Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **3**

3 Vergleich zwischen Kotlin und dem Google Web Toolkit **5**

4 Konzeption des Servers **7**

4.1 Anforderungen 7

4.1.1 Ressource: Player (Spieler) 7

4.1.2 Ressource: Match (Partie) 8

4.1.3 Ressource: Draw (Zug) 8

4.2 Verwendete Bibliotheken/Frameworks 8

4.2.1 Spring 8

4.2.2 SQLite 10

4.2.3 ORMLite 11

4.3 Ressourcenzugriffe mithilfe von Spring Controllern 13

4.3.1 Player Controller 13

4.3.2 Match Controller 14

4.3.3 Draw Controller 14

4.3.4 Game Controller 14

4.3.5 Error Controller 14

5 Konzeption des Clients **11**

6 Implementation des Servers **13**

7 Implementation des Clients **15**

8 Fazit **17**

9 Ausblick **19**

Literatur **21**

Anhang **33**

A First chapter of appendix **33**

A.1 Parameters 33

KAPITEL 4

Konzeption 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. Im zweiten Abschnitt werden die verwendeten Bibliotheken bzw. Frameworks in den Punkten Zweck, Einrichtung und Verwendung näher erläutert. Der letzte 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. Des weiteren sollen alle Ressourcenelemente in einer SQLite Datenbank gespeichert werden, welche für jedes Element eine eindeutige ID generiert.

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.

4.1.1 Ressource: Player (Spieler)

Neben der ID die wie schon in Abschnitt 4.1 erwähnt durch die SQLite Datenbank generiert werden soll, besitzt der Player noch die Information über den Name und das Passwort des Players.

Eine nachträgliche Änderung des Namens soll dabei nicht gestattet sein. Das Passwort hingegen soll zu jeder Zeit aktualisierbar sein.

4.1.2 Ressource: Match (Partie)

Auch bei der Ressource Match wird die ID durch SQLite generiert. Des weiteren besitzt ein Match Informationen über die beiden Spieler(Weiß/Schwarz) und deren Figurenstellung. Ein Match weiß außerdem welche Farbe am Zug ist, ob einem Spieler eine Rochade zur Verfügung steht, ob ein Feld existiert für ein Schlag en passant und wie viele Halbzüge getätigt wurde seit dem zuletzt ein Bauer gezogen oder eine Figur geschlagen wurde. Zusätzlich kann über ein Match ermittelt werden ob ein Spieler im Schach steht oder das Spiel schon bis zum Schachmatt gespielt wurde. Zuletzt hält das Match noch den aktuellen Stand in der Forsyth-Edwards-Notation (FEN).

4.1.3 Ressource: Draw (Zug)

Die Ressource Draw speichert zusätzlich zur ID die Farbe des Spielers, die Art der Spielfigur, Start- und Endfeld, ob eine Figur geschlagen wurde und wenn ja ob durch en passant und ob seitens der Dame oder des König rochiert wurde. Zusätzlich hält der Draw seine Daten in der Standard Algebraic Notation (SAN).

4.2 Verwendete Bibliotheken/Frameworks

Die verwendeten Bibliotheken sollen die Umsetzung des Projektes vereinfachen und beschleunigen. Dabei werden diese in den nachfolgenden Unterabschnitten 4.2.1 bis 4.2.3 in den Punkten Zweck, Einrichtung und Verwendung näher betrachtet.

4.2.1 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.[1]

Um Spring in einem Projekt einzubinden, müssen folgende Zeilen, zusätzlich zu denen welche Kotlin benötigt, in der Build-Datei „build.gradle“ eingefügt werden:

```
1 buildscript {
2   repositories {
```

1 siehe [Cos17]

```
    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 4.1: Einbindung des Spring Framework mithilfe von Gradle

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.

Für die Erstellung eines Representational State Transfer (REST) Application Programming Interface (API) muss zunächst ein Controller für eine Ressource angelegt werden, dabei ist es sinnvoll für jede einen eigenen Controller zu definieren. Innerhalb werden anschließend alle Einstiegspunkte erzeugt. Als Beispiel soll ein klassisches *Hello World* dienen siehe Listing 4.2. 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 das API, *@GetMapping* ein Einstiegspunkt (Request Methode: GET) und *@RequestParam* einen Parameter für diese Funktion definiert.

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

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

```
  }
}
```

Listing 4.2: Beispiel: Spring Controller

Abschließend muss nur noch der Startpunkt für die Applikation definiert werden. Dafür wird wieder mithilfe einer Annotation ein Klasse erzeugt, welche aber keinerlei Informationen benötigt. Anschließend muss diese in der Main-Funktion gerufen werden siehe Listing 4.3.

```
@SpringBootApplication
class Application

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

Listing 4.3: Beispiel: Spring Application Class

Für eine detaillierte Beschreibung dieses Beispiels stehen auf den Internetseiten [Har] und [Inc] weitere Informationen bereit.

4.2.2 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.[1]

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:

```
repositories {
  mavenCentral()
}

dependencies {
  compile group: 'org.xerial', name: 'sqlite-jdbc', version: '3.21.0.1'
}
```

1 siehe [Hip]

Listing 4.4: Einbindung der Bibliothek SQLite mithilfe von Gradle

Ein 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 verwendet werden soll[1], wird eine nähere Betrachtung für die Verwendung von SQLite mithilfe des JDBC Treibers vernachlässigt.

4.2.3 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 Abbildung 4.5 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 Kapitel 4.2.2 für SQLite erläutert, der Datenbank-Treiber für das verwendete Datenbank-System eingebunden werden.

```
1 repositories {
2   mavenCentral()
3 }
4
5 dependencies {
6   compile "com.j256.ormlite:ormlite-core:5.1"
7   compile "com.j256.ormlite:ormlite-jdbc:5.1"
8 }
```

Listing 4.5: Einbindung der Bibliothek ORMLite mithilfe von Gradle

Für die Persitierung einzelner Klassen können durch ORMLite bereitgestellte Annotationen verwendet werden. (siehe Abbildung 4.6)

ggf. Syntax-Highlighting einrichten

```
1 @DatabaseTable(tableName = "accounts")
2 public class Account {
```

1 siehe Kapitel 4.2.3

```
  @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 4.6: Beispiel: Persistierung einer Klasse mittels ORMLite[1]

Der Zugriff auf die Datenbank 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 die DAOs 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 sogenannten Query-Buildern. Zur Veranschaulich der Verwendung von ORMLite zeigt die Abbildung 4.7 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());

```

1 Quelle: [Wat]

```
15 connectionSource.close();
```

Listing 4.7: Beispiel: Verwendung von ORMLite[1]

4.3 Ressourcenzugriffe mithilfe von Spring Controllern

Die einzelnen Zugriffe auf die Ressourcen werden in den Kapiteln 4.3.1 bis 4.3.5 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 diesen Request kann ermittelt werden welche Request-Methoden für den jeweiligen Einstiegspunkt zur Verfügung stehen. Alle Anfragen geben dabei ihr Feedback in der JavaScript Object Notation (JSON) zurück.

ggf. noch XML erwähnen bzw. wenn es nicht mit angeboten wird erwähnen das es vernachlässigt wird.

4.3.1 Player Controller

Der Player Controller soll zwei Einstiegspunkt an den glsplURI „/player/“ und „/player/id“. Der Parameter *id* dient dabei als Platzhalter für eine ID eines Players, welche wiederum eine Zahl darstellt.

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 ID wird dabei von der SQLite-Datenbank automatisch mittels *Autoincrement* erzeugt. Bei erfolgreicher Erstellung des Players wird dieser als Objekt zurückgegeben, 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 soll das Passwort aktualisiert werden können. Der GET- und DELETE-Request beziehen sich dabei auf einen Player anhand der ID innerhalb des Uniform Ressource Indetifier (URI) und benötigen keine weiteren Parameter. Um das Passwort des Players zu aktualisieren muss dieses natürlich als Parameter mitgegeben werden, aber auch bei diesem Request wird anhand der ID der Player identifiziert.

1 verändert nach [Wat]

4.3.2 Match Controller

4.3.3 Draw Controller

4.3.4 Game Controller

Der Game Controller soll dazu dienen Match bezogene Daten zu ermitteln. Dabei soll mittels GET-Request ein Liste aller Draws die ein Match besitzt, über den URI „/match/id/draw“, bereitgestellt werden. Der Parameter *id* dient dabei wieder als Platzhalter für die ID des Matches von welchem alle Draws ermittelt werden sollen.

4.3.5 Error Controller

Mithilfe des Error Controllers soll gewährleistet werden, dass alle nicht in der REST-API definierten Einstiegspunkte, bzw. nicht definierte Request-Methoden an den Einstiegspunkten abgefangen werden. Tritt so ein Fall auf soll „Wrong Request“ vom Server zurückgegeben werden. Somit wird verhindert das ein Nutzer des API einen Serverfehler, mit dem HTTP-Statuscode *500*[1].

1 siehe [Kre15, A.2.5]

Literatur

[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. 8).

[Har] HARIRI, HADI, EDOARDO VACCHI und SÉBASTIEN DELEUZE: „Creating a RESTful Web Service with Spring Boot“. (), Bd. URL: https://kotlinlang.org/docs/tutorials/spring-boot-restful.html (besucht am 04. 04. 2018) (siehe S. 10).

[Hip] HIPP, WYRICK & COMPANY INC.: „About SQLite“. (), Bd. URL: https://www.sqlite.org/about.html (besucht am 09. 04. 2018) (siehe S. 10).

[Inc] INC., PIVOTAL SOFTWARE: „Building a RESTful Web Service“. (), Bd. URL: https://spring.io/guides/gs/rest-service/ (besucht am 04. 04. 2018) (siehe S. 10).

[Kre15] KRETZSCHMAR, CHRISTOPH: „Demonstration eines RESTful Webservices am Beispiel eines Schachservers“. Bachelor. Hochschule für Technik und Wirtschaft Dresden, 2015 (siehe S. 14).

[Lim] LIMITED, TUTORIALS POINT INDIA PRIVATE: „SQLite - Java“. (), Bd. URL: https://www.tutorialspoint.com/sqlite/sqlite_java.htm (besucht am 09. 04. 2018) (siehe S. 11).

[Wat] WATSON, GRAY: „OrmLite - Lightweight Object Relational Mapping (ORM) Java Package“. (), Bd. URL: http://ormlite.com/ (besucht am 10. 04. 2018) (siehe S. 12, 13, 27).

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

4.1 Einbindung des Spring Framework mithilfe von Gradle 8
4.2 Beispiel: Spring Controller 9
4.3 Beispiel: Spring Application Class 10
4.4 Einbindung der Bibliothek SQLite mithilfe von Gradle 10
4.5 Einbindung der Bibliothek ORMLite mithilfe von Gradle 11
4.6 Beispiel: Persistierung einer Klasse mittels ORMLite[1] 11
4.7 Beispiel: Verwendung von ORMLite[1] 12

1 Quelle: [Wat]
1 verändert nach [Wat]

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 11. April 2018 Felix Dimmel

A First chapter of appendix

A.1 Parameters