

REST API mit Java Spring

Management von Daten einer Hobby-Eishockeymannschaft mit
Spring Boot und Hibernate

BACHELORARBEIT

MARC RAEMY

Juni 2020

Unter der Aufsicht von:

Prof. Dr. Jacques PASQUIER-ROCHA

and

Pascal GREMAUD

Software Engineering Group

Danksagung

Mein Dank geht an Prof. Dr. Jacques Pasquier-Rocha und Pascal Gremaud für die unkomplizierte und sehr unterstützende Betreuung dieser Arbeit. Ebenso an alle Personen, die sonst in einer Form zum erfolgreichen Abschluss der Arbeit beigetragen haben. Andreas Ruppen sei gedankt für die Erstellung des LaTeX-Templates, das für diese Arbeit verwendet werden konnte.

Abstract

In dieser Arbeit wurde ein Application Programming Interface (API) nach REST-Prinzipien mit dem Java Spring Framework erstellt. Die API dient zur Speicherung von Daten einer Hobby-Eishockeymannschaft. Dazu wurden Spring Boot, Java Persistence API (JPA), Hibernate und eine PostgreSQL-Datenbank eingesetzt. Die Arbeit beschreibt diese Tools und den Entwicklungsprozess anhand dieses Fallbeispiels.

Keywords: Java Spring Framework, Spring Boot, ORM, Hibernate, REST API

Inhaltsverzeichnis

1. Einleitung	2
1.1. Motivation und Ziel	2
1.1.1. Eishockeyclub HC Keile	2
1.1.2. Anwendungsfälle einer möglichen HC Keile-Applikation	4
1.1.3. Übergeordnete Ziele und Abgrenzung des Themas	5
1.2. Aufbau der Arbeit	6
1.3. Notation and Konventionen	7
2. Daten, Anwendungsfälle und Endpoints	8
2.1. Daten HC Keile	8
2.2. Anwendungsfälle (Use Cases)	9
2.2.1. System	10
2.2.2. Akteure	10
2.2.3. Assoziationen	11
2.2.4. Anwendungsfälle	11
2.3. Endpoints	14
3. Spring Framework, ORM und REST	16
3.1. Spring Framework	16
3.1.1. Dependency Injection	18
3.1.2. Aspect Oriented Programming	18
3.1.3. Spring Framework Kern	19
3.1.4. Spring Boot	19
3.1.5. Spring Projekte	20
3.2. Objekt/Relationales Mapping (ORM)	22
3.2.1. Object/relational Mismatch	22
3.2.2. JPA und Hibernate	23
3.3. RESTful API	23

4. Schema der Datenbank	25
4.1. Entity Relationship Model (ERM)	25
4.2. Verbale Beschreibung	26
4.3. Tabellen	27
4.4. Domain Model vs. ERM	28
5. Programmierung und Dokumentation der API	30
5.1. Installation von Spring	30
5.1.1. Spring Initializr	30
5.2. Quellcode	31
5.2.1. Controller	31
5.2.2. Repositories	32
5.2.3. Entities	34
5.2.4. Main-Klasse	35
5.3. Persistence Layer	35
5.3.1. JPA und Hibernate	35
5.3.2. Database Layer	37
5.4. Dokumentation mit Swagger	38
6. Abschliessende Bemerkungen	41
6.1. Zusammenfassung	41
6.2. Probleme und Herausforderungen	41
A. Abkürzungen	43
B. Lizenz	45
Literaturverzeichnis	46
Webreferenzen	46

Abbildungsverzeichnis

1.1. Mannschaft HC Keile	3
2.1. Use Case Diagram	10
2.2. Screenshot Beispiel Daten Spiel HC Keile	13
3.1. Spring Framework Core	21
4.1. Entitäten-Beziehungsmodell	26
5.1. Spring Initializr Onlinetool	31
5.2. Swagger Interface	40

Tabellenverzeichnis

2.1. Auszug Rohdaten, CSV Format	9
4.1. Tabelle Game	27
4.2. Tabelle Goal	27
4.3. Tabelle Player	28
4.4. Tabelle Opponent	28
4.5. Tabelle Games played	28

Listings

5.1. JpaRepository Interface	33
5.2. Maven Dependencies Swagger	38
5.3. Swagger Configuration Class	38

1

Einleitung

1.1. Motivation und Ziel	2
1.1.1. Eishockeyclub HC Keile	2
1.1.2. Anwendungsfälle einer möglichen HC Keile-Applikation	4
1.1.3. Übergeordnete Ziele und Abgrenzung des Themas	5
1.2. Aufbau der Arbeit	6
1.3. Notation and Konventionen	7

1.1. Motivation und Ziel

Das Ziel der vorliegenden Arbeit ist es, ein Application Programming Interface (API) zu programmieren, das von einer Applikation, die den Hobby-Eishockeyclub HC Keile bei seinen Aktivitäten unterstützen könne, eingebunden werden könnte. In diesem Kapitel wird der HC Keile und seine Aktivitäten näher vorgestellt, dann wird das Ziel und der Aufbau dieser Arbeit vorgestellt.

1.1.1. Eishockeyclub HC Keile

In der Eishockeyszene in der Schweiz gibt es eine breite Palette von Mannschaften, die ausserhalb von Verbands- und Meisterschaftsstrukturen regelmässig Freundschaftsspiele untereinander bestreiten. Das wird etwa „wilde Liga“ oder „Plausch-Cup“ (auf Hochdeutsch übersetzt „Spasch-Cup“) genannt. In der Region Freiburg mit ihrer grossen Eishockeytradition ist diese Szene vermutlich noch grösser als in anderen Teilen der Schweiz. Der HC Keile ist eine Mannschaft aus Freiburg (CH), die in dieser Form den Eishockeysport seit 2011 als Hobby praktiziert. Die Mannschaft spielt nur Freundschaftsspiele und unterhält keine Trainingsbetrieb. Die Spiele des HC Keile finden in unregelmässigen Abständen üblicherweise am frühen Sonntagabend in der Eishalle im St.Léonard in Freiburg statt. Danach wird im Café du Gothard eingekehrt.

Die meisten Spieler sind nebenbei in anderen Vereinen oder auf privater Basis sportlich Aktiv oder spielen und trainieren noch in anderen Plausch-Eishockeymannschaften. Der Kern der Mannschaft besteht aus einer Gruppe von Freunden, die aus dem Freiburgischen

Sensebezirk stammen und sich seit der Kindheit kennen oder in ihrer Zeit als aktive Fussballer beim SC Dürdingen und beim FC Freiburg kennengelernt haben.

Der HC Keile hat eine ausgeprägte Tradition des nicht ganz ernst gemeinten „Selbst-Kultes“, mit einer Facebook-Seite, einer Whatsapp-Gruppe, diversen Awards und Pokalen, die in erster Line der eigenen Unterhaltung dienen, sowie einer jährlichen Vereins-Generalversammlung im Restaurant „Pinte des Trois Canards“ im Galterntal in Freiburg, wo auch die Ursprünge des städtischen Eishockeyclubs HC Freiburg-Gottéron liegen. Der Verein führt auch Scoring- und weitere Statistiken zu Team und Spielern, die an der Generalversammlung präsentiert werden.



Abbildung 1.1.: Mannschaft HC Keile

Rollen und Aufgaben

Seit dem ersten Spiel werden an den Spielen des HC Keile nebst dem Resultat und Namen des Gegners die Torschützen und Assistgeber sowie die anwesenden Spieler des HC Keile notiert. Resultat, Gegner und Scorerpunkte vor allem zum Spass, das letzte daneben auch aus Gründen der Verrechnung der Kosten für die Eismiete unter den Spielern.

Ein Spieler des HC Keile hat das Amt der Datenmanager, ein zweiter ist seine Stellvertreter. Diese Spieler notieren jeweils an den Spielen, welche Spieler des HC Keile das Spiel absolviert haben, wer die Tore geschossen und Assists gegeben hatte, sowie das Resultat und gegen welche Mannschaft der Gegner wurde. Die Statistiken trägt sie dann zu Hause von Hand in einer Excel-Tabelle ein, die als Datenbank der HC Keile-Statistiken dient.

An der Generalversammlung präsentiert diese Person jeweils Auswertungen, die er in den Excel-Tabellen vorgenommen hat. Diese umfassen die Scorerstatsitiken, Siegesquoten, Anzahl gespielte Spiele pro Spieler und weitere Kennzahlen wie z.B. die Sieges-quote von jedem Spieler.

Ein weiteres Amt ist das Amt des Spiele-Organisators. Er organisiert die Spiele und Gegner und führt ein Doodle, in dem sich die Spieler für die Spiele eintragen können. Der Kassier der Mannschaft stützt sich auf dieses Doodle, um am Ende des Jahres den Spielern entsprechend der Anzahl der gespielten Spiele eine Rechnung für Ihren Beitrag an die Eismiete zu stellen und kümmert sich um alle übrigen finanziellen Angelegenheiten.

1.1.2. Anwendungsfälle einer möglichen HC Keile-Applikation

Im Folgenden werden die Anwendungsfälle (engl. „Use Case“) einer möglichen Applikation beschrieben, welche in dieser Arbeit programmierte API nutzt. Das Ziel der Applikation wäre, die Erhebung und Konsultation der Spielerstatistiken und Resultate des HC Keile zu erleichtern respektive zu ermöglichen.

Für weitere Bereiche der Aktivitäten des HC Keile besteht im Moment wenig Bedarf für eine Applikation: Die Organisation der Spiele mit Hilfe des Doodle, wie oben beschrieben, funktioniert gut. Und Für die finanziellen Abwicklungen, die einen geringen Umfang haben, besteht ebenfalls kein Bedarf für eine Applikation, abgesehen von einer Funktionalität, welche die API abdeckt (siehe Anwendungsfall 3 unten). Die API beschränkt ihren Anwendungsbereich deshalb auf die Spielerstatistiken und Resultate des HC Keile. Bei Bedarf könnte sie künftig erweitert werden. Das ist jedoch nicht Teil dieser Arbeit.

Anwendungsfall 1: Daten der Spiele erheben

Die Daten werden nach den Spielen von Hand notiert (siehe Beispiel eines Eintrags in Abbildung 2.2), die Datenmanager*in trägt diese danach zu Hause ein zweites Mal von Hand in eine Excel Tabelle ein. Für die Präsentation der Daten an der Generalversammlung macht sie Auswertungen lokal im Excel und präsentiert die Ausdrucke davon.

