im wesentlichen aus den Funktionen *GET*, *HEAD*, *POST*, *PUT* und *DELETE*, welche alle in der HTTP-Spezifikation [Fie] definiert sind. Die Methoden *PATCH* oder auch *LINK* waren dabei nicht von Anfang in der Spezifikation enthalten, sonder wurden erst in nachträglich hinzugefügt.

Ein Client kann mithilfe der Methode *GET*, ohne genaues Wissen der Ressource, eine Repräsentation dieser abfragen, denn diese allgemeinen Schnittstellen werden für jede Ressource verwendet. Durch diese mögliche Vorhersagbarkeit sind die Qualitätsmerkmale aus den Kapiteln 2.1.4 erfüllt. Zusätzlich ist noch anzumerken das durch Benutzung der Methode *GET* keine unerwünschten Effekte befürchtet werden müssen, denn diese ist idempotent. Einfach ausgedrückt kann ein lesender Zugriff auf eine Ressource keine Änderung des Zustands hervorrufen.

Unterschiedliche Repräsentationen von Ressourcen

Um mit den Daten, welcher eine API zurückgibt, umzugehen muss der Client wissen in welchem Format er die Daten zurückbekommen möchte. Dazu muss er das gewünschte Format mithilfe von HTTP Content Negotiation von der API abzufragen. Dabei besteht die Möglichkeit mehrere gewünschte Formate, mit unterschiedlicher Priorität, anzugeben. Der Server gibt anschließend das Format zurück, welches Kompatibel und die Höchste Priorität besitzt. Die gängigste Methode ist dabei die gewünschten Formate im *Accept-Header* des Requestes mitzusenden.

Statuslose Kommunikation

Das letzte Grundprinzip besagt das es keinen Sitzungsstatus, welcher auf dem Server über mehrere Anfragen gehalten wird, geben darf. Der Kommunikationsstatus muss demzufolge in der Ressource selber oder im Client gespeichert werden. Durch die Statuslose Kommunikation wird die Kopplung zwischen Server und Client verringert. Das wiederum bringt den Vorteil das beispielsweise ein Neustart des Servers den Client nicht stören oder sogar zum Absturz bringen kann, denn dieser würde es gar nicht erst mitbekommen. Ein weiterer nicht unerheblicher Vorteil welcher daraus resultiert ist die Möglichkeit der Lastenverteilung auf unterschiedliche Serverinstanzen.

2.1.6 HATEOAS

„Hypermedia As The Engine Of Application State“ oder kurz HATEOAS ist ein Design-Konzept, welches von vielen APIs die sich selber als RESTful bezeichnen missachtet wird. Diesen Begriff bzw. diese Aussage kann nach Kai Spichale mit nachfolgenden Bedeutungen beschrieben werden [Spi16, S. 156]:

- „»Hypermedia« ist eine Verallgemeinerung des Hypertexts mit multimedialen Anteilen. Beziehungen zwischen Objekten werden durch Hypermedia Controls abgebildet.“
- „Mit »Engine« ist ein Zustandsautomat gemeint. Die Zustände und Zustandsübergänge der »Engine« beschreiben das Verhalten der »Application«.“
- „Im Kontext von REST kann man »Application« mit Ressource gleichsetzen.“
- „Mit »State« ist der Zustand der Ressource gemeint, deren Zustandsübergänge durch die »Engine« definiert werden.“

Grundgedanke hinter diesen Konzept ist die Selbstbeschreibung einer API. Das Ziel soll dabei sein, dass Clients nicht genau über die API Bescheid wissen müssen, sondern die Ressource dem Client zeigt wie er durch die API navigieren kann. Als Resultat des ganzen benötigt der Client nur das Wissen über die Uniform Ressource Locator (URL) des Einstiegs, alles weitere bekommt er vom Server in Form von Links mitgeteilt.

Durch das Konzept HATEOAS wird ein dynamischer Workflow erzeugt, wodurch die Ressource volle Kontrolle über die API bekommt. Damit ist es beispielsweise möglich anhand von Clients unterschiedliche Links bzw. Navigationspunkte auszuliefern oder Links auszutauschen ohne das der Client davon etwas bemerkt. Um diese Vorteile zu nutzen ist es natürlich notwendig die bereitgestellten Links zu verwenden.

2.2 Das Build-Tool Gradle

Gradle ist ein Open-Source Build-Tool, welches in erster Linie für die Java-Welt entwickelt wurde, aber mittlerweile auch andere Sprachen wie zum Beispiel Kotlin unterstützt. Es basiert auf den Erfahrungen von anderen großen Build-Tools wie Ant und Maven und hat damit große Akzeptanz gefunden. Indiz dafür ist der Wechsel zu Gradle von einigen großen und bekannten Projekten, wie zum Beispiel Android oder dem Spring Framework.

Die in diesem Kapitel behandelten Fakten und Erklärungen entstammen aus dem Buch „Introducing Gradle“ von Balaji Varanasi und Sudha Belida. (vgl. [Var15])

2.2.1 Eigenschaften von Gradle

Declarative Dependency Management

Gradle biete eine komfortable Möglichkeit mit Abhängigkeiten umzugehen. Denn viele Projekte verwenden andere Bibliotheken oder Frameworks, welche gegebenenfalls eigenen Abhängigkeiten aufweisen. Dadurch kann es sehr mühselig werden die ganze Abhängigkeitsstruktur mit alle Versionen zu verwalten. Es müssen nur die benötigten Abhängigkeiten definiert werden und Gradle

kümmert sich um den Download dieser, inklusive aller Unterabhängigkeiten. Es ist wichtig zu wissen das Gradle nur das *Was* aber nicht das *Wie* gesagt werden muss.

Declarative Builds

Damit Skripte für den Build-Prozess einfach und verständlich sind, verwendet Gradle dafür eine Domain-specific language (DSL) auf Basis der Programmiersprache Groovy. Dadurch werden eine Reihe von Sprach-Elementen bereitgestellt, welche einfach zusammengestellt werden können und ihre Absicht klar zum Ausdruck bringen.

Build by Convention

Um den Konfigurationsaufwand von Projekten zu minimieren bietet Gradle eine Reihe von Standardwerten und Konventionen an. Durch die Einhaltung dieser werden Build-Skripte sehr prägnant, sie sind aber dennoch nicht daran gebunden. Da die Skripte auf Groovy basieren können die Konventionen leicht, durch schreiben von Groovy-Code, umgangen werden.

Incremental Builds

In größeren Projekten kommt es oft zu sehr langsame Build-Zeiten, weil andere Tools im Gegensatz zu Gradle versuchen den Code immer zu säubern und neu aufzubauen. Durch die inkrementellen Builds, welche Gradle bereitstellt, kann dieses Problem umgangen werden. Tasks die der Build-Prozess ausführt, werden sofern keine Änderungen festgestellt wurden übersprungen. Dafür wird überprüft ob sich die Ein- oder Ausgänge geändert haben.

Gradle Wrapper

Mithilfe dieses Features ist es möglich das Projekt zu bauen obwohl keine Installation von Gradle vorhanden ist. Der Gradle Wrapper stellt dafür eine Batch-Datei für Windows- und ein Shell-Skript für Linux- bzw. Mac-Umgebungen bereit. Ein weiterer Vorteil ist die Erstellung neuer Continuous Integration (CI) Server, welche die Build-Prozesse somit ohne zusätzliche Konfiguration ausführen können.

Plugins

Mithilfe von Plugins bzw. Erweiterungen können die Funktionalitäten von Gradle beliebig erweitert oder angepasst werden. Dabei dienen die sie als Kappselung von Build- oder Task-Logik und können bequem verteilt bzw. in anderen Projekten wieder verwendet werden. Somit ist beispielsweise eine Unterstützung einer zusätzlichen Programmiersprache ohne weitere möglich.

2.2.2 Verwaltung von Projekten und Tasks

Innerhalb von Gradle wird zwischen zwei grundlegenden Build Bereichen unterschieden, den Projekten und den Tasks. Gradle kann dabei für ein oder mehrere Projekte verwendet werden, wobei jedes Projekt wiederum ein oder mehrere Tasks beinhaltet.

Projekte

Standardmäßig wird innerhalb des Projektverzeichnisses nach der Datei *build.gradle* gesucht, welche ein Projekt identifizieren und in welcher die benötigten Tasks definiert werden. Es ist dabei möglich die *build.gradle* Datei umzubenennen, dadurch muss aber bei einem Aufruf dieser explizit mit angegeben werden.

Zusätzlich neben den Tasks können weitere Informationen, wie zum Beispiel die Abhängigkeiten oder die Repositorys, aus welchen Gradle die Abhängigkeiten beziehen soll, definiert werden. Auch die Angabe des Namens, einer Version, einer Beschreibung oder eines Elternprojektes ist möglich.

Tasks

Die Tasks sind das Kernstück eines jeden Projektes, mithilfe dieser lässt sich der geschriebene Quellcode kompilieren, Tests starten oder fertig zusammengebaute Applikation auf einen Server veröffentlichen.

Jeder Task verfügt dabei über eine Funktion *doFirst* und eine Funktion *doLast* um Funktionalitäten vor oder nach einem Task auszuführen. Diese können beispielsweise auch in, durch Plugins bereitgestellte, Tasks verwendet werden, um in den Prozess einzugreifen ohne die eigentliche Logik zu verändern. Mithilfe der Funktion *dependsOn* können Abhängigkeiten zu anderen Tasks definiert werden, welche zuvor ausgeführt werden sollen. Des weiteren besteht die Möglichkeit bestimmte Typen für Tasks anzugeben. So können diese standardmäßig als *Zip*, [Copy], *Exec* oder *Delete* definiert werden. *Copy* erlaubt dabei zum Beispiel das Kopieren von Dateien oder *Exec* das Ausführen von Kommandozeilen-Befehle.

In dem Listing 2.1 wird ein Beispiel für ein kopierenden Task gezeigt. Dieser kopiert alle Dateien mit den Endungen *.js* und *.js.map*, aus dem Build-Verzeichnis der Source-Dateien, in ein Unterverzeichnis *js* des *dist*-Verzeichnisses. Dabei ist außerdem eine Abhängigkeit zum Build-Task definiert, wodurch dieser immer zuvor ausgeführt wird.

```
task copyCompiledFilesIntoDist(type: Copy) {
  from "$buildDir/classes/kotlin/main/"
  into "${projectDir}/dist/js/"
  include "*.js"
  include "*.js.map"
}

copyCompiledFilesIntoDist.dependsOn(build)
```

Listing 2.1: Beispiel: Gradle-Task

2.3 Schachnotationen FEN und SAN

FEN und SAN sind Notationen welche für die elektronische Verarbeitung von Schachspielen entwickelt wurden. Beide Notationen werden als Text dargestellt und ermöglichen so eine Statuslose Kommunikation. Die FEN dient dabei zur Repräsentation eines ganzen Schachbrettes und die SAN zur Darstellung eines Zuges.

Die FEN setzt sich dabei aus mehreren Teilen zusammen, welche aus der Figurenstellung, dem aktuellem Spieler, der Möglichkeit zur Rochade, dem Feld zum schlagen „en passant“, den Halbzügen und der aktuellem Zugnummer bestehen. Einzelne Teile werden dabei durch Leerzeichen getrennt. Die Abbildung 2.1 zeigt dabei die FEN einer Startposition.

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

Abbildung 2.1: Startposition eines Schachspiels in der FEN

Die SAN besteht im allgemeinen aus dem Buchstaben der bewegenden Spielfigur und dem Zielfeld. Für eine Bauernfigur wird kein Buchstabe angegeben. Die Abbildung 2.2 zeigt dabei den Zug eines Bauern vom Feld *a2* nach *a4* und die Abbildung 2.3 den eines Springers von b1 nach c3.

a4

Abbildung 2.2: Beispiel SAN: Bauer zieht von a2 nach a4

Nc3

Abbildung 2.3: Beispiel SAN: Spring zieht von b1 nach c3

Die Buchstabencodes der einzelnen Figuren sind in der Tabelle 2.2 aufgelistet. Dabei ist zu beachten das die Codes kleingeschrieben für die Farbe Schwarz und groß für die Farbe Weiß stehen. In der SAN werden sie aber immer groß geschrieben. Genauere Informationen zur FEN und zur SAN können in der Bachelorarbeit [Kre15, S. 9–10] von Christoph Kretzschmar nachgelesen werden.

Tabelle 2.2: Figurenbedeutung in der FEN und SAN (Quelle: [Kre15, Tabelle 2.1])

	R	**N**	**B**	**Q**	**K**	**P**
Bedeutung	Rook	Knight	Bishop	Queen	King	Pawn
Figur	Turm	Springer	Läufer	Königin	König	Bauer

2.4 Schachregeln

Für das Reibungslose Verständnis dieser Arbeit empfiehlt es sich Wissen über die Regeln des Spiels Schach zu besitzen. Da eine detaillierte Erläuterung dieser aber den Rahmen der Arbeit sprengen würde, können sämtliche Regeln in dem Buch [Los08] nachgeschlagen werden. In diesem Buch sind neben den allgemeinen, auch die Bewegungsregeln der einzelnen Figuren niedergeschrieben. Zusätzlich enthält es noch eine Reihe von Tipps zum Einstieg und ein paar Testfragen zu bekannten Partien, mit beigefügten Lösungen.

KAPITEL 3

Vergleich zwischen Kotlin und dem Google Web Toolkit

KAPITEL 4

Konzept des Servers

Inhalt dieses Kapitels soll die Planung sein, welche für die Umsetzung des RESTful Schachservers benötigt wird. Dabei dient der erste Abschnitt für eine Erläuterung der Anforderungen, welche der Server mitbringen bzw. erfüllen soll. Enthalten ist dabei auch eine Erläuterung der benötigten Ressource. Im zweiten Abschnitt dieses Kapitels befasst sich anschließend damit, wie der Zugriff auf einzelne Ressourcen des REST-Server erfolgen soll. Dabei werden alle möglichen Request-Methoden für die jeweiligen Ressourcen näher beleuchtet.

4.1 Anforderungen

Die Grundanforderungen an den RESTful Schachserver sollen in erster Linie die Bereitstellung aller benötigten Ressourcen sein. Dabei sollen Elemente erstellt, ggf. bearbeitet und gelöscht werden können. Zusätzlich soll die Möglichkeit bestehen, einzelne oder alle gespeicherten Elemente einer Ressource abzufragen. Beim erstellen eines neuen Ressourcenelements soll dieses in einer SQLite Datenbank gespeichert und die ID automatisch durch SQLite generiert werden.

Um ein Schachspiel abzubilden bedarf es dabei der Ressourcen Player (Spieler), Match (Partie) und Draw (Zug), welche in den nachfolgenden Unterabschnitten 4.1.1 bis 4.1.3 näher betrachtet werden.

Als abschließende Anforderung ist noch die Fehlerresistenz zu erwähnen. Denn die im Rahmen dieser Arbeit entstandene Praktikumsaufgabe

Verweis auf Praktikumsaufgabe im Anhang

soll durch zukünftige Studenten absolviert werden, wobei der Server als Grundlage dienen soll.

Zur Unterstützung der Erläuterungen in den Kapiteln 4.1.1 bis 4.1.3 bietet die Abbildung 4.1, in Form eines Unified Modeling Language (UML) Klassendiagramms, eine Veranschaulichung.

4.1.1 Ressource: Player (Spieler)

Neben der ID, welche schon im Abschnitt 4.1 erwähnt und durch die SQLite Datenbank generiert werden soll, besitzt der Player noch Informationen über seinen Name und sein Passwort.

Nach dem anlegen eines neuen Players soll eine Änderung des Namens nicht gestattet sein, die des Passwortes hingegen schon.

4.1.2 Ressource: Match (Partie)

Neben der ID besitzt ein Match Informationen über die beiden Spielteilnehmer und deren Figurenstellung auf dem Schachbrett. Zusätzlich wird registriert welcher der beiden Player als nächstes seinen nächsten Zug tätigen muss, welche Möglichkeiten zum rochieren bestehen, ob ein Schlag „en passant“ möglich ist und wenn ja auf welches Feld gezogen werden muss und wie viele Halbzüge gespielt wurden. Der Wert der Halbzüge wird zurückgesetzt sobald eine Bauernfigur gezogen oder eine beliebige Figur geschmissen wurde. Zusätzlich kann über ein Match ermittelt werden ob ein Spieler im Schach steht oder ob das Spiel schon bis zum Schachmatt gespielt wurde. All diese Informationen werden zusätzlich noch als String in der FEN[1] gespeichert.

4.1.3 Ressource: Draw (Zug)

Die Ressource Draw speichert zusätzlich zur ID die Farbe des Spielers, die Art der Spielfigur, Start- und Endfeld des Zuges, ob eine Figur geschlagen wurde, wenn ja ob durch en passant und ob seitens der Dame oder des König rochiert wurde. Die Informationen werden ähnlich zum Match als String gespeichert, aber in diesem Fall in der SAN[1].

1 siehe Kapitel 2.3

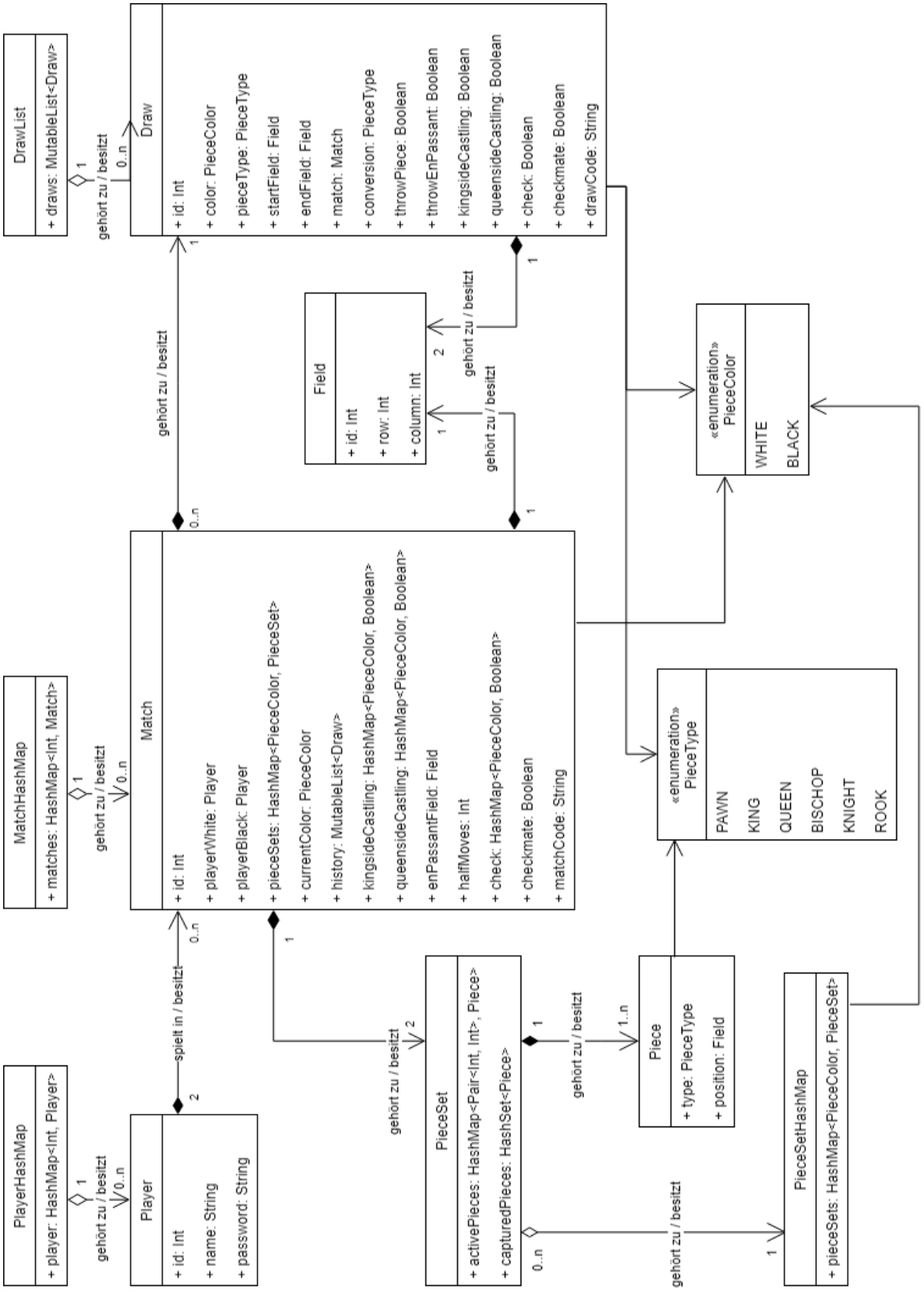

Abbildung 4.1: Klassendiagramm: Modelle des Servers

4.2 Ressourcenzugriffe mithilfe von Controllern

Die einzelnen Zugriffe auf die Ressourcen werden in den Kapiteln 4.2.1 bis 4.2.3 nach ihrer Art bzw. deren Aufgaben in einzelne Controller unterteilt, um eine gute Übersicht zu wahren. Für alle Einstiegspunkte der REST-API soll die Request-Methode „OPTIONS“ bereitstehen, über welchen ermittelt werden kann welche Methoden für den jeweiligen Einstiegspunkt zur Verfügung stehen.

Etwaige Requestparameter sollen in dem Format JavaScript Object Notation (JSON) oder x-www-form-urlencode mitgeschickt werden können. Die gesendeten Anfragen sollen ihr Feedback je nach Wunsch, via Content Negotiation, entweder in JSON oder in Extensible Markup Language (XML) zurücksenden. Dabei sollen drei Strategien bereitgestellt werden, entweder mittels Suffix, einem URL-Parameter oder dem HTTP Accept-Header.

4.2.1 Player Controller

Der Player Controller soll zwei Einstiegspunkt an den URIs `/player/` und `/player/{id}` zur Verfügung stellen. Der Parameter `{id}` dient dabei als Platzhalter für die ID eines Players, welche wiederum als Zahl dargestellt wird.

Am ersten Einstiegspunkt soll eine Liste aller Spieler über einen GET-Request bereitgestellt werden können. Des weiteren soll an diesem die Möglichkeit bestehen einen neuen Player mithilfe eines POST-Requests zu erzeugen. Dabei muss als Parameter der Name und das Passwort des Players mitgegeben werden. Die SQLite-Datenbank soll die ID dabei automatisch mittels Auto-increment erzeugen. Bei erfolgreicher Erstellung des Players soll dieser zurückgegeben werden, ansonsten `NULL`.

Am zweiten Einstiegspunkt soll ein einzelner existierender Player über einen GET-Request bereitgestellt, über einen DELETE-Request gelöscht und über einen PATCH-Request aktualisiert werden können. Nur das Passwort darf dabei laut Anforderungen[1] aktualisiert werden, wobei dieses zusätzlich als Parameter an den Request mit angehangen werden muss.

Für ein besseres Verständnis bietet die Abbildung 4.2 eine visuelle Verdeutlichung der Einstiegspunkte des Player Controllers.

1 siehe Kapitel 4.1

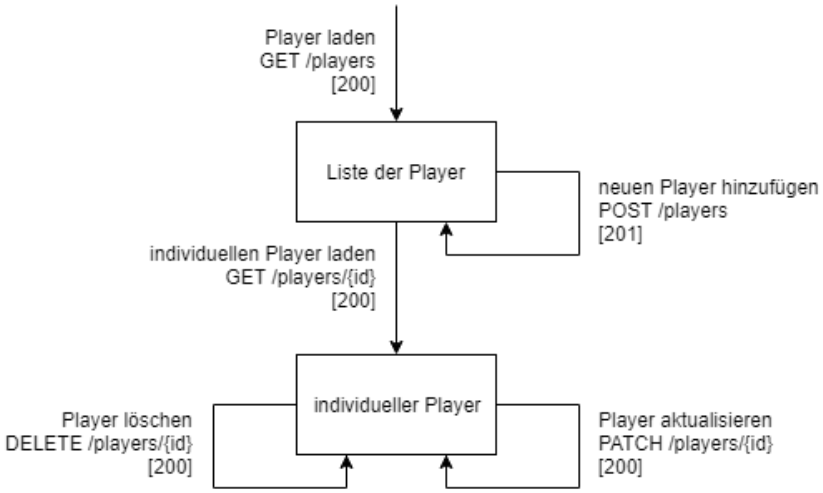

Abbildung 4.2: Player Controller - Übersicht der Einstiegspunkte

4.2.2 Match Controller

Die URIs `/match/`, `/match/{id}`, `/match/id/draws` und `/match/id/pieceSets` sollen durch den Match Controller bereitgestellt werden.

Dabei soll die erste URI ebenso wie beim „Player Controller“ Zurverfügungstellung einer Liste aller registrierten Matches und dem anlegen neuer dienen. Das Bereitstellen der Liste soll mittels GET- und das anlegen mittels POST-Request erfolgen. Der GET-Request soll zwei optionale boolesche Parameter bereitstellen, womit das Senden der Historie von Draws bzw. der Figurenstellung bestimmt werden soll. Standardmäßig sollen dabei die Parameter `TRUE` sein. Weiter unten in diesem Kapitel wird beschrieben wie diese Informationen separat geholt werden können. Um ein neues Match zu registrieren, müssen dabei die ID’s der beiden Spielteilnehmer mitgeschickt werden. Anhand des Parameternamens soll festgelegt werden welcher Spieler Weiß und welcher Schwarz spielen soll.

Der zweite Einstiegspunkt soll in diesem Controller ausschließlich dazu verwendet werden, um einzelne Matches mithilfe eines GET-Request anzufordern oder mithilfe eines DELETE-Request zu löschen. Für das anfordern eines einzelnen Matches stehen ebenso zwei boolesche, wie beim Anfordern einer Liste, bereit. Wenn ein Nutzer ein Match löscht, sollen ebenfalls alle zugehörigen Draws gelöscht werden. Um eine unrechtmäßige Manipulation der Match-Daten durch einen Nutzer zu verhindern, soll keine Möglichkeit bereitstehen ein Match zu aktualisieren. Die Aktualisierung eines Matches soll ausschließlich durch das hinzufügen von Draws erfolgen.[1]

1 siehe Kapitel 4.2.3

Die letzten beiden Einstiegspunkte sollen dazu dienen große Match bezogene Daten separat zu ermitteln. Dabei soll mittels GET-Request an der URI `/match/id/draw` eine Liste aller Draws und über die URI `/match/id/pieceSets` eine Map mit allen Figuren der beiden Spielteilnehmer bereitgestellt werden. Neben den aktuell auf dem Spielfeld stehenden Figuren sollen dabei auch die geschmissen mitgeschickt werden.

Die Abbildung 4.3 bietet für die zuvor definierten Einstiegspunkte eine grafische Veranschaulichung.

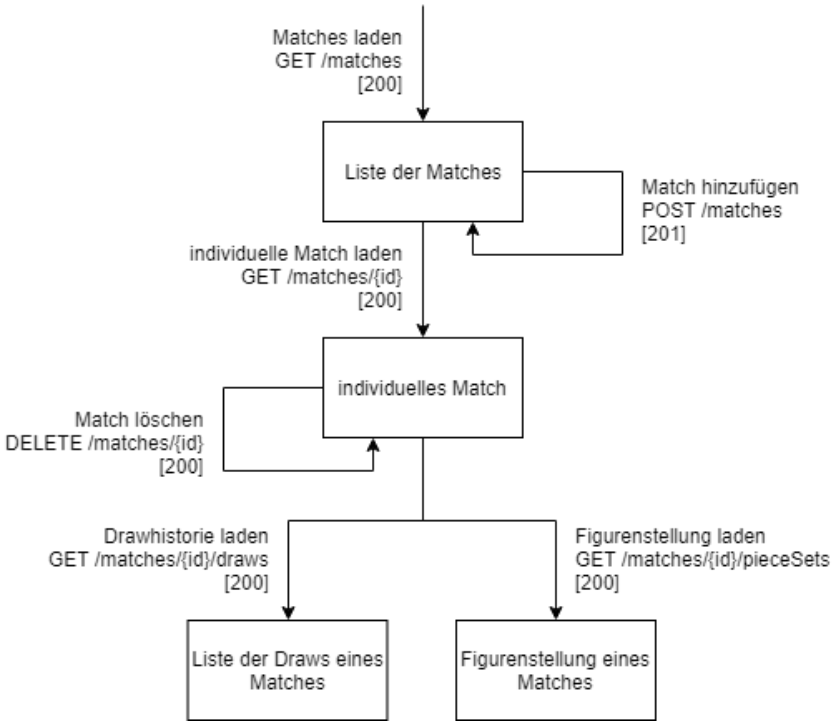

Abbildung 4.3: Match Controller - Übersicht der Einstiegspunkte

4.2.3 Draw Controller

Auch der Draw Controller stellt wieder zwei Einstiegspunkte an den URIs `/draw/` und `/draw/{id}` bereit.

Am ersten Einstiegspunkt soll wieder das bereitstellen einer Liste mittels GET- und das hinzufügen mittels POST-Request von Draws zur Verfügung gestellt werden. Für das hinzufügen eines neuen Draws ist die ID des Matches und der Draw-Code in der SAN als Parameter von nöten. Zusätzlich soll die Möglichkeit bestehen die Zeilen- und die Spaltennummer der Startposition mitzugeben. Wenn diese Informationen nicht mitgegeben werden, so soll der Controller die Startposition kalkulieren. Spalten sollen dabei als Nummern und nicht als Buchstaben mitgegeben

werden[1]. Nach erfolgreicher Validierung des Draw-Codes soll der Draw dem zugehörigen Match hinzugefügt und die Match-Daten aktualisiert werden.

Über den zweiten Einstiegspunkt soll wieder die Abfrage nach einem einzelnen Draw möglich sein. Eine Möglichkeit zum Löschen oder Aktualisieren des Draws soll nicht bestehen, da sonst der Nutzer wieder eine Möglichkeit hätte das Match zu manipulieren. Das Löschen von Draws soll über die Löschung des dazugehörigen Matches erfolgen[2].

Wie in den vorherigen Kapiteln bietet die Abbildung 4.4 ein Veranschaulichung.

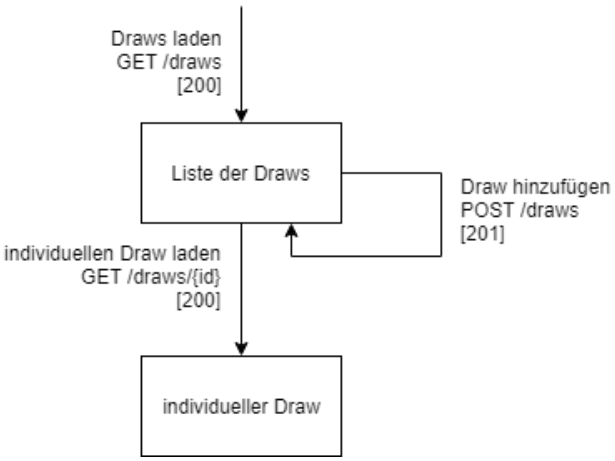

Abbildung 4.4: Draw Controller - Übersicht der Einstiegspunkte

1 A → 8; ...; H → 8
2 siehe Kapitel 4.2.2

KAPITEL 5

Konzept des Clients

In diesem Kapitel soll die Planung des Schachclients näher erläutert werden, welches als Grundbaustein für die Implementierung dienen soll. Dabei dient der erste Abschnitt zur Beschreibung der benötigten Anforderung, welche der Client erfüllen soll. Im zweiten Abschnitt sollen die einzelnen Ansichten, welche für eine bequeme Nutzerinteraktion benötigt werden, näher beschrieben.

5.1 Anforderungen

Grundlegen soll der Client als Visualisierung des Servers dienen. Dafür soll dieser eine Verwaltung von Playern und Matches, inklusive dem anlegen neuer und dem bearbeiten, bereitstellen. Um anschließend auch Schach spielen zu können muss der Client dafür eine komfortable Möglichkeit, in Form eines Schachbrettes, bieten.

Natürlich muss der Client außerdem mit dem Server kommunizieren können. Dafür muss dieser Requests senden und die empfangenen Response-Nachrichten verarbeiten können. Da der Server für manche Request spezielle Parameter benötigt, wie zum Beispiel einen String in der SAN, muss der Client auch diese ermitteln können.

Des weiteren soll der Client als Single Page Application (SPA) erstellt werden und muss daher eine Möglichkeit zum Austauschen der einzelnen Ansichten, welche in dem Kapitel 5.2 genauer definiert werden, bieten.

Als letzte Grundanforderung soll eine innovative und benutzerfreundliche Bedienung der Anwendung sein, so das bis auf die Schachregeln keine weiteren Grundvoraussetzungen benötigt werden.

5.2 Mock-Up-Entwicklung der benötigten Client-Ansichten

In diesem Abschnitt sollen die im Kapitel 5.1 zuvor definierten Anforderungen konkretisiert und visuell aufbereitet werden. Die in diesem Kapitel erstellten Mock-Ups sollen die Implementierung

vereinfachen bzw. beschleunigen.

5.2.1 Startansicht

Diese Ansicht soll als Einstiegspunkte für Nutzer dienen, welche über diese die Möglichkeit bekommen sollen zur Player-Ansicht bzw. zur Match-Ansicht zu wechseln. Die Abbildung 5.1 visualisiert dabei die zuvor definierten Anforderungen.

Abbildung 5.1: Mock-Up: Startansicht des Clients

5.2.2 Player-Ansicht

Inhalt dieser Ansicht soll die Möglichkeit zur Verwaltung von Playern sein. Dafür soll eine Tabelle mit allen angelegten Playern und ein Formular zum anlegen bzw. bearbeiten bereitstehen. Visuell unterstreicht die Abbildung 5.2 die definierten Anforderungen.

5.2.3 Match-Ansicht

Mithilfe der Match-Ansicht soll die Verwaltung der Matches möglich sein. Hierfür soll wie bei der Player-Ansicht eine Tabelle mit angelegten Matches und ein Formular zum anlegen bereitstehen. Durch die Abbildung 5.3 werden die Anforderung grafisch dargestellt.

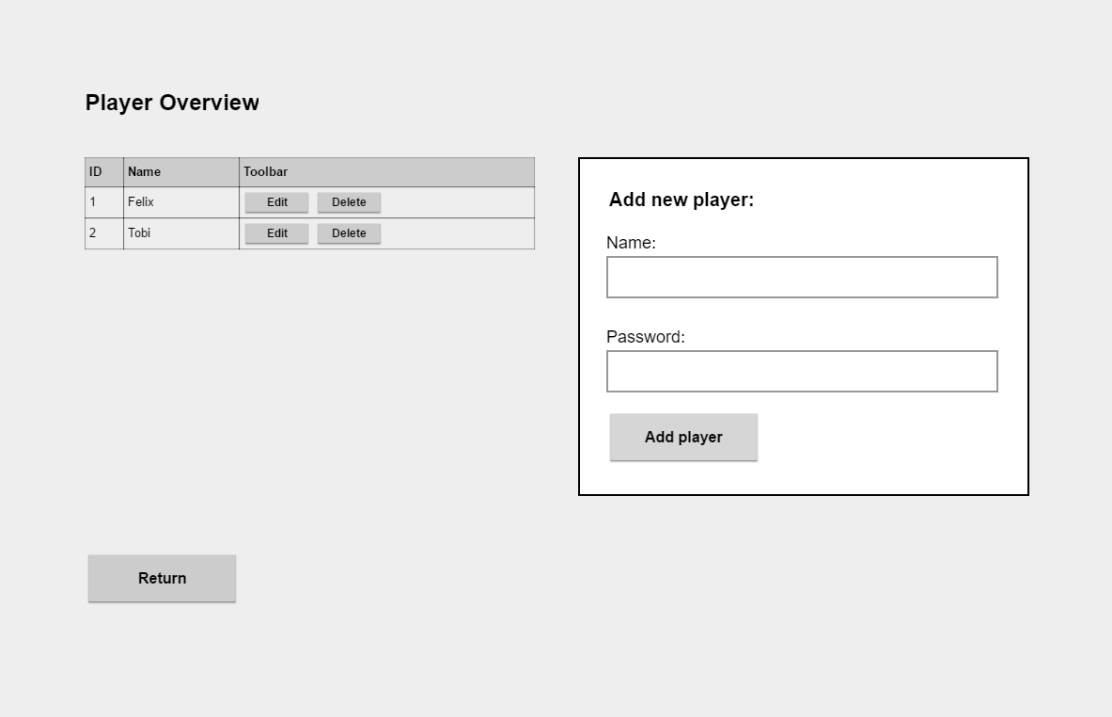

Abbildung 5.2: Mock-Up: Player-Ansicht des Clients

5.2.4 Ansicht eines gestarteten Matches

Durch diese Ansicht soll die Möglichkeit zum Schach spielen bereitgestellt werden. Um spielen zu können wird in erster Linie ein Schachbrett benötigt, auf welchem die Figuren dargestellt werden. Mittels „Drag & Drop“ sollen dabei Spielfiguren bewegt und mittels „Mouseover“ sollen möglichen Züge anzeigt werden können. Neben dem Schachbrett soll eine Reihe von Informationen bereitgestellt werden. Inhalt dieser sollen die geschmissenen Figuren, eine Liste aller Züge und ob sich ein Player im Schach befindet sein. Die Abbildung 5.4 zeigt die visuelle Darstellung der Anforderungen.

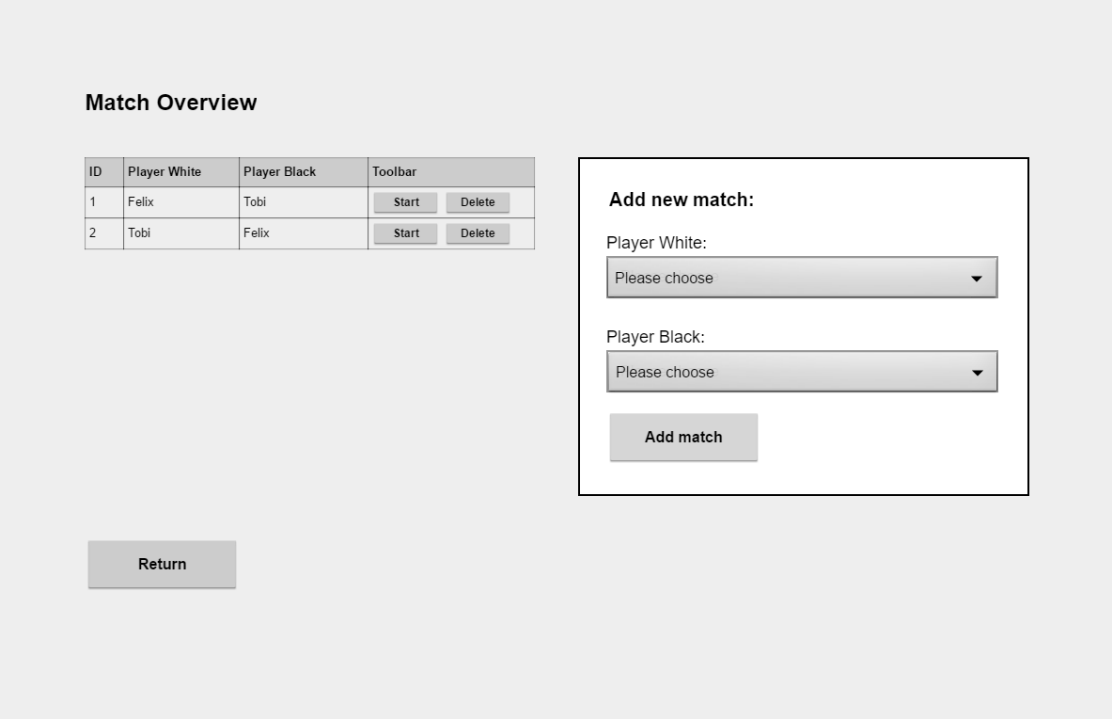

Abbildung 5.3: Mock-Up: Match-Ansicht des Clients

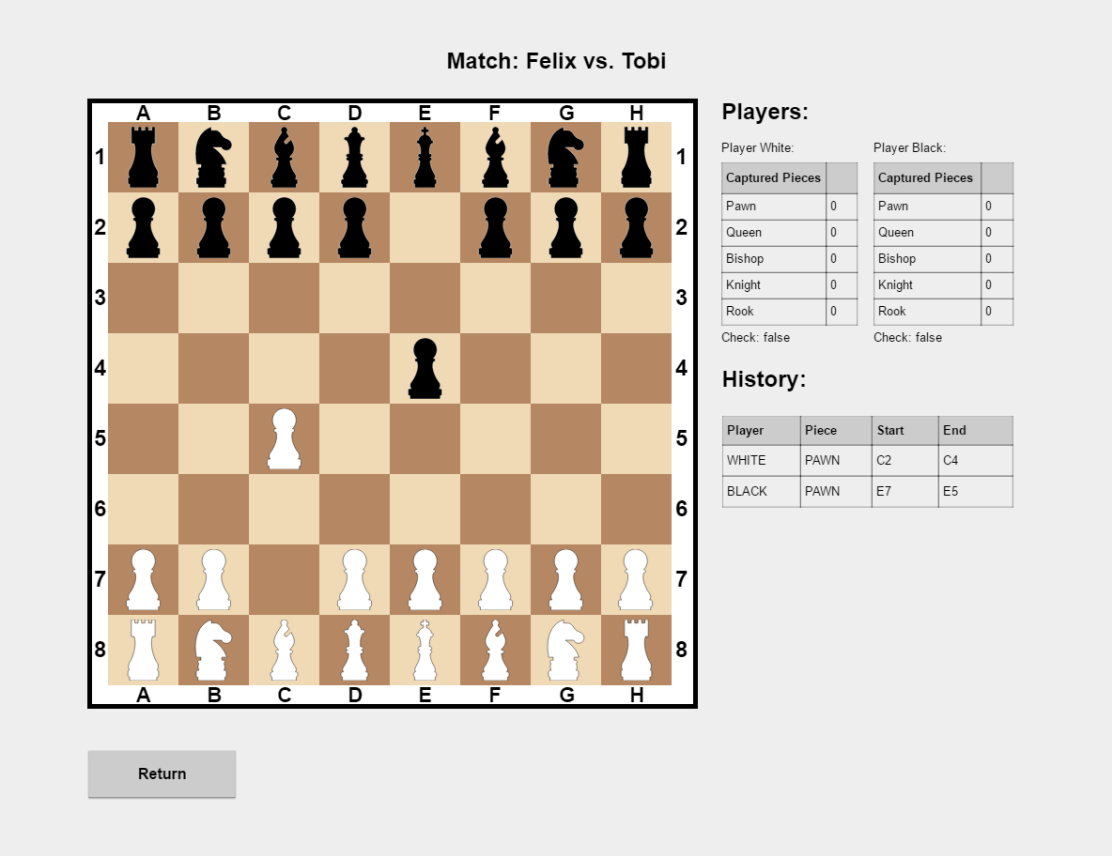

Abbildung 5.4: Mock-Up: Ansicht eines gestarteten Matches

KAPITEL 6

Implementation des Servers

Inhalt dieses Kapitels soll die Umsetzung des Konzeptes aus Kapitel 4 sein. Dafür wird im ersten Abschnitt auf die Bibliotheken bzw. Frameworks eingegangen, welche für die Umsetzung verwendet wurden. Der zweite Teil wird die Anbindung zur Datenbank näher betrachtet. Anschließend folgt eine Erläuterung zur Konfiguration von Content-Negotiation einer Spring-Applikation. In dem vierten Unterkapitel wird die Fehlerbehandlung bzw. das Exceptionhandling näher erläutert. Der letzte Abschnitt befasst sich mit der Ermittlung bzw. Analyse der Schachnotationen FEN und SAN.

6.1 Verwendete Bibliotheken/Frameworks

Die verwendeten Bibliotheken bzw. Frameworks sollen die Umsetzung der Anforderungen aus dem Kapitel 4.1 vereinfachen und beschleunigen. Dabei werden diese in den nachfolgenden Unterabschnitten 6.1.1 bis 6.1.4 in den Punkten Zweck, Einrichtung und Verwendung näher betrachtet.

6.1.1 Spring

Spring wird als ein leichtgewichtiges Framework für die Umsetzung von Java Applikationen beschrieben. Dabei bezieht sich das leichtgewichtig nicht auf die Größe oder Anzahl der enthaltenen Klassen. Es ist eher als geringer Aufwand an Änderungen am eigenen Programmcode zu verstehen, um die Vorteile des Frameworks nutzen zu können. [Cos17] Spring biete eine Vielzahl an Einsatzmöglichkeiten, aber für die Umsetzung des Servers soll es für die Erzeugung der REST-API dienen.

Um Spring in einem Projekt einzubinden, müssen die Zeilen aus dem Listing 6.1, zusätzlich zu denen welche Kotlin benötigt, in der Build-Datei „build.gradle“ eingefügt werden. Zu beachten ist dabei das Gradle nur eine Lösung für die Einbindung ist. Da aber für die Umsetzung ebenfalls Gradle verwendet wird, werden alle anderen Lösungen an dieser Stelle vernachlässigt.

ggf. irge
wo besc
ben wie
Kotlin i
Gradle e
bindet

Für die Erstellung einer REST API muss zunächst ein Controller für eine Ressource angelegt

```
buildscript {
  repositories {
    mavenCentral()
  }
  dependencies {
    classpath "org.jetbrains.kotlin:kotlin-allopen:1.2.30"
    classpath "org.springframework.boot:spring-boot-gradle-plugin:1.5.4.RELEASE"
  }
}
apply plugin: "kotlin-spring"
apply plugin: "org.springframework.boot"

repositories {
  mavenCentral()
}

dependencies {
  compile "org.springframework.boot:spring-boot-starter-web"
}
```

Listing 6.1: Einbindung des Spring Framework mithilfe von Gradle

werden, dabei ist es sinnvoll für jede einen eigenen Controller zu definieren. Innerhalb werden anschließend alle Einstiegspunkte erzeugt. Als Beispiel soll das Listing 6.2 in Form eines klassischen `Hello World` dienen. Dafür wird eine Klasse `GreetingController` mit einer Funktion `getGreeting` definiert. Als Parameter bekommt diese Funktion einen Namen übergeben, welcher standardmäßig den String `World` hält. An dieser Stelle kommt das Spring Framework ins Spiel. Dieses stellt eine Reihe von Annotation bereit, wobei `@RestController` einen Controller für die API, `@GetMapping` einen Einstiegspunkt mit der Request-Methode `GET` und `@RequestParam` einen Parameter für diese Funktion definiert. Abschließend muss nur noch der Startpunkt für die

```
import org.springframework.web.bind.annotation.*

@RestController
class GreetingController {
  @GetMapping("/greeting")
  fun getGreeting(@RequestParam name: String = "World"): String {
    return "Hello $name!"
  }
}
```

Listing 6.2: Beispiel: Spring Controller

Applikation definiert werden. Dafür wird wieder mithilfe einer Annotation ein Klasse erzeugt, welche aber keinerlei Informationen benötigt. Anschließend muss diese in der Main-Funktion, wie im Listing 6.3 zu sehen, gerufen werden. Für eine detaillierte Beschreibung dieses Beispiels stehen

```
@SpringBootApplication
class Application

fun main(args: Array<String>) {
  SpringApplication.run(Application::class.java, *args)
}
```

Listing 6.3: Beispiel: Spring Application Class

auf den Internetseiten [Har] und [Inc] weitere Informationen zur Verfügung.

6.1.2 SQLite

SQLite ist eine OpenSource Bibliothek, welche ein dateibasiertes relationales Datenbanksystem bereitstellt. Der größte Unterschied zu anderen relationalen SQL-Datenbank besteht darin, das SQLite keinen separaten Serverprozess besitzt und somit als eingebettete Datenbank-Engine betrachtet werden kann. Alle Tabellen, Indizes, Trigger und Sichten einer Datenbank werden dabei in einem plattformunabhängigen Format in einer einzigen Datei gespeichert. Das bedeutet, dass Datenbankdateien bequem zwischen 32-Bit und 64-Bit-Systemen oder Little-Endian- und Big-Endian-Architekturen getauscht werden können.[Hip]

Für die Einbindung von SQLite in ein Projekt, mithilfe der Datenbankschnittstelle Java Database Connectivity (JDBC), sind folgende Zeile in der Build-Datei von Gradle vonnöten: Für ein

```
1 repositories {
2   mavenCentral()
3 }
4
5 dependencies {
6   compile group: 'org.xerial', name: 'sqlite-jdbc', version: '3.21.0.1'
7 }
```

Listing 6.4: Einbindung der Bibliothek SQLite mithilfe von Gradle

einfaches Beispiel für die Verbindung zu einer SQLite Datenbank, die Erstellung von Tabellen und für das Absenden von SQL-Abfragen stellt [Lim] ein Tutorial bereit. Da für die Implementierung eine Object-relation mapping (ORM) Bibliothek, welche im Kapitel 6.1.3 verwendet wird, wird eine nähere Betrachtung für die Verwendung von SQLite mithilfe des JDBC Treibers vernachlässigt.

6.1.3 ORMLite

ORMLite ist ein OpenSource ORM Projekt von Gray Watson, welches für Java entwickelt wurde, aber in Kotlin durch die Möglichkeit der Interoperabilität mit Java ebenfalls verwendet werden kann. Die Bibliothek unterstützt dabei eine Reihe von Datenbank-Systemen, wie zum Beispiel MySQL, Postgres oder SQLite.

Um ORMLite in ein Gradle Projekt einzubinden müssen die Zeilen aus dem Listing 6.5 in die Build-Datei eingetragen werden. Dabei muss neben der Core-Bibliothek die JDBC-Bibliothek von ORMLite eingebunden werden, welches für die Verbindung zur Datenbank zuständig ist. Da aber JDBC mit mehreren Datenbank-Systemen kommunizieren kann muss noch, wie in dem Kapitel 6.1.2 für SQLite erläutert, der Datenbank-Treiber für das verwendete Datenbank-System eingebunden werden. Für die Persistierung zeigt das Listing 6.6 wie einzelne Klassen durch die

```
repositories {
  mavenCentral()
}

dependencies {
  compile "com.j256.ormlite:ormlite-core:5.1"
  compile "com.j256.ormlite:ormlite-jdbc:5.1"
}
```

Listing 6.5: Einbindung der Bibliothek ORMLite mithilfe von Gradle

von ORMLite bereitgestellten Annotationen verwendet werden können. Der Zugriff auf die Da-

```
@DatabaseTable(tableName = "accounts")
public class Account {
  @DatabaseField(id = true)
  private String name;

  @DatabaseField(canBeNull = false)
  private String password;
  ...
  Account() {
    // all persisted classes must define a no-arg constructor with at least package visibility
  }
  ...
}
```

Listing 6.6: Beispiel: Persistierung einer Klasse mittels ORMLite [Wat]

tenbank erfolgt mittels Database Access Object (DAO) Klassen, welche für jede Tabelle erzeugt werden müssen. Mit diesen DAOs können anschließend Datensätze erstellt, bearbeitet und gelöscht werden. Zu dem stellen diese eine Reihe von Funktionen bereit um Datensätze abzufragen, wie zum Beispiel die Abfrage nach alle Datensätzen in der Datenbank oder nach einem bestimmten Objekt anhand seiner ID. Wenn diese Standard Funktionen nicht ausreichen besteht des weiteren die Möglichkeit komplexe Abfragen zu generieren mithilfe von einem sogenannten Query-Builder. Zur Veranschaulich der Verwendung von ORMLite zeigt das Listing 6.7 ein Minmalbeispiel.

```
String databaseUrl = "jdbc:sqlite:path/to/account.db";
ConnectionSource connectionSource = new JdbcConnectionSource(databaseUrl);

Dao<Account,String> accountDao = DaoManager.createDao(connectionSource, Account.class);

TableUtils.createTable(connectionSource, Account.class);

String name = "Jim Smith";
Account account = new Account(name, "_secret");
accountDao.create(account);

Account account2 = accountDao.queryForId(name);
System.out.println("Account: " + account2.getPassword());

connectionSource.close();
```

Listing 6.7: Beispiel: Verwendung von ORMLite (verändert nach [Wat])

6.1.4 Fasterxml

Mithilfe von Fasterxml werden inkrementelle Parser- und Generator-Abstraktionen bereitgestellt, welche in der Standardimplementierung die JSON verarbeiten können. Dabei bietet das OpenSource-Projekt noch weitere Unterstützung für andere Datenformate, wie beispielsweise XML oder Comma-separated Values (CSV), an.[18]

Für die Einbindung in ein Gradle-Projekt müssen die Zeilen aus dem Listing 6.8 in die Build-Datei hinzugefügt werden. Um anschließend den XML-Support für ein Spring-Projekt einzurichten,

```
repositories {
  mavenCentral()
}

dependencies {
  compile group: 'com.fasterxml.jackson.core', name: 'jackson-core', version: "$fasterxml_jackson_version"
  compile group: 'com.fasterxml.jackson.core', name: 'jackson-annotations', version: "
$fasterxml_jackson_version"
  compile group: 'com.fasterxml.jackson.core', name: 'jackson-databind', version: "
$fasterxml_jackson_version"
}
```

Listing 6.8: Einbindung der Bibliothek Fasterxml mithilfe von Gradle

müssen die Modelle mit Annotations erweitert werden. Das Listing 6.9 zeigt dabei wie ein Modell zu konfigurieren ist. Mit diesen Einstellungen wird die Content-Negotiation gewährleistet, welche im Kapitel 6.3 näher beleuchtet wird.

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
data class Player(
  @XmlElement
  val id: Int = 0,
  @XmlElement
  var name: String = "",
  @XmlElement
  var password: String = ""
)
```

Listing 6.9: Beispiel: Verwendung von Fasterxml

6.2 Anbindung an die Datenbank

Für die Anbindung ist zu aller erst die Vorbereitung der Models, wie im Kapitel 6.1.3 erläutert, vonnöten. Als nächstes muss vor einer Interaktion eine Verbindung zur Datenbank aufgebaut, alle benötigten Tabellen und DAOs angelegt werden. Das Listing 6.10 zeigt dabei die Umsetzung dieser Notwendigkeiten. Dabei wird erst eine Verbindung aufgebaut sobald diese benötigt wird und zwar dann wenn eine DAO geholt wird. Wenn ein DAO angefordert wird, so wird die Verbindung ausschließlich aufgebaut sofern noch keine besteht. Ähnlich sieht das beim anlegen der Tabellen aus, denn diese werden ausschließlich nur dann angelegt sofern diese nicht existieren. Dieser Fall

```
class DatabaseUtility {
  companion object {
    private var connection: JdbcConnectionSource? = null
    var playerDao: Dao<Player, Int>? = null
      get() {
        if (field == null) connect()
        return field
      }
    ...
    private fun connect() {
      if (connection != null) return
      connection = JdbcConnectionSource("jdbc:sqlite:chessgame.db")
      createTables()
      createDaos()
    }

    private fun createTables() {
      TableUtils.createTableIfNotExists(connection, Player::class.java)
      ...
    }

    private fun createDaos() {
      playerDao = DaoManager.createDao<Dao<Player, Int>, Player>(connection, Player::class.java)
      ...
    }
  }
}
```

Listing 6.10: Verbindungsaufbau & Initialisierung der SQLite Datenbank

tritt beispielsweise nach einem Neustart des Servers auf.

6.3 Spring-Konfiguration für Content-Negotiation

Content-Negotiation ist laut Kapitel **??** ein wichtiges Qualitätsmerkmal für eine REST-API. Das Framework Spring[1] bietet auch hierfür Lösungen an, welche aber nicht standardmäßig zur Verfügung stehen. Die JSON wird dabei ohne weitere Konfiguration unterstützt, nur für XML müssen die Modelle mithilfe der Bibliothek Fasterxml[2] angepasst werden. Anschließend muss die Spring-Konfiguration, wie in Listing 6.11 zu sehen, erweitert werden. Die Implementierung zeigt dabei die Umsetzung der drei Strategien, welche in dem Kapitel 4.1 gefordert wurden. Die Internetseite [SRL16] bietet für dieses Thema eine ausführlichere Erläuterung der einzelnen

```
@Configuration
class WebConfig : WebMvcConfigurerAdapter() {
  override fun configureContentNegotiation(configurer: ContentNegotiationConfigurer?) {
    configurer!!
      .favorPathExtension(true)
      .favorParameter(true)
      .parameterName("mediaType")
      .ignoreAcceptHeader(false)
      .useJaf(false)
      .defaultContentType(MediaType.APPLICATION_JSON)
      .mediaType("xml", MediaType.APPLICATION_XML)
      .mediaType("json", MediaType.APPLICATION_JSON)
  }
}
```

Listing 6.11: Spring-Konfiguration der drei Content-Negotiation-Strategien

Strategien, wie diese angewendet und eingerichtet werden.

6.4 Exceptionhandling

Laut den Anforderungen aus dem Kapitel 4.1 soll der Server so wenig wie möglich anfällig für Fehler sein. Dafür ist der richtige Umgang und auch eine verständliche, für den Endnutzer lesbare, Fehlermeldung unerlässlich. Die Fehler können dabei in einzelnen Fehlerarten unterteilt werden.

Wenn ein Nutzer eine Ressource an der URI `/player/15` anfordert, aber kein Player mit der ID 15 existiert, dann wird ein Fehler mit dem HTTP-Statuscode 400 zurückgegeben. Schon der Statuscode alleine signalisiert dem Nutzer das die angeforderte Ressource nicht gefunden wurde. Sollte ein Benutzer der API einen Draw hinzufügen möchte aber der mitgeschickte Draw-Code in der SAN nicht valide ist, so wird ein Fehler mit dem HTTP-Statuscode 409 zurückgegeben. Dieser zeigt dem Nutzer das ein Konflikt, welchen er verursacht hat, aufgetreten ist. Für alle weiteren Fehler welche nicht durch den Nutzer verursacht wurden, wird dieser mit dem HTTP-Statuscode 500 zurückgeben. Da so ein Fehler nur zurückgegeben wird, sofern ein unerwarteter Server-Fehler

1 siehe Kapitel 6.1.1
2 siehe Kapitel 6.1.4

aufgetreten ist, sollte dieser gar nicht bzw. nur selten auftreten. Ein Beispiel könnte ein Verbindungsabbruch zur Datenbank sein, worauf der eigentliche Nutzer keinen Einfluss hat. Das Listing 6.12 zeigt dabei die Konfiguration der Spring-Applikation, für den Umgang mit Fehlern.

Damit der API-Benutzer nicht nur einen Statuscode zum aufgetretenen Fehler erhält, wird zu-

```
@RestControllerAdvice
class ExceptionHandler {
  @ExceptionHandler(value = [IllegalArgumentException::class])
  @ResponseStatus(NOT_FOUND)
  fun handleIllegalArgumentException(ex: Exception, request: WebRequest): ErrorResponseObject {
    return generateErrorResponseObject(ex, request, NOT_FOUND)
  }

  @ExceptionHandler(value = [RuntimeException::class])
  @ResponseStatus(CONFLICT)
  fun handleRuntimeException(ex: Exception, request: WebRequest): ErrorResponseObject {
    return generateErrorResponseObject(ex, request, CONFLICT)
  }

  @ExceptionHandler(value = [Exception::class])
  @ResponseStatus(INTERNAL_SERVER_ERROR)
  fun handleUnknownException(ex: Exception, request: WebRequest): ErrorResponseObject {
    return generateErrorResponseObject(ex, request, INTERNAL_SERVER_ERROR)
  }

  private fun generateErrorResponseObject(ex: Exception, request: WebRequest, statusCode: HttpStatus):
ErrorResponseObject {
    return ErrorResponseObject(...)
  }
}
```

Listing 6.12: Spring-Konfiguration des Exceptionhandling

sätzlich im Nachrichtenrumpf der Antwort ein Fehlerobjekt, in Form der Klasse `ErrorResponseObject`[1], mitgeschickt. Dieses hält einen Zeitstempel, den Statuscode, die Bezeichnung des Statuscodes, welche Exception des Fehler verursacht hat, eine detaillierte Fehlermeldung und den Pfad an welchen der Fehler aufgetreten ist. Das Fehlerobjekt wird dabei in dem angeforderten Format zurückgegeben[2].

1 siehe Listing 6.13
2 siehe Kapitel 6.3

```
class ErrorResponseObject(
  val timestamp: Date = Date(),
  val statusCode: Int = 500,
  val error: HttpStatus = HttpStatus.INTERNAL_SERVER_ERROR,
  val exception: String = "",
  val message: String = "No message available",
  val path: String = ""
) {
override fun toString(): String {
  return "ErrorResponseObject{" +
      "timestamp=" + timestamp +
      ", status=" + statusCode +
      ", error=" + error +
      ", exception=" + exception +
      ", message=" + message +
      ", path=" + path +
      '}'.toString()
  }
}
```

Listing 6.13: Spring-Konfiguration des Exceptionhandling

6.5 Analyse/Ermittlung der FEN bzw. SAN

Nach den Anforderungen aus dem Kapitel 4.1 wurde definiert, das Änderungen an einem Match ausschließlich über das hinzufügen eines Draws erfolgen sollen. Trotzdem muss eine Validierung des übermittelten Draw-Code durchgeführt werden, um zu überprüfen das dieser auch möglich ist. Dafür wurde der regulärer Ausdruck aus der Abbildung 6.1 entwickelt, welcher in acht Teile bzw. Gruppen unterteilt werden kann.

Der erste Part ermittelt dabei die Art der Spielfigur. Anhand des Fragezeichen-Symbols ist

$$\underbrace{([KQBNR])?}_{1}\underbrace{([a-h]|[1-8])?}_{2}\underbrace{(x)?}_{3}\underbrace{([a-h])}_{4}\underbrace{([1-8])}_{5}\underbrace{([QBRN])?}_{6}\underbrace{(e\backslash.p\backslash.)?}_{7}\underbrace{(\backslash+\{1,2\}|\#)?}_{8}$$

Abbildung 6.1: Regulärer Ausdruck zur Validierung eines Strings in der SAN

zu sehen das dieser Teil optional ist, was dran liegt das kein Buchstabe angegeben wird wenn eine Bauernfigur gezogen wurde. Der zweite Teil zeigt an von welcher Spalten oder Reihe die Figur gestartet ist. Diese Information is notwendig wenn zwei Figuren der selben Art auf das Zielfeld gelangen können. Part drei gibt an ob eine Figur geschmissen wurde. In den Abschnitten vier und fünf wird das Zielfeld ermitteln, wobei der vierte Part die Spalte und der fünfte die Reihe darstellt. Teil sechs gibt die Art der Spielfigur an in welche sich ein Bauer umwandelt, wenn er die hinterste Reihe erreicht hat. Der vorletzte Part zeigt qn ob ein „Schlagen im Vorbeigehen“ bzw. ein „Schlagen en passant“ durchgeführt wurde. Im letzten Abschnitt ermittelt ob der Draw zu einem Schach oder einem Schachmatt führt. Dabei werden zwei Schreibweise für ein Schachmatt akzeptiert, einmal „++“ und zum anderen „#“. Zu beachten ist das mithilfe des regulären Ausdrucks nicht ermittelt werden kann, ob eine Rochade durchgeführt wurde. Dies muss zuvor separat geprüft werden.

Mithilfe des regulären Ausdrucks kann anschließend anhand der Figurenstellung, welche im Match gespeichert sind, überprüft werden ob der Draw valide ist. Dafür wird als erstes das Startfelder ermittelt, sofern dieses nicht als Request-Parameter mit übergeben wurde. Wichtig hierbei ist das immer nur ein einziges Startfeld ermittelt werden darf, hierfür müssen ggf. die Informationen aus dem zweiten Teil des regulären Ausdrucks heran gezogen werden. Anschließend werden alle möglichen Felder ermittelt, zu welchen sich die Figur bewegen kann. Ist das übermittelte Zielfeld nicht enthalten, dann ist der Draw invalide. Wenn hingegen das Zielfeld enthalten ist, dann wird zusätzlich überprüft ob eine Figur geschlagen wird. Das gleiche gilt für Schlagen „en passant“, Schach und Schachmatt. Ist einer dieser Werte nicht oder falsch angegeben, dann ist der Draw invalide.

Ist der Draw valide wird dieser schließlich in der Datenbank gespeichert und das dazugehörige Match wird angepasst. Dafür wird die Position der bewegten Figur, ggf. die geschlagene Figur, Informationen zu Schach oder Schachmatt, die Möglichkeit zur Rochade und die Halbzüge aktualisiert. Anhand der neuen Informationen kann anschließend der neue Wert der FEN ermittelt werden. Nach erfolgreicher Aktualisierung wird das Match in der Datenbank angepasst.

KAPITEL 7

Implementation des Clients

7.1 Verwendete Bibliotheken/Frameworks

7.1.1 RequireJS

RequireJS ist ein für JavaScript entwickelte Bibliothek zum laden von Modulen. Dabei ist es für Nutzung innerhalb des Browsers optimiert, kann aber auch für andere Umgebungen genutzt werden. Ziel von solchen Bibliotheken wie RequireJs soll sein den eigenen Code zu beschleunigen und die Qualität zu steigern.[ANOa]

Für die Modulbeschreibung innerhalb von JavaScript stehen mehrere Formate bereit. RequireJS setzt dabei auf das Format Asynchronous Module Definition (AMD). Das Listing 7.1 stellt ein einfaches Beispiel für die Definition eines AMD-Moduls bereit. In dem Zeitschriftartikel [Bra17] können zu den einzelnen Modul-Formaten und deren Einsatzmöglichkeiten nähere Informationen nachgelesen werden.

```
1 define(['jquery'] , function ($) {
2   return function () {};
3 });
```

Listing 7.1: Beispiel: Moduldefinition mittels Asynchronous Module Definition (AMD) [ANOb]

7.1.2 kotlinx.html

Die Kotlinx.html ist eine offiziell von JetBrains entwickelte Bibliothek, welche eine DSL für das Erstellen bzw. Ergänzen von HTML-Code bereitstellt. Sie kann dabei für die Java Virtual Machine (JVM) oder JavaScript Plattform verwendet werden.

Diese Bibliothek ermöglicht es den sämtlichen HTML-Code in Kotlin-Code auszulagern. Das bringt einen großen Vorteil mit, denn das Erstellen des Codes wir durch eine statisch Typisierung abgesichert. Dadurch können mögliche Fehler im HTML-Code bereits während der Übersetzung des Quellcodes erkannt werden. Vergessene oder gar falsch verschachtelte HTML-Tags werden

dadurch vermieden. Ein Beispiel für die Verwendung stellt das Listing 7.2 dar, welches den HTML-Code aus Listing 7.3 generiert.

```
import kotlinx.html.*
import kotlinx.html.dom.*

val myDiv = document.create.div {
  p { +"text inside" }
}
```

Listing 7.2: Beispiel: Verwendung der Bibliothek kontlinx.html [**kotlinxExample**]

```
<div>
  <p>
    text inside
  </p>
</div>
```

Listing 7.3: Beispiel: Verwendung der Bibliothek kotlinx.html (Ergebnis)

7.1.3 kotlinx.serialization

7.1.4 kotlinx.coroutines

KAPITEL 8

Fazit

KAPITEL 9

Ausblick

Literatur

[ANOa] ANONYM: *Require.js. A JavaScript Module Loader*. URL: http://requirejs.org/ (besucht am 13. 04. 2018) (siehe S. 39).

[ANOb] ANONYM: *Why AMD?* URL: http://requirejs.org/docs/whyamd.html (besucht am 13. 04. 2018) (siehe S. 39).

[ARD17a] ARD/ZDF-MEDIENKOMMISSION: „ARD/ZDF-Onlinestudie 2017: Neun von zehn Deutschen sind online. Bewegtbild insgesamt stagniert, während Streamingdienste zunehmen - im Vergleich zu klassischem Fernsehen jedoch eine geringe Rolle spielen." *Media Perspektiven* (Sep. 2017), Bd. URL: http://www.ard-zdf-onlinestudie.de/ardzdf-onlinestudie-2017/ (besucht am 22. 03. 2018) (siehe S. 1).

[ARD17b] ARD/ZDF-MEDIENKOMMISSION: „Kern-Ergebnisse der ARD/ZDF-Onlinestudie 2017". *Media Perspektiven* (Sep. 2017), Bd. URL: http://www.ard-zdf-onlinestudie.de/files/2017/Artikel/Kern-Ergebnisse_ARDZDF-Onlinestudie_2017.pdf (besucht am 26. 03. 2018) (siehe S. 1).

[Bra17] BRAUN, HERBERT: „Modul.js. Formate und Werkzeuge für JavaScript-Module". *c't Heft 3/2017* (2017), Bd.: S. 128–133 (siehe S. 39).

[Cos17] COSMINA, JULIANA, ROB HARROP, CHRIS SCHAEFER und CLARENCE HO: *Pro Spring 5. An In-Depth Guide to the Spring Framework and Its Tools*. English. 5th. Apress, 11. Nov. 2017 (siehe S. 29).

[18] *Fasterxml*. 27. März 2018. URL: https://github.com/FasterXML/jackson-core (besucht am 19. 05. 2018) (siehe S. 33).

[Fie00] FIELDING, ROY THOMAS: „Architectural Styles and the Design of Network-based Software Architectures". phd. University of California, Irvine, 2000. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (besucht am 07. 05. 2018) (siehe S. 3).

[Fie08] FIELDING, ROY THOMAS: *REST APIs must be hypertext-driven*. 20. Okt. 2008. URL: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven (besucht am 14. 05. 2018) (siehe S. 3).

[Fie] FIELDING, ROY THOMAS u. a.: *Hypertext Transfer Protocol – HTTP/1.1*. URL: https://tools.ietf.org/html/rfc2616 (besucht am 12. 05. 2018) (siehe S. 8).

[Har] HARIRI, HADI, EDOARDO VACCHI und SÉBASTIEN DELEUZE: *Creating a RESTful Web Service with Spring Boot*. URL: https://kotlinlang.org/docs/tutorials/spring-boot-restful.html (besucht am 04. 04. 2018) (siehe S. 31).

[Hip] HIPP, WYRICK & COMPANY INC.: *About SQLite*. URL: https://www.sqlite.org/about.html (besucht am 09. 04. 2018) (siehe S. 31).

[Inc] INC., PIVOTAL SOFTWARE: *Building a RESTful Web Service*. URL: https://spring.io/guides/gs/rest-service/ (besucht am 04. 04. 2018) (siehe S. 31).

[Inc17] INC., STACK EXCHANGE: *Developer Survey 2017*. 2017. URL: https://insights.stackoverflow.com/survey/2017#technology (besucht am 26. 03. 2018) (siehe S. 1).

[Kre15] KRETZSCHMAR, CHRISTOPH: „Demonstration eines RESTful Webservices am Beispiel eines Schachservers“. Bachelor. Hochschule für Technik und Wirtschaft Dresden, 2015 (siehe S. 12, 13).

[Lim] LIMITED, TUTORIALS POINT INDIA PRIVATE: *SQLite - Java*. URL: https://www.tutorialspoint.com/sqlite/sqlite_java.htm (besucht am 09. 04. 2018) (siehe S. 31).

[Los08] LOSSA, GÜNTER: *Schach lernen. Ein Leitfaden für Anfänger des königlichen Spiels; Der entscheidene Zug zum zwingenden Mattangriff*. German. Joachim Beyer Verlag, 2008 (siehe S. 13).

[Mas] MASHKOV, SERGEY: *DOM trees*. URL: https://github.com/Kotlin/kotlinx.html/wiki/DOM-trees (besucht am 13. 04. 2018).

[Sha17] SHAFIROV, MAXIM: *Kotlin on Android. Now official*. Mai 2017. URL: https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/ (besucht am 26. 03. 2018) (siehe S. 1).

[Spi16] SPICHALE, KAI: *API-Design. Praxishandbuch für Java- und Webservice-Entwickler*. German. 1st. dpunkt.verlag GmbH, Dez. 2016 (siehe S. 3–6, 8).

[SRL16] SRL., BAELDUNG: *Spring MVC Content Negotiation*. 20. Aug. 2016. URL: http://www.baeldung.com/spring-mvc-content-negotiation-json-xml (siehe S. 35).

[Var15] VARANASI, BALAJI u. a.: *Introducing Gradle*. English. 1st. Apress, 23. Dez. 2015 (siehe S. 9).

[Wat] WATSON, GRAY: *OrmLite - Lightweight Object Relational Mapping (ORM) Java Package*. URL: http://ormlite.com/ (besucht am 10. 04. 2018) (siehe S. 32, 51).

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Acknowledgments

I thank ?? and ?? for giving me the opportunity to write this bachelor/master/phd thesis at ??, and for their professional advise.

I thank in particular the ?? team who readily/willingly provided information at any time and ??.

I would also like to than all people who supported me in writing this thesis.

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die Zitate deutlich kenntlich gemacht zu haben.

Dresden, den 22. Mai 2018 Felix Dimmel

A First chapter of appendix

A.1 Parameters