Es wäre sehr praktisch, wenn man, via eines Smartphones oder Tablets mit Internetverbindung, die Daten direkt online in die Datenbank eintragen könnte. Beispielsweise eines Webformulars via Browser (oder einer Applikation, was aber wegen verschiedenen Betriebssystemen schon schwieriger ist). Dies Funktionalität ist der erste und Hauptanwendungsfall, den die API abdecken soll.

Daraus ergeben sich weitere Anwendungsfälle. Erstens, wenn der Eintrag fehlerhaft war, den Spieleintrag zu korrigieren. Und zweitens, wenn Spieler am Spiel waren, die noch nicht in der Datenbank gespeichert sind, oder eine gegnerische Mannschaft, die noch nicht gespeichert ist, dass man diese Entitäten in die Datenbank eintragen kann.

Anwendungsfall 2: Daten der Spiele und Spieler darstellen

Da es zweitens die Spieler interessiert und es ihnen Spass macht, die Scorerliste und die Resultate und Anzahl absolvierte Spiele zu verfolgen, wäre eine weitere Funktionalität, welche die API einem Client ermöglichen sollte, die Scoringstatistiken der Spieler sowie die Resultate der Spiele darzustellen. Somit könnten die Statistiken laufend online eingesehen und verfolgt werden, von überall her, und würden nicht nur an der Generalversammlung einmal pro Jahr präsentiert. Es wäre eine Ergänzung, da nicht alle Auswertungen, die an der Versammlung präsentiert werden, ständig verfügbar sein müssten - dieses Traktandum macht Spass und müsste nicht unbedingt gestrichen werden - sondern einfach, dass

in Ergänzung dazu laufend die Scorerlisten online einsehbar wären. Das ist der zweite Anwendungsfall, den die in dieser Arbeit erstellte API abdecken sollte

Anwendungsfall 3: Anzahl Spiele pro Spieler*in pro Saison ablesen

Der Kassier des HC Keile verschickt, um die Kosten der Miete der Eishalle zu verrechnen am Ende jeder Saison eine individuelle Rechnung an jede Spieler*in, in Abhängigkeit der Anzahl Spiele, die sie absolviert hat. Bisher liest er die Anzahl Spiele pro Spieler*in aus dem Doodle, was relativ mühsam ist.

Der dritte Anwendungsfall der API ist deshalb, zu ermöglichen, dass der Kassier die Anzahl der absolvierten Spiele der Saison jeder Spieler*in bequem ablesen kann, ohne dass er mühsam alle Doodle-Einträge durchforsten muss. Das würde den Abrechnungsprozess erleichtern.

Diesen drei Zwecken soll die in dieser Arbeit erstellte API dienen. Kapitel 2 beschreibt diese Anwendungsfälle im Detail.

1.1.3. Übergeordnete Ziele und Abgrenzung des Themas

Übergeordnete Ziele

Das übergeordnete Ziel der Arbeit ist, die in den Vorlesungen im Softwareengineering und in diversen Wirtschaftsinformatikvorlesung an der Universität Freiburg erlangten Kenntnisse anhand eines Fallbeispiels in der Praxis anzuwenden. Da in den Vorlesungen die Programmiersprache Java unterrichtet wurde, wurde ein Framework in dieser Programmiersprache für die Arbeit gewählt, um die Java Kenntnisse zu vertiefen. Speziell die Kenntnisse in Web- und Applikationsentwicklung sollen angewendet und vertieft werden.

In Absprache mit Prof. Pasquier und Pascal Gremaud wurde das Java Spring Framework gewählt. Da dort häufig ORM eingesetzt wird, und dies eine interessante und praktische Technologie ist, wurde für die Persistenzschicht ORM mit der JPA und dem Hibernate Framework ausgewählt. Für Entwicklungszwecke wurde im Verlauf der Arbeit die in-Memory-Datenbank H2 genutzt und aus persönlicher Sympathie und Interesse PostgreSQL als persistente Datenbanktechnologie. Damit sollen aktuell in der Praxis häufig verwendeten Tools für das Erstellen von Webapplikationen verwendet und kennengelernt werden.

Im Rahmen der Umsetzung werden weiter diverse im Wirtschaftsinformatik erworbenen Kenntnisse angewendet: Entwurf und Konzeption der relationalen Datenbank mit Erstellen eines Entitäten-Beziehungsmodells und des daraus Ableitens der Entsprechenden Beziehungen und Tabellen, Definition der Anwendungsfälle und ihre Darstellung in einem Anwendungsfalldiagramm sowie das Darstellen von Prozessen mit der Business Process Modelling Notation (BPMN). Eher in den Bereich Softwareengineering gehört das Übersetzen der Anforderungen in entsprechende Endpoints der API sowie das Implementieren dieser Endpoints gemäss REST-Prinzipien. Das Thema ORM mit Spring Data JPA und Hibernate nimmt dabei einen relativ grossen Teil der Arbeit ein.

Abgrenzung des Themas

Es ist nicht das Ziel der Arbeit ein vollständiges Back-End, oder eine Fullstack-Applikation inklusive Client zu programmieren. Der Umfang der Arbeit konzentriert sich grösstenteils auf die Back-End-Seite und dort auf die Entwicklung der REST-API sowie ihrer Dokumentation mit dem gängigen Dokumentationstool Swagger. Aspekte wie Authentifizierung und Sicherheit oder das Testen mit Spring Boot sind nicht Teil der Arbeit und für diesen konkreten Anwendungsfall auch nicht nötig. Die Endpoints sind nach REST-Prinzipien erstellt, die REST-Thematik wird jedoch nur am Rande berührt und steht nicht im Zentrum der Arbeit. Es handelt sich um eine Praxisarbeit, welche so weit wie möglich mit den theoretischen Konzepten in die Tiefe geht. Es ist jedoch nicht Ziel der Arbeit, diese Konzepte erschöpfend theoretisch abzuhandeln.

1.2. Aufbau der Arbeit

Kapitel 1: Einleitung

In der Einleitung werden das Ziel und Thema der Arbeit, ihr Aufbau sowie Informationen zu Notation und Konventionen vorgestellt.

Kapitel 2: Daten, Anwendungsfälle und Endpoints

In diesem Kapitel werden die Daten des HC Keile, die Anwendungsfälle sowie die daraus abgeleiteten Endpoints beschrieben.

Kapitel 3: Spring Framework, ORM und REST

In diesem Kapitel werden die verwendeten Technologien und Konzepte beschrieben: Das Java Spring Framework, Spring Boot, ORM, Hibernate und REST.

Kapitel 4: Schema der Datenbank

In diesem Kapitel wird ein Entitäten-Beziehungs-Modell sowie die daraus abgeleiteten Tabellen beschrieben.

Kapitel 5: Programmierung und Dokumentation der API

In diesem Kapitel werden der Quellcode, das Vorgehen bei der Programmierung, die Dokumentierung der API mit Swagger vorgestellt.

Kapitel 6: Abschliessende Bemerkungen

Im abschliessenden Kapitel werden die Zielerreichung der Arbeit überprüft und die Probleme und Herausforderungen der Arbeit beschrieben.

Anhang

Der Anhang enthält Literatur- und Onlinereferenzen, die Liste der Abkürzungen sowie eine Lizenzangabe für die Software.

GitHub Repository

Der Quellcode sowie alle Dokumente des Projekts sind auf GitHub verfügbar unter <https://github.com/MarcRaemy/keilestats-api>.

1.3. Notation and Konventionen

Das Format der Arbeit basiert auf einem LaTeX-Template, das Andreas Ruppen für die Softwareengineering-Forschungsgruppe der Universität Freiburg (CH) erstellt hat. Die Notationen und Konventionen wurden grösstenteils daraus übernommen.

- Formatierung:
 - Abkürzungen werden wie folgt verwendet: Hypertext Transfer Protocol (HTTP) bei der ersten Verwendung und HTTP bei jeder weiteren Verwendung;
 - Webadressen in folgender Form: `http://localhost:8080/api`;
 - Format Quellcode:

```
1 public double division(int _x, int _y) {  
2     double result;  
3     result = _x / _y;  
4     return result;  
5 }
```

- Die Arbeit ist in sechs Kapitel unterteilt, die jeweils Unterkapitel enthalten. Jedes Unterkapitel hat Paragraphen, welche eine gedankliche Einheit repräsentieren.
- Grafiken, Tabellen und Auflistungen sind innerhalb eines Kapitels nummeriert. Beispielsweise wird eine Referenz auf Grafik *j* des Kapitels *i* nummeriert als *Figure i.j*.
- Bezüglich der geschlechtlichen Form wird, da nur in seltenen Fällen Frauen oder intersexuelle Menschen beim HC Keile mitspielen, der Kürze und Einfachheit halber die männliche Form verwendet. Frauen und intersexuelle Personen sind dabei mitgemeint. An gewissen Stellen wird in der Arbeit ebenfalls die Konvention mit weiblicher Form und Stern verwendet, z.B. „Entwickler*innen“, welche Männer und intersexuelle Personen ebenfalls einschliesst.

2

Daten, Anwendungsfälle und Endpoints

2.1. Daten HC Keile	8
2.2. Anwendungsfälle (Use Cases)	9
2.2.1. System	10
2.2.2. Akteure	10
2.2.3. Assoziationen	11
2.2.4. Anwendungsfälle	11
2.3. Endpoints	14

2.1. Daten HC Keile

Die Daten zu den Torschützen, Assistgeber und Resultaten der Spiele seit dem ersten Spiel des HC Keile sind als Excel-Export im .csv oder .txt Format als Rohdatensatz oder als Excel-File festgehalten. Die Daten wurden vom Datenmanager*in in Excel auf seine Art formatiert. Tabelle 2.1. zeigt ein Auszug im .csv-Format aus dieser Excel-Datei. Diese Daten bei einem Live-gehen einer Applikation umformatiert und in die Datenbank migriert.

Statistiken der Spieler und Spiele

Form der Rohdaten									
ID	MatSaison	MatchNr	SpAka	SpPos	SpT	SpA1	SpA2	GeT	SpRes
...									
1145	201617	22	Stefu	G	0	0	0	4	gewonnen
1146	201617	22	Twenta	V	0	0	1	0	gewonnen
1147	201617	22	Doemu	V	0	1	0	0	gewonnen
1148	201617	22	Reamy	V	0	0	0	0	gewonnen
1149	201617	22	Hunk	V	0	0	0	0	gewonnen
1150	201617	22	Elu	S	2	0	1	0	gewonnen
1151	201617	22	Michu	S	1	2	0	0	gewonnen
1152	201617	22	Chraebli	S	2	0	1	0	gewonnen
1153	201617	22	Dave	C	2	0	0	0	gewonnen
1154	201617	22	Sebi	S	0	0	0	0	gewonnen
1155	201617	22	Flogge	C	0	1	0	0	gewonnen
1156	201617	22	Pavel	S	2	0	1	0	gewonnen
1157	201617	22	Roman	S	1	1	0	0	gewonnen
1158	201718	1	StefuG	G	0	0	0	8	verloren
1159	201718	1	Stephu	V	0	0	0	0	verloren
1160	201718	1	Twenta	V	0	0	0	0	verloren
1161	201718	1	Michu	S	0	2	0	0	verloren
1162	201718	1	Fongs	V	0	0	0	0	verloren
1163	201718	1	Sebi	G	2	0	0	0	verloren
1164	201718	1	Flogge	G	0	0	1	0	verloren
1165	201718	1	Dave	C	0	0	1	0	verloren
1166	201718	1	Katja	S	0	0	0	0	verloren
...									

Tabelle 2.1.: Auszug Rohdaten, CSV Format

ID = ID des Eintrags/Excel-Zeile, *MatSaison* = Saison, *MatchNr* = Nummer Match der laufenden Saison (1 = erstes Spiel der Saison), *SpAka* = Rufname des Spielers, *SpPos* = Position (*G* = Goalie, *S* = Sturm (Flügel), *C* = Center, *V* = Verteidiger), *SpT* = erzielte Tore, *SpA1* = Anzahl erste Assists, *SpA2* = Anzahl zweite Assists, *GeT* = Gegentore (nur für Goalie), *SpRes* = Resultat -> gewonnen, verloren oder unentschieden

Die Migration der Daten in die Datenbank der API steht nicht im Zentrum dieser Arbeit und wird nicht weiter beschrieben.

2.2. Anwendungsfälle (Use Cases)

Ausgehend von den im Kapitel 1 beschriebenen Ausgangslage und den Zielen der Arbeit werden in diesem Unterkapitel Anwendungsfälle aus diesen Zielen abgeleitet. Abbildung 2.1. gibt eine Übersicht über diese Anwendungsfälle.

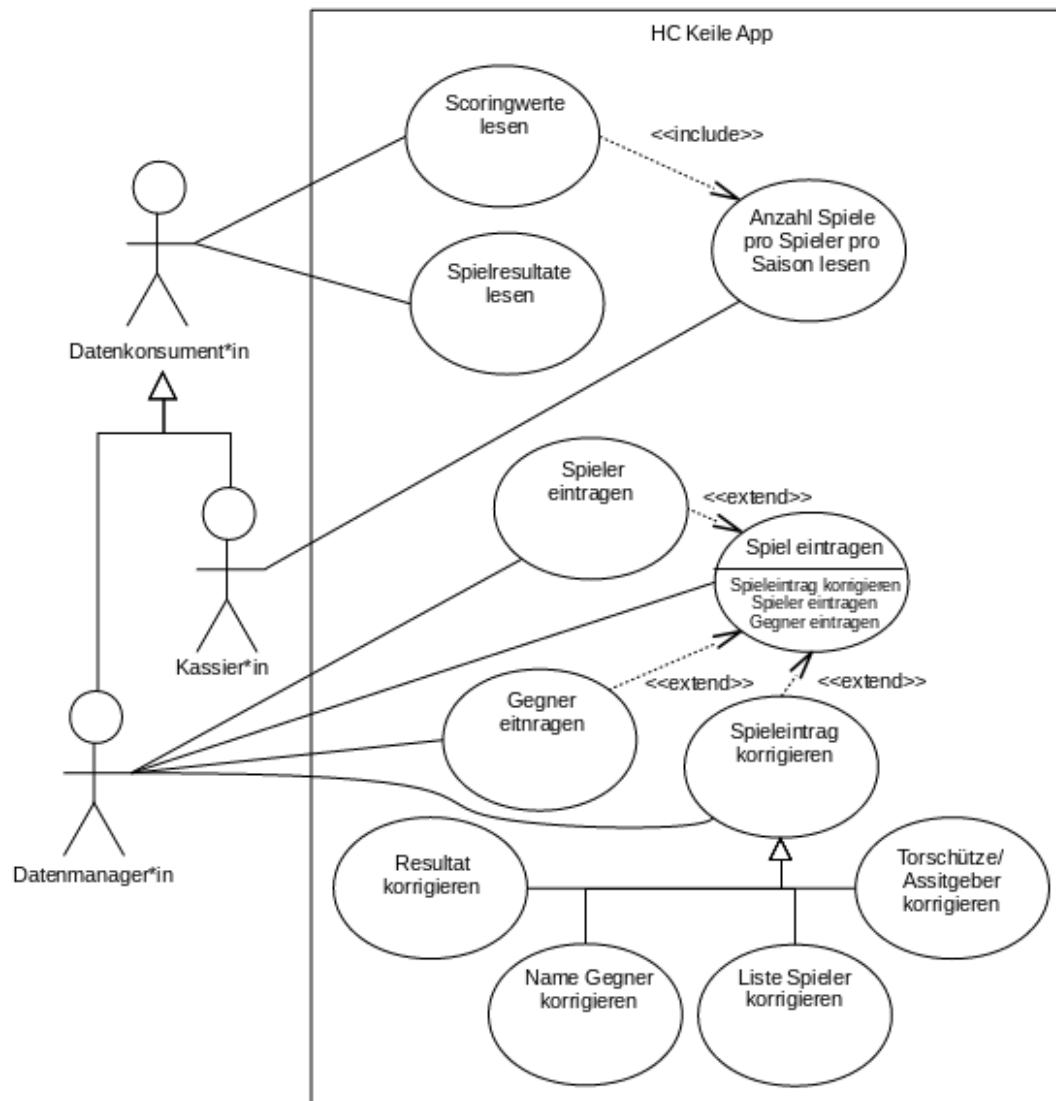


Abbildung 2.1.: Use Case Diagramm

2.2.1. System

Das Rechteck repräsentiert das System, die Applikation, respektive alles, was in der App. im System passiert. Ausserhalb des Rechtecks sind von der Applikation externe Akteure/Elemente - alles was ausserhalb des Systems passiert, dargestellt. In diesem Fall ist das System eine mögliche Applikation für den HC Keile, welche die API, die in dieser Arbeit programmiert wird, nutzt,

2.2.2. Akteure

Die API dient in erster Linie dem HC Keile und seinen Spielern. Ausserhalb des HC Keile dürften sehr wenige Personen einen Nutzen und Interesse an den Spielresultaten und Scoringzahlen des HC Keile haben. Die Nutzenden und übergeordnete Akteure sind somit die Spieler*innen des HC Keile. Einige haben eine spezifische Aufgabe, für welche die Application Programming Interface (API) relevant ist.

Das ist erstens die Kassier*in, die aus den Daten die Anzahl Spiele pro Spieler abliest und anhand dessen die Rechnung für die Eismiete für jede Spieler*in erstellt und zweitens die Datenmanager*in, welche die Scoringdaten, Resultat und Gegner an den Spiele notiert und für den HC Keile sammelt. Dies sind Spezialfälle einer Spieler*in, die für das System relevant sind und sind deshalb im Diagramm in einer Generalisierungsbeziehung zur Akteur*in Datenkonsument*in dargestellt, als welche die Spieler*in dort, etwas allgemeiner formuliert, bezeichnet ist.

2.2.3. Assoziationen

Die Begriffe Assoziation und Beziehung werden in diesem Abschnitt synonym verwendet. Die Generalisierungsbeziehung zwischen den Akteuren ist im vorhergehabten Abschnitt beschrieben.

Der Use Case „Spieleintrag korrigieren“ kann spezifischer bedeuten, entweder den Namen eines Torschützen oder Assistgebers zu korrigieren, die Spielerliste des Spiels zu korrigieren, das Resultat zu korrigieren, oder evtl. sogar den Namen des Gegners zu korrigieren. Dies ist mit Hilfe einer Generalisierungsbeziehung im Use Case Diagramm dargestellt.

Der Use Case „Spieleintrag korrigieren“ erweitert den Use Case „Spiel eintragen“. Das bedeutet, dass in einigen Fällen, aber nicht jedes Mal, wenn ein Spiel eingetragen wird, danach eine Korrektur vorgenommen werden.

Die selbe Beziehung besteht zwischen dem Use Case „Scoringwerte lesen“ und „Anzahl Spiele pro Spieler und Saison lesen“. Wenn man die Scoringwerte liest, könnte es interessant sein, diese nach Saison zu ordnen, was automatisch auch die Anzahl Spiele, die ein Spieler pro Saison gespielt hat, anzeigen würde.

Der Use Case „Anzahl Spiele pro Spieler und Saison lesen“ ist weiter enthalten im Use Case „Betrag Eismiete berechnen“. Letzterer besteht hauptsächlich darin, diese Anzahl aus dem System abzulesen. Der restliche Teil der Berechnung findet ausserhalb des Systems statt.

2.2.4. Anwendungsfälle

Spielresultate lesen

Die Spieler des HC Keile sind interessiert daran, einen Überblick über die Spiele und Resultate der laufenden und verganener Spielzeiten zu haben. Dies ist der erste Use Case „Spielresultate lesen“. Die Datenkonsument*in konsultiert im System die Spielresultate des HC Keile. Das System stellt dabei das Datum, Gegner, Saison, evtl. Ort und Zeit sowie das Resultat der Spiele zur Verfügung.

Scoringwerte lesen

Zweitens möchten die Spieler*innen des HC Keile wissen, wer am meisten Tore und Assists erzielt hat, respektive, wie viele er oder sie selber erzielt hat und wie er im Vergleich zu den übrigen Spieler*innen steht. In dieser oder in verganenen Saisons.

Der zweite Use Case ist also das Konsultieren der Scoringwerte. Von Interesse sind dabei die Anzahl Tore, die jede Spieler*in geschossen hat, die Anzahl ersten und zweiten Assists, die Summe davon als Gesamtzahl der Scorerpunkte, wie viele Spiele die Person

gespielt hat, zu welcher Saison die Spiele gehören, sowie auf welcher Position (Verteidiger, Stürmer, Goalie) die Spieler*in spielt. Diese könnten dann vom Client beispielsweise in einer schönen Tabelle dargestellt werden. Als Vorbild dient hier etwa das Statistikportal der Schweizer Eishockeymeisterschaft (welches jedoch viel mehr Daten umfasst) <https://www.sihf.ch/de/game-center/national-league/#/mashup/players/player/points/desc/page/0/2019>

Um die Scorerpunkte besser interpretieren zu können ist auch relevant, wie viele Spiele ein Spieler gespielt hat. Der use Case „Anzahl Spiele pro Spieler pro Saison lesen“, ist daher als mit einer include-Beziehung in diesen Use Case assoziiert.

Spiel eintragen

Bei den Spielen des HC Keile werden Statistiken notiert, bis jetzt jeweils von Hand. Folgende Daten werden aufgeschrieben: Name des Gegners, Anzahl Tore des Gegners, Anzahl Tore HC Keile, Spieler des HC Keile, die am Spiel teilgenommen haben, Namen der Torschützen und Assistgeber des HC Keile. Über die Namen der Torschützen des Gegners wird nicht Buch geführt.

Die Daten werden dann bisher von Hand zu Hause von der Datenmanager*in in eine Excel-File übertragen, das auf einem privaten Rechner abgespeichert und nicht online zugänglich ist. Dieser Prozess hat soll in durch diesen Use Case erleichtert werden.

Und zwar gibt die Datenmanager*in die Daten nach dem Spiel in der Kabine, oder von unterwegs, oder von wo immer sie will, via eines Web-Formulars, das die API nutzt, in die Datenbank ein. Somit können die Daten nur einmal statt wie bisher zwei Mal aufgeschrieben werden und von überall her direkt in die Datenbank übertragen werden.

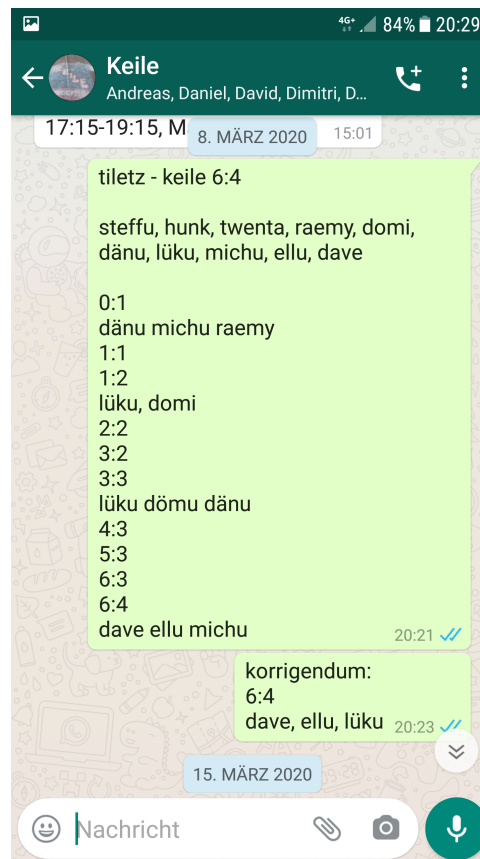


Abbildung 2.2.: Screenshot Beispiel Daten Spiel HC Keile

Datum, Ort und Zeit des Spiels eintragen

Die Spieler konsultieren bis jetzt das doodle, um nach zu schauen, wann das nächste Spiel ist, wo, und ob sie eingeschrieben sind. Das dürfte auch so beibehalten werden. Dennoch ist es denkbar, dass, künftig künftige Spiele auch über eine mögliche HC Keile-App konsultiert werden könnten. Zu diesem Zweck müssten Spiele eingetragen werden können, mit künftigem Datum, Ort und Zeit, jedoch ohne Liste der Spieler, die teilgenommen haben, Torschützen und Resultat. Das wäre der use case „Datum, Ort und Zeit eines Spiels eintragen“. Zu diesem Use Case bestehen aber noch offene Fragen. Vor allem ist der Prozess der Spieleplanung und Doodle-Management abgekoppelt und wird von einer anderen Person erledigt, als der Prozess des Statistiken notierens. D.h. man vermischt hier gewisse Dinge. Zweitens ist fraglich, ob in der Praxis es wirklich praktischer ist, wenn die Spieler auf einer Applikation den Spielplan konsultieren würden, statt direkt im doodle, was eigentlich sehr gut funktioniert. Könnte man evtl. das doodle direkt einbinden und so automatisch einen Eintrag für künftige Spiele generieren? Da diese Fragen in der Praxis noch nicht geklärt sind, wird dieser Use Case vorerst eher ausgeklammert. Jedoch könnte er bei der Programmierung der API dennoch im Hinterkopf behalten und falls möglich berücksichtigt werden.

Spieleintrag korrigieren

Nun kann es sein, dass ein Spiel eingetragen wird, bei dem ein Eintrag fehlerhaft ist. Zum Beispiel könnte bei einem Tor versehentlich der falsche Spieler als Assistgeber vermerkt sein, oder bei den teilnehmenden Spielern ist ein falscher oder fehlender Name auf der Liste. Daher wäre ein weiterer Use Case, den man vielleicht auch als Teil des Use Cases „Spiel eintragen“ modellieren könnte, „Spieleintrag korrigieren“.

Betrag Eismiete berechnen

Für die Ausübung des Eishockeysports braucht es Eis, es muss daher eine Eishalle gemietet werden, wofür Kosten anfallen. Diese belaufen sich auf rund 600 Franken pro Spiel und werden unter den beiden Mannschaften aufgeteilt. Beim HC Keile zahlt jeder Spieler seinen Anteil nur an den Spielen, an denen er teilgenommen hat.

Das Geld wird nicht direkt bar an den Spielen eingezogen, sondern am Ende der Saison mit einer für jeden Spieler individuellen Rechnung vom Kassier eingezogen. Dafür muss die Kassier für jeden Spieler wissen, wie viele Spiele er in der abgelaufenen Saison absolviert hat. Bisher schaut er dies in den Doodle-Einträgen nach, was relativ umständlich ist.

2.3. Endpoints

Ein Endpoint ist die URI, also die Adresse, über die eine Ressource aufgerufen wird (Quelle: Traversy Media, YouTube Channel), verbunden mit einer entsprechenden HTTP-Abfrage (z.B. GET, PUT, POST), zwei HTTP-Requests können denselben Endpoint haben, aber verschiedene HTTP-Methoden (z.B. GET und POST). Um einen Endpoint zu definieren, muss man die Methode (z.B. GET) spezifizieren, sowie den Endpoint, d.h. die URL auf die die Abfrage angewendet wird. Weiter ist zu definieren, welche Daten genau in welchem Format vom Request zurückgegeben werden. Aus diesen Anforderungen lassen sich folgende Endpoints ableiten:

- Ressource: Player
 - GET \players
 - GET \players\{player_id}
 - POST \players
 - PUT \players\{player_id}
 - DELETE \players\{player_id}
- Ressource: Game
 - GET \games
 - GET \games\{game_id}
 - POST \games
 - PUT \games\{game_id}
 - DELETE \games\{game_id}
- Ressource: Goal

- GET \goals
- POST \goals
- PUT \goals\{goal_id}
- DELETE \goals\{goal_id}
- Ressource: Opponent
 - GET \opponents
 - GET \opponents\{opponent_id}
 - POST \opponents
 - PUT \opponents\{opopnent_id}
 - DELETE \opponents\{opponent_id}

3

Spring Framework, ORM und REST

3.1. Spring Framework	16
3.1.1. Dependency Injection	18
3.1.2. Aspect Oriented Programming	18
3.1.3. Spring Framework Kern	19
3.1.4. Spring Boot	19
3.1.5. Spring Projekte	20
3.2. Objekt/Relationales Mapping (ORM)	22
3.2.1. Object/relational Mismatch	22
3.2.2. Java Persistence API (JPA) und Hibernate	23
3.3. RESTful API	23

3.1. Spring Framework

Das open source Java Framework Spring bietet eine Implementation formalisierter best practices der Java Softwarearchitektur im Business-Kontext. [4]. Die erste Version des Frameworks wurde vom australischen Programmierer und Musikwissenschaftler Rod Johnson im Jahr 2002 im Buch „Expert One-on-One J2EE Design and Development“ [3] entwickelt. Johnson versucht darin, eine Antwort zu finden auf die Komplexität der Entwicklung von Unternehmens-Anwendungen, an die anspruchsvolle Anforderungen wie komplexen, datenintensiven Datenbanktransaktionen, Sicherheits- und Authentifizierung, Zugang übers Web oder Monitoring der Performance und Sicherheit gestellt werden.[8]

Er suchte nach einem Weg, solche Businessapplikationen auf möglichst einfache und schnelle Weise programmieren zu können und diese flexibel, leicht wartbar und einfach erweiterbar zu halten [8]. Dies wird zu einem wichtigen Teil erreicht, indem Spring die Konzepte der Dependency Injection (DI) und des Aspect Oriented Programming (AOP) umsetzt [4]

Vorläufer: JavaBeans

In den Neunzigerjahren wurde von Sun Microsystems die JavaBeans-Spezifikation entwickelt [42]. Dabei handelt es sich um ein Softwarekomponentenmodell, das Entwicklungsrichtlinien enthält, welche die Wiederverwendung von einfachen Java-Objekten und ihre Zusammenstellung zu komplexeren Applikationen vereinfachen sollte [8, S.4]. Sie entstanden aus der Notwendigkeit heraus, Java-Klassen möglichst einfach zu instantziieren und über Remote Method Invocation (RMI) in verteilten Applikationen zu verwenden. Die Haupteigenschaften von JavaBeans sind ein öffentlicher, parameterloser Konstruktor, Serialisierbarkeit und öffentliche Zugriffsmethoden (Getters und Setters) [42]. Dies ermöglicht ein Management der dependencies innerhalb der Anwendung durch das Framework.

Dependencies

Als dependencies werden erstens Libraries, Dokumente, Teile anderer Programme oder derselben Applikation bezeichnet, die ein Programm braucht, um ausgeführt werden zu können. Built tools helfen, die Links zu den nötigen Ressourcen herzustellen und diese verfügbar und aktuell zu halten. Zweitens werden Objekte als dependencies anderer Objekte bezeichnet, die Teil der Klasse des anderen Objekts sind. Z.B. der Typ eines Feldes oder des Parameters einer Methode. Das Objekt hängt ab somit ab von anderen Objekten.

Von Enterprise JavaBeans zu Spring

JavaBeans wurden ursprünglich für die Entwicklung von Benutzeroberflächen-Widgets entwickelt. Für Businessapplikationen, die aufgrund von z.B. Sicherheits-, Persistenz- oder Monitoringfunktionalitäten höhere Anforderungen stellen, war es nicht üblich, mit dieser Art von Programmierung zu arbeiten [8].

Ein erster Versuch JavaBeans im Businesskontext einzusetzen war die Enterprise Java Beans (EJB)-Spezifikation. Damit sollten komplexere Anwendungen für Bedürfnisse von Unternehmen mit Einsatz der flexiblen und leichtgewichtigen JavaBeans erstellt werden. Die Kombination dieser Konzepte in EJB war jedoch zu praxisfern umgesetzt, weshalb die Entwicklung schwerfällig blieb und sich die Entwickler davon abwandten. Der Grund war vor allem, weil die Klassen, die die Businesslogik implementierten, zu sehr mit externem JavaBeans-Code „verunreinigt“ wurden [8, S.4].

Parallel dazu wurde das Spring Framework entwickelt, mit dem gleichen Ziel, JavaBeans für Businessapplikationen nutzbar zu machen [8]. Der Trend der Java Entwicklung ging mit Spring weg von den komplexen EJB-Objekten hin zu simplen Plain Old Java Object (POJO)'s und dazu, diese auch für beispielsweise Persistenz-Funktionalitäten zu verwenden. Dies jedoch, ohne, dass sie davon wissen, respektive ohne, dass der Code dieser Klassen mit Code von diesen Querschnitts-Funktionalitäten wie Persistenz oder Sicherheit verunreinigt wurde. Dies war vorher noch nicht möglich.

Spring ist das bekannteste und am weitesten entwickelte Framework für die Entwicklung von Businessapplikationen mit JavaBeans. [8, S.4]. Der Begriff „Bean“ oder „JavaBean“ wird dabei nicht vollkommen strikt verwendet. Er bezeichnet oft einfach eine Anwendungskomponente, eine Instanz einer Klasse, ohne in jedem Fall der JavaBeans-Spezifikation [2] zu 100% zu folgen. [8, S.5].

3.1.1. Dependency Injection

Eine der wichtigsten Funktionen des Spring Frameworks ist es, „loose coupling“, statt „tight coupling“ zwischen den Objekten zu ermöglichen, diese zentrale Funktion wird mittels sogenannter „dependency injecton“ umgesetzt.

In einer grösseren Applikation wird eine grosse Anzahl Klassen instantiiert. Diese Objekte hängen von anderen Objekten ab, die sie als Felder oder für Ihre Methoden benötigen, ihre „dependencies“. Das Instantzieren von Objekten und deren Übergabe an eine andere Klasse wird im Spring-Umfeld „wireing“ genannt. Die Instanziierung und das wireing wird dabei durch das Framework via Konfigurationsinformationen umgesetzt. Die Konfiguration teilt dem Framework mit, welche JavaBeans instantziiert und mit welchen anderen Objekten sie verknüpft werden müssen. [26] Damit kann die Applikation besser gewartet, erweitert, modifiziert und migriert werden [35].

Container Interface: Application Context

Die Objekte werden also nicht im Quellcode mittels des „new“-Operators durch die Programmierer*in instantziiert, sondern das Framework instantziiert die Objekte zur Laufzeit. Dies wird durch einen Container gemanagt. Dieser liest die Konfigurations-Metadaten und „injiziert“ die Instanzen und ihre Dependencies in die Beans, die erstellt. Das Programm wird so konfigurierbar, ohne den Code zu verändern. Der Container in Spring ist eine Implementation eines Interfaces mit dem Name „Application Context“. [24]

Die Metadaten können in xml, als Java Annotationen oder als Java Code selber definiert sein[34]. Eine sogenannte Bean Factory, die dem Factory Design-Pattern folgt, erstellt die Instanzen und übergibt sie an die aufrufenden Klassen.[34]

3.1.2. Aspect Oriented Programming

Nebst der DI ist die aspektorientierte Programmierung ein zweites Kernfeature von Spring [8]. Aspects sind nichts anderes als spezifische Java Klassen, die Anforderungen, welche über verschiedene Schichten und Klassen der eigentlichen Applikation benötigt werden (sogenannte „cross cutting concerns“) enkapsulieren und implementieren. Dies sind typischerweise Funktionalitäten wie Sicherheit, Authentifizierung, Transaktionen oder Metrics.

Aspect Oriented Programming verfolgt das Prinzip, diese losgelöst von der Funktionalität der anderen Programmteile, d.h. der Business Logik, zu implementieren. Einzelne Klassen kümmern sich nur um ihre Kernfunktion und enthalten keinen Code von anderen Aufgaben, wie z.B. Datenbanktransaktions- oder Sicherheitsfunktionalitäten. Sie werden flexibel mit den Klassen der Business-Logik verschaltet. Dadurch erspart man sich, dass in jeder Klasse einer Applikation sich wiederholender Code von nicht zu der Klasse gehörender Funktionalität enthalten ist. Dies würde zu unwartbaren und viel Boilerplate-Code (der gleiche Code, der in vielen Klassen wieder und wieder geschrieben werden muss) enthaltenden Klassen und Applikationen führen [8, S.5].

3.1.3. Spring Framework Kern

Basis von allem ist der Spring Framework Core. Es stellt den Container zur Verfügung, in dem die Applikation gebildet wird und der für die Erstellung der Beans und die dependency injection zuständig ist [9, S. 26]. Dazu enthält es weitere essentielle Komponenten wie das Spring Model-View-Controller (MVC) Framework, welches das Hauptframework für Webapplikationen ist, Support für Persistenzfunktionalitäten mit Java Database Connectivity (JDBC) oder eine Testumgebung.[9, S.26].

3.1.4. Spring Boot

Spring Boot ist mittlerweile essenzieller Bestandteil der Entwicklung mit Spring. Es ermöglicht eine sehr schnelle und einfache Entwicklung von Applikationen, indem es eine Autokonfiguration der Applikationen zur Verfügung stellt[9]. Spring Boot wird im Abschnitt 3.3. näher eingegangen.

Während das Spring-Ökosystem eine breite Palette von Frameworks für Business-Applikationen zur Verfügung stellt, dient Spring Boot dazu, mit möglichst wenig Programmier- und Konfigurationsaufwand und in möglichst kurzer Zeit eine komplette, einsatzfähige Applikation, insbesondere von Microservices erstellen zu können [26][17]. Dies wird erreicht, indem sich Spring Boot um die Konfiguration der Applikation kümmert, gemäss best Practices und Konventionen, die sich in der Industrie etabliert haben. Es setzt damit den Grundsatz „Convention over Configuration“um. [39]

Autokonfiguration

Die Autokonfiguration der Applikation ist die Hauptfunktion von Spring Boot. Es liest die Klassen und JAR-files im classpath und erstellt basierend darauf eine automatische konfiguration der Applikation.[22] Wenn z.B. Hibernate im classpath ist sowie eine spezifische Datenbank, konfiguriert Spring Boot Hibernate automatisch für diese spezifische Datenbank. Gleichzeitig ist es auch möglich, auf einfache Weise manuelle Änderungen an der Konfiguration vorzunehmen. Nebst der Autokonfiguration sind die Überwachung der Applikation (Metrics und health checks), das Testing bei der Entwicklung und der eingebettete Tomcat-Server wesentliche Elemente einer Spring Boot-Applikation

Metrics, Testing und Spezifikation der Umgebung

Die Library für Metrics heisst Spring Boot Actuator. Damit kann man z.B. prüfen, ob ein Service erreichbar ist, wie oft er aufgerufen wurde, wie oft ein Aufruf fehlgeschlagen ist etc.. Weiter stellt Spring Boot zusätzliche Unterstützung für das Testen der Applikation zur Verfügung im Vergleich zum Spring-core Framework. Dazu ermöglicht Spring Boot das flexible Spezifizieren von Umgebungseigenschaften der Applikation [9].

Spring Boot generiert dabei keinen Code und ist selber auch kein Applikations- oder Webserver [26]. Es soll es der Programmier*in ermöglichen, sich auf die Logik und die Funktionalität der spezifischen Applikation zu konzentrieren, und möglichst wenig Code für das Framework oder für die Konfiguration schreiben zu müssen [9, S.26].

Integrierter Tomcat Server

Spring Boot versucht, alles, was man für eine vollständig funktionierende Applikation braucht, mitzubringen [9, S. 22]. So ist ein Tomcat Server ein Teil einer Spring Boot Applikation und wird automatisch für diese konfiguriert.

Das Konzept der eingebetteten Servers ist, dass der Server im JAR-file der Applikation mitliefert wird. Er läuft somit in einem Container. Damit erspart man sich das Installieren und Verbinden der Applikation mit einem externen Server, der auf der entsprechenden Maschine läuft. Für Microservices, die über ein Netzwerk verknüpft sind, bietet das eine grosse Vereinfachung. [26]

3.1.5. Spring Projekte

Auf Basis des Kerns des Spring Frameworks existieren eine grossen Anzahl Projekte, die für die eine grosse Vielfalt von Funktionalitäten, Anwendungsfällen und Applikationsarten Libraries und Frameworks zur Verfügung stellen. Beispielsweise für Messaging, Unterstützung für alle möglichen Arten von Datenbanken, Webapplikationen, Sicherheitsfunktionalitäten, Cloudapplikationen, Androidapplikationen etc.. Die Liste ist beliebig erweiterbar. Einen Überblick erhält man auf der Webseite des Spring-Frameworks, <https://spring.io/projects/spring-framework>.

Diese Frameworks und Libraries können online mit dem Spring Initializr, <https://start.spring.io/> - der Spring Boot verwendet - sehr einfach durch Anwählen von Checkboxes je nach Bedarf heruntergeladen werden. Somit erhält man ein bereits vorkonfiguriertes Applikationsgerüst, mit dem man in sehr kurzer Zeit und mit sehr wenig Code eine lauffähige Applikation erstellen kann. Spring Boot verwendet dabei eine aus der Sicht seiner Entwickler sinnvolle Standardkonfiguration, nach dem Prinzip „Convention over Configuration“. Spring ist in diesem Sinne nicht nur ein Framework, sondern ein Framework of Frameworks.[8]

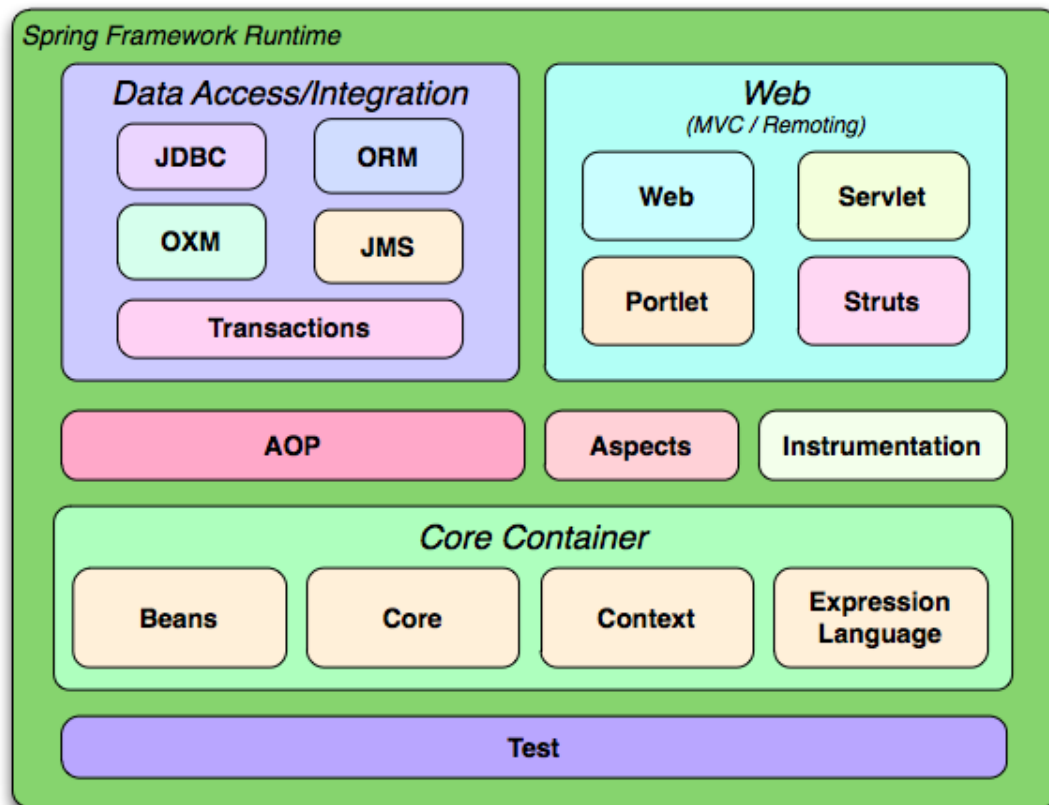


Abbildung 3.1.: Spring Framework Core

Im folgenden eine Übersicht ausgewählter, elementarer Teile von Spring und ausgewählter Spring Projekte, die aber in keinsten Weise eine vollständige Auflistung der grossen Anzahl und Vielfalt Spring-basierter Frameworks darstellt.

Spring Data

Über die Basisunterstützung für Datenbankkonnektivität des Framework Cores hinaus bieten die Datenbank-Module von Spring Data Unterstützung für die Verbindung mit nahezu allen gängigen Datenbanktechnologien. Folgende Datenbankmodule stehen beispielsweise nebst anderen zur Verfügung [9][S. 75] :

- Spring Data JPA: Verbindung mit relationalen Datenbanken.
- Spring Data MongoDB: Verbindung mit einer Mongo document database.
- Spring Data Neo4j: Verbindung mit einer Neo4j Graph-Datenbank.
- Spring Data Redis: Verbindung zu einem Redis key-value store.
- Spring Data Cassandra: Verbindung zu einer Cassandra Datenbank.

Dazu bietet Spring eine sehr einfache Art, diese Datenbankkonnektivität zu implementieren, indem einzig ein simples Java-Interface definiert werden muss [9]. Spring Data JPA liess sich unter anderem auch von Hibernate inspirieren. JPA mit Spring ist eine der verbreitetsten Methoden zur Persistenzimplementation mit Java. [8, S. 148]

Spring Security

Das Spring Security Framework ist ein umfassendes Framework, um Sicherheitsfunktionalitäten zu implementieren. Damit können beispielsweise Authentifizierung, Autorisierung oder API-Sicherungsfunktionen implementiert werden. w[9].

Spring Integration und Spring Batch

[9].

Spring Cloud

[9].

3.2. ORM

3.2.1. Object/relational Mismatch

Klassen in Java entsprechen nicht eins zu eins den Tabellen in relationalen Datenbanken. Dieses Problem wird auch „object/relational mismatch“ genannt. Es gibt mehrere Arten von nicht-Übereinstimmung zwischen den beiden Konzepten. ORM ist eine Lösung für dieses Problem, die aus der Praxis über Jahre hinweg entwickelt wurde [1]. Hibernate ist eines der ersten und bekanntesten ORM-Frameworks. Bei früheren Datenbank-Mapping-Tools oder Libraries wie JDBC musste man aufwändigen Code schreiben, die Queries von Hand ausformulieren und die Ergebnisse von Hand weiterverarbeiten. Mit Hibernate und ORM muss man Queries nicht mehr selber schreiben. Das Problem beim Queries selber Schreiben ist, dass wenn die Datenbank verändert wird, auch die Queries angepasst werden müssen. Das kann eine sehr aufwändige Aufgabe sein, wenn man viele und grosse Queries für eine grosse Applikation verwalten muss. [22]. Mit Hibernate wird diese Aufgabe automatisiert.

JDBC und Spring JDBC

Java Database Connection (JDBC) ist die Library in Java welche Abfragen auf Datenbanken ermöglicht. Dabei müssen jeweils die Verbindungen und das Trennen von Verbindungen zur Datenbank gemanagt werden, sowie das SQL-Statement, das Resultat muss von Hand iteriert und die Ergebnisse einzeln ausgelesen werden, diese müssen gespeichert und zurückgegeben werden und alle möglichen Arten von Fehlern müssen behandelt werden. Kurz: es braucht viel Code und es ist aufwändig, sich um alles kümmern zu müssen.

Spring JDBC vereinfacht diesen Prozess merklich, in dem es Automatisierungen vieler dieser Schritte zur Verfügung stellt und Programmierung auf abstrahierter Ebene ermöglicht, die viel weniger Code und Aufwand benötigt. [26]

Objekt/relationales Mapping

Deutlich mehr Features und leistungsfähiger als JDBC und Spring JDBC sind Tools und Frameworks für objekt/relationales Mapping. Vor allem, wenn Anwendungen komplexer werden, reichen JDBC-Tools oft nicht mehr aus. [8, S. 34]

Mit objekt/relationalem Mapping kann der Persistenz-Aspekt automatisiert und abgetrennt von der Business-Logik bewältigt werden. JPA und Hibernate sind bekannte Open Source Frameworks für ORM. Sie definieren Sets von Annotationen und Interfaces, mit denen das Framework automatisch die Queries generiert und die Tabellen in der Datenbank erstellt.[26]

3.2.2. JPA und Hibernate

JPA ist das Persistenzmodul von Spring, das den Support für die Verbindung zu und das Ablegen, Lesen und Ändern von Daten in Datenbanken und der Modifikation von Datenbanken selber bereitstellt. Das Interface `JpaRepository` stellt eine von mehreren Abstraktionen sowie Standardimplementationen für die Basis-Datenbankoperationen, CRUD-Operationen (Create, Retrieve/Read, Update, Delete/Destroy), zur Verfügung. Das verbreitetste ORM-Framework, Hibernate, implementiert JPA, bietet aber noch viel mehr Funktionalitäten [22].

Um ORM umzusetzen muss man angeben, wie die als normale Java-Objekte POJO's implementierten Entitäten zueinander in Verbindung stehen. Wenn die entsprechenden Annotationen in den Klassen gesetzt sind, kann Hibernate diese lesen, interpretieren und die entsprechenden Queries in den Datenbanken vornehmen [26]. Durch wenige Annotationen in den Java-Entity-Klassen kann Hibernate all diese Datenbankfunktionalitäten automatisch generieren. [21]

Transparent persistence

Hibernate implementiert das Konzept der „transparent persistence“. Das heisst, dass die Klasse, die in der Datenbank gespeichert wird, nichts davon wissen muss. Sie wird mit keinem Code für Datenbank-Abspeicherung verunreinigt und muss keine Interfaces implementieren oder Klassen erweitern. Das Konzept der „separation of concerns“, d.h. dass sich die Business-Logik der Applikation nicht um die Persistenz kümmern muss und umgekehrt, wird damit realisiert.

3.3. RESTful API

REST API's sind ein leichtgewichtiger und weborientierter Ansatz für Services. [Pasquier, 2019]. Es ist der Ansatz der in den letzten Jahren am populärsten geworden ist und SOAP den Rang abgelaufen hat. Es handelt sich bei REST (Representational State Transfer) um ein Set von Richtlinien und Prinzipien (somit um ein Entwurfsmuster oder ein Architekturstil) für die Programmierung von Schnittstellen (Interfaces) zwischen verteilten Systemen und verteilten Applikationen. [Tilkov et al., 2015, S. ix]

REST basiert auf HTTP. Die allgemeinen Ziele für verteilte Systeme, die REST zu erreichen hilft, sind loose Kopplung, Interoperabilität, Wiederverwendung, Performance und

Skalierbarkeit [7]. REST-Abfragen sind schneller und datenärmer als SOAP-Abfragen, da ein weniger grosser Overhead mitgeschickt wird als beim zuvor am weitesten verbreiteten Standard „Simple Object Access Protocol (SOAP)“. Im Unterschied zu SOAP-Webservices funktioniert die Abfrage bei REST-API's nur über die URL.

REST Prinzipien Nutzung von Hypermedia ..

4

Schema der Datenbank

4.1. Entity Relationship Model (ERM)	25
4.2. Verbale Beschreibung	26
4.3. Tabellen	27
4.4. Domain Model vs. ERM	28

Die Nutzenden des Webservices sollen einen Spieler auswählen und Informationen erhalten zu: Wie viele Spiele er gespielt, wie viele Tore er erzielt und wie viele erste und zweite Assists er produziert hat. Zweitens sollen die User ein Spiel auswählen und Daten zum Resultat, welche Spieler gespielt haben, wie viele Tore und Assists jeder erzielt hat, erhalten.

4.1. ERM

Die Daten können in folgendem Entitäten-Beziehungsmodell (ERM) dargestellt werden:

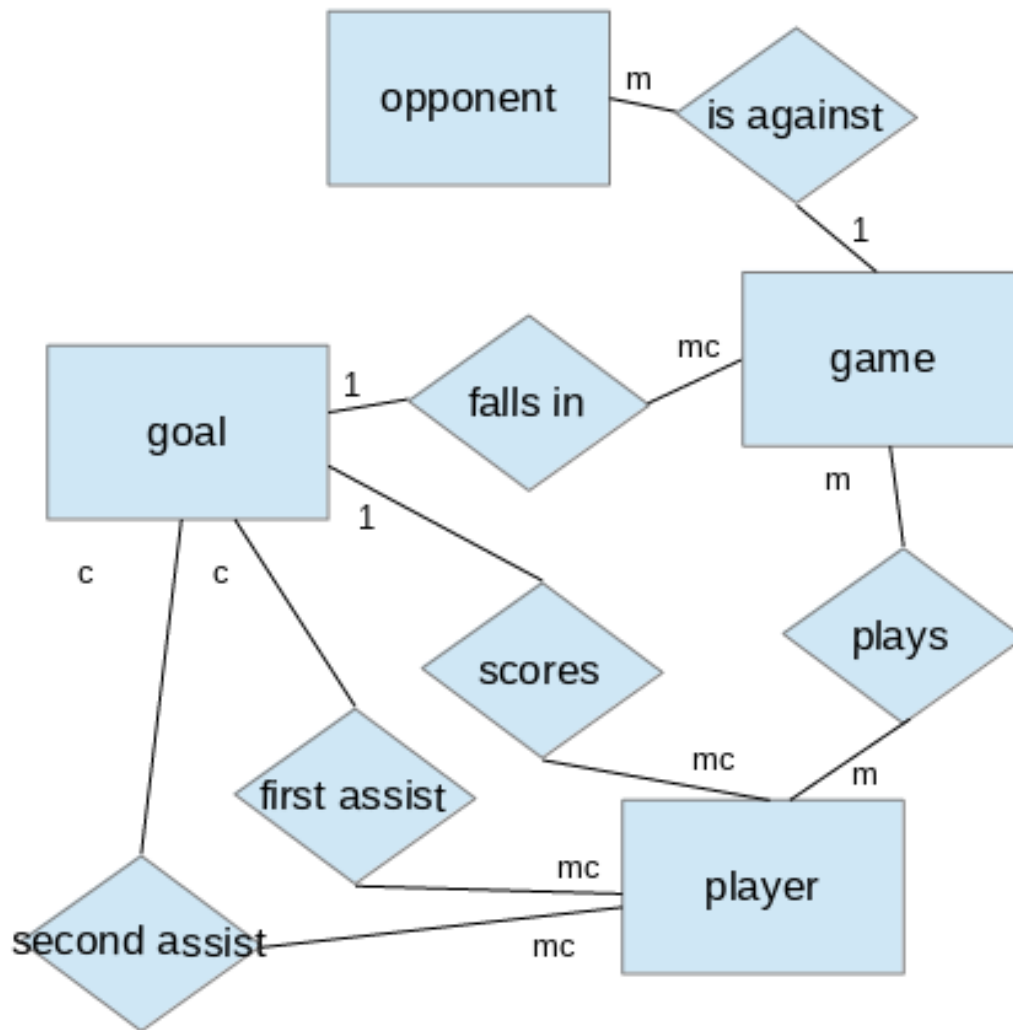


Abbildung 4.1.: Entitäten-Beziehungsmodell

4.2. Verbale Beschreibung

Mehrere Spieler (player) spielen ein Spiel (game). Ein Spiel hat keins, eins oder mehrere Tore des HC Keile (goal). Einem Spiel ist genau eine gegnerische Mannschaft zugeordnet. Diese kann dies in einem oder mehreren Spielen sein.

Ein Spieler spielt eins oder mehrere Spiele. Er kann keine, eins oder mehrere Tore, erste (first assist) oder zweite (second assist) Assists erzielt haben. Die Assists sind als Attribut einem bestimmten Tor zugeordnet, das wiederum einem Spiel zugeordnet ist. Im Fall des Torhüters ist für jedes Spiel vermerkt, ob er keins, eins oder mehrere Gegentore erhalten hat. Eigentore gibt es im Eishockey nicht, wenn eine Mannschaft den Puck ins eigene Tor manövriert, gilt der Spieler der gegnerischen Mannschaft, der den Puck als letzter berührt hat, als Torschütze.

Ein Tor wird genau einem Spiel zugeordnet. Es enthält neben der Spieler-ID des Torhüters die ID des ersten und zweiten Assists als Attribute. Ein Tor kann keins, eins oder zwei Assists haben. Jedes erste Assist und jedes zweite Assist gehören zu genau einem Tor. Jedes Tor, jedes erste Assist und jedes zweite Assist sind genau einem

Spieler zugeordnet. Zwischen den Entitäten “player” und “game” besteht die komplex-komplexe Beziehung “plays” (Spieler spielt Spiel). Diese Relation wird in einer eigenen Tabelle abgebildet.

4.3. Tabellen

Daraus ergeben sich die folgenden Tabellen: Je eine Tabelle pro Entität: “game”, “goals”, “opponent” und “player”, dazu eine Tabelle für die komplex-komplexe Beziehung “player plays game”. Die restlichen Relationen werden mittels eines Fremdschlüssels in den erwähnten Tabellen modelliert.

Game

attribute	format	primary key	foreign key	not null
game_id	INT	✓		✓
game_date	DATE			✓
game_status	CHAR			
goals_opponent	INT			✓
goals_keile	INT			
opponent_id	INT		✓	

Tabelle 4.1.: Tabelle Game

Goal

attribute	format	primary key	foreign key	not null
goal_id	INT	✓		✓
game_id	INT		✓	✓
scorer_id	INT		✓	✓
assist1_id	INT		✓	
assist2_id	INT		✓	

Tabelle 4.2.: Tabelle Goal

Player				
attribute	format	primary key	foreign key	not null
player_id	INT	✓		✓
firstname	CHAR			✓
lastname	CHAR			
date_of_birth	DATE			
address	CHAR			
phone	CHAR			
email	CHAR			
position	CHAR			

Tabelle 4.3.: Tabelle Player

Opponent				
attribute	format	primary key	foreign key	not null
opponent_id	INT			✓
name	CHAR			

Tabelle 4.4.: Tabelle Opponent

Games played				
attribute	format	primary key	foreign key	not null
player_id	INT	✓	✓	✓
game_id	INT	✓	✓	✓

Tabelle 4.5.: Tabelle Games played

4.4. Domain Model vs. ERM

Die Literatur zu Hibernate [1, S. 64] beschreibt als Ausgangspunkt für die Softwareentwicklung mit Hibernate das Erstellen eines Domain Models. Im Buch von Bauer und King [1, S. 64] wird das Domain Model als abstrakte, formalisierte Repräsentation der realen Entitäten, welche die Software abbilden und unterstützen soll, sowie deren logischen Beziehungen untereinander dargestellt. Daraus werden in der objektorientierten Programmierung die Klassen und Unterklassen sowie ihre Verbindungen im Programm abgeleitet.

Das Entitäten-Beziehungsmodell hat eine ähnliche Herangehensweise und Funktion. Es stellt ebenfalls die realen Entitäten und ihre logischen Beziehungen dar, welche formalisiert und abstrahiert dargestellt werden. Es wird jedoch verwendet, um daraus die Struktur der relationalen Datenbank abzuleiten und zu normalisieren (siehe Kapitel 3). In

einem Domain Model können noch mehr Arten von Verbindungen und Funktionalitäten dargestellt werden, als im Entitäten-Beziehungsmodell.[41]

Für diese Arbeit hier würden höchstwahrscheinlich nicht riesige Unterschiede zwischen einem Domain Model und einem Entitäten-Beziehungsmodell bestehen. Bauer und King erwähnen auch, dass Hibernate nicht nur Domain Models aus, sondern auch von Tabellen oder einem anderen Model ausgehend eingesetzt werden kann. [1, S. 64]. Auch wenn bei grösseren Applikationen das Domain Model wohl die bessere Wahl ist, da Hibernate hilft, Objekte in Relationen zu übersetzen und nicht umgekehrt. Daher wird hier als Ausgangspunkt für die Programmierung das Entitäten-Beziehungsmodell wie in Kapitel 3 dargestellt, verwendet.

5

Programmierung und Dokumentation der API

5.1. Installation von Spring	30
5.1.1. Spring Initializr	30
5.2. Quellcode	31
5.2.1. Controller	31
5.2.2. Repositories	32
5.2.3. Entities	34
5.2.4. Main-Klasse	35
5.3. Persistence Layer	35
5.3.1. JPA und Hibernate	35
5.3.2. Database Layer	37
5.4. Dokumentation mit Swagger	38

5.1. Installation von Spring

5.1.1. Spring Initializr

Ein Spring-Projekt kann sehr einfach und schnell mit Hilfe des Onlinetools Spring Initializr auf <https://start.spring.io/> erstellt werden.

Auf dieser Webseite wählt man zuerst das gewünschte Built-Tool (Maven oder Gradle) aus, dann die Programmiersprache (Java, Kotlin oder Groovy), dann die Version von Spring Boot - es wird also bereits davon ausgegangen, dass man Spring Boot nutzt - und definiert einen Projektnamen und einen Namen für das Paket für das Projekt. Dann kann man unten bei den Dependencies die gewünschten Libraries und Frameworks, die man für sein Projekt braucht, anwählen.

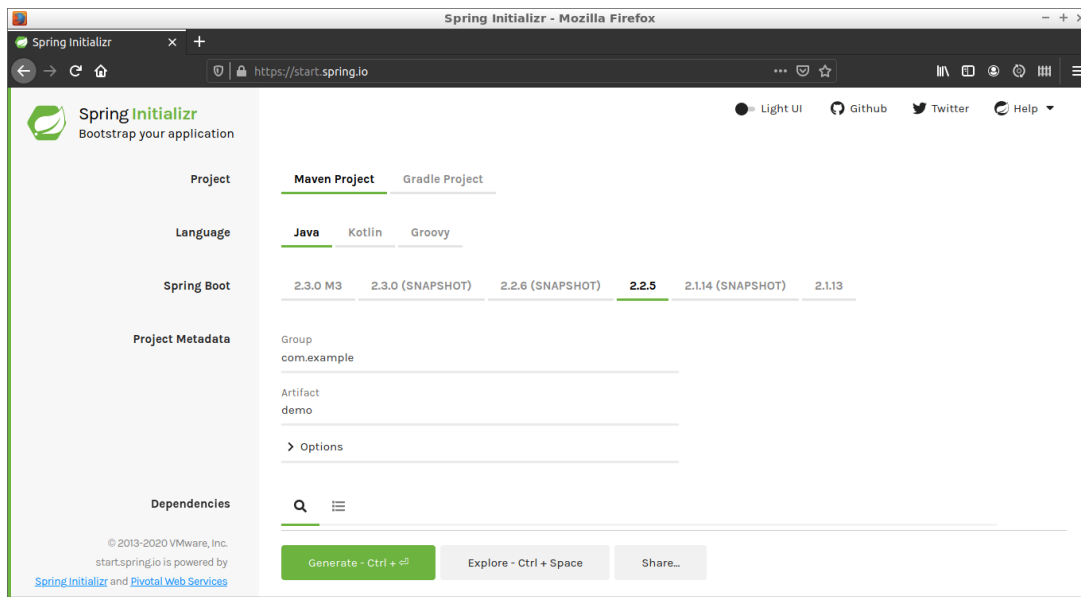


Abbildung 5.1.: Spring Initializr Onlinetool

Das Resultat lädt man als Jar-File und Maven- respektive Gradleprojekt auf seinen Computer und importiert es in seine IDE. Dort hat man nun eine fertig eingerichtete Projektstruktur zur Verfügung, in welcher bereits die Main-Klasse erstellt ist.

In dem man z.b. das starter web-Paket als Dependency im Spring Initializer wählt, erhält man das Spring MVC Framework, ein Logging Framework, das Spring Core-Framework, sowie ein Validierungsframework in die Applikation integriert. Ähnlich ist es mit Spring Boot starter-JPA. Damit erhält man JPA, mit Hibernate eine Standardimplementation von JPA sowie eine automatische Konfiguration dieser Teile der Applikation. [26]

Wenn die Applikation startet, sorgt die Spring Boot Autokonfiguration dafür, dass die Beans im Spring application context (dem Container) erstellt und konfiguriert werden, und somit Spring MVC genutzt werden kann, sowie dass der eingebettete Tomcat-Server konfiguriert und gestartet wird [9, S.26]

Projektstruktur in Eclipse

Die Ordnerstruktur in der IDE ist folgendermassen aufgebaut. Ein Package beinhaltet die Klassen, die Entitäten, die in der Datenbank abgespeichert werden, ein Package beinhaltet die Repository-Interfaces für diese Entitäten und das dritte Package enthält die Controller, welche die Requests an die Endpoints verarbeiten. Auf derselben Ebene wie die Packages ist die Main-Klasse, welche das Programm startet, die Konfiguration liest und die Beans kreiert

5.2. Quellcode

5.2.1. Controller

Im MVC-Schema ist die Funktion des Controllers, dass er HTTP-Requests die auf eine spezifische Domain geschickt werden, pro Domain abarbeitet und implementiert, wie die

einzelnen Requests verarbeitet werden sollen.

Der Unterschied eines Controllers, der mit der `@RestController`-Annotation annotiert ist, zu einem traditionellen Spring MVC Controller ist, dass anstatt, dass eine html-Seite gespiesen wird mit den Resultaten der durch den Request aktivierten Methode, nur ein Objekt generiert wird, dessen Daten direkt via http-Response, dem client zurückgegeben werden. [34] Die `@RestController` Annotation ist eine Abkürzung für `@Controller` und `@ResponseBody` und markiert die Klasse als Klasse, bei der jede methode ein Objekt zurückgibt anstatt eine view (html). [34] Sie gibt Spring ebenfalls an, dass der Rückgabewert der Methode direkt in den Response Body der Response gespeist werden soll, anstatt in ein Model für eine View.[9, S. 142]

@RequestBody Annotation

Die `@RequestBody`-Annotation sagt Spring, dass die Daten aus dem Body des Http-Request in das Objekt, das dem Controller übergeben wird, konvertiert werden sollen. Ohne diese Annotation geht Spring davon aus, dass es die Parameter des Http-Requests an das Objekt binden soll [9][S. 146]

HTTP-Request-Annotationen

Es ist gute Praxis, auf Klassenlevel bei den Controller-Klassen die „RequestMapping“-Annotation zu benutzen und dabei den basis URI-Pfad zu definieren, und dann bei den Methoden, welche bei den einzelnen Requests aufgerufen werden, so spezifisch wie möglich zu sein und zu spezifizieren, um welche Art von Dazu verwendet man die "Get/Post/-Put/DeleteMappingAnnotationen. [9, S. 35]

Die `@RestController`-Annotation hat zwei Hauptzwecke. Erstens wird mit ihr die Klasse als Component für das component scanning markiert. Zweitens sagt sie Spring, dass die Rückgabewerte aller handler-methoden der Klasse direkt in den Response-Body der HTTP-Response geschrieben werden sollte, anstatt dass sie in das Model gespiesen und eine View daraus generiert werden soll [9]

Wie den Controller für die Post-Methode umsetzen, wenn Informationen, die bereits in der Datenbank sind, z.B. ein Gegner, oder die Listen, auch eingegeben werden müssen? Idee: Klasse bauen, welche die vollständigen Daten inkl. Listen in Datenbank postet, (in repository-Methode). Davor die Kasse füllen mit Rückgaben aus entsprechenden Abfragen aus der Datenbank. Dann Zusammenfügen mit Infos/Klasse, die man im Request-Body, via JSON übergeben hat.

5.2.2. Repositories

Die Interfaces, mit denen Zugriff auf die Persistenzschicht sprich auf die Datenbank in Spring Boot programmiert werden kann, heissen „Repositories“(oder, eher veraltet, Data Access Objects (DAO's) genannt). Sie werden eingesetzt, um auf die sogenannte Hibernate-Session zugreifen zu können, in der die zu speichenden oder lesenden Daten und Objekte von Hibernate instantziiert, gemanaged, manipuliert und die Persistenz-operation durchgeführt werden. Dies mit Hilfe der in Repository-Interface und seinen Sub-Interfaces definierten Methoden.

In Hibernate 5.4 muss das Repository-Interface nicht mehr von der Programmier*in implementiert werden. Wenn die Dependency „spring-boot-starter-jpa“ in das Projekt integriert ist, reicht es, ein Repository-Interface zu definieren, welches beispielsweise das JpaRepository-Interface erweitert. Dieses hat die Form: JpaRepository<T,ID> [25]. Es muss lediglich der Objekt-Typ der Entität spezifiziert werden, der mit Hilfe dieses Repositories bearbeitet wird sowie der Typ des Identifikationsschlüssels der Entität. Sobald dieses Interface definiert ist, kann man es bereits für die entsprechenden Datenbankmanipulationen verwenden. Das „Repository“-Interface ist das Haupt-Markerinterface für Persistenzoperationen.

Es findet dabei im Hintergrund keine Codegenerierung oder Bytecode-Manipulation statt. Die Methodenaufrufe werden mit Hilfe einer MethodInterceptor-Klasse abgefangen. Sie schaut zuerst, ob es eine Implementierung der Methode im Quellcode gibt, falls nicht, versucht sie, die nötigen SQL-Abfragen sonst aus dem Code zu ermitteln, z.B. aus im Code explizit formulierten Queries oder über den Namen der Methode („save“, „find“etc.). Falls das nicht geht, greift die Klasse auf eine von Spring Data zur Verfügung gestellte Basisklasse zurück, die eine Standardimplementierung der Interface-Methode zur Verfügung stellt. [6, S. 221] [22]

Für das Repository-Interface, wie für viele andere Interfaces auch, bietet Spring bereits Standardimplementationen der Methoden, auf die es zurückgreift, wenn der Programmierer keine Implementation spezifiziert und wenn, wie oben beschrieben, andere Strategien des Frameworks, die nötigen Befehle abzuleiten, nicht möglich sind.

```
1 package ch.keilestats.api.application.repositories;
2
3 import org.springframework.data.jpa.repository.JpaRepository;
4 import org.springframework.stereotype.Repository;
5 import ch.keilestats.api.application.entities.Game;
6
7 @Repository
8 public interface GameRepository extends JpaRepository<Game, Long>{
9 }
```

Listing 5.1: JpaRepository Interface

Transaktionen: JPA Persistence Context respektive Hibernate Session

Der Persistence Context ist ein wichtiges Konzept von JPA. Er liefert die Umgebung, in der nötigen Objekte instanziiert und Daten und Operationen geladen werden, um die gewünschten Manipulationen an der Datenbank vorzunehmen. In Hibernate liefert das Session-Interface den Zugang zum Persistence Context, der in Hibernate „Session“heisst. Es muss in neueren Hibernate-Versionen nicht mehr explizit verwendet werden, sondern Hibernate nutzt es automatisch im Hintergrund.

Die Session wird am Anfang der Transaktion (normalerweise einem Methodenaufruf einer der Repository-Methoden) kreiert und nach dem Ende aller Datenbankoperationen wieder aufgelöst.

5.2.3. Entities

Aus den Enterprise JavaBeans Spezifikationen stammt das Konzept der Entity-Beans. Das sind Business-Objekte, die in einer relationalen Datenbank persistent gespeichert werden.

Mit der `@Entity` – Annotation von JPA, die bei einer Klasse angebracht wird, wird angegeben, dass die Klasse einer Tabelle in der Datenbank entspricht. Ihre Felder also den Tabellenspalten. Es braucht für den Primärschlüssel eine `@Id`-Annotation und die Spalten können auch explizit mit `@Column` angegeben werden (dabei kann auch der Name der Spalte in der Datenbank präzisiert werden, wenn er von Feldnamen abweicht), müssen aber nicht. Den Primärschlüssel lässt man idealerweise selber vom Framework generieren. Dies kann man tun mit der Annotation `@GeneratedValue`. Damit „mappt“ man die Entitäts-Klassen mit der Datenbank.

Es braucht zwingend auch einen leeren Konstruktor, auf den Hibernate zurückgreifen kann. Zudem kann man auch einen Konstruktor ohne den Primärschlüssel erstellen, somit kann man einen Eintrag in die Datenbank machen, und den Primärschlüssel vom Framework erstellen lassen. [22]

Bei einer `@OneToOne`-Annotation, wird, wenn man die Entität aus der Datenbank lädt („fetched“), ein join auf den mit als `@OneToOne` und als Fremdschlüssel markierte Tabelle gemacht, und auch alle Werte aus dieser Tabelle gelesen. Dies wird Eager Fetch genannt. [26]

Unidirektionale Beziehung mit der `@OneToOne`-Annotation wird eine Beziehung genannt, die in einer Klasse ein Feld mit einer einfach-einfach-Beziehung zu einer anderen Entität als Fremdschlüssel definiert, jedoch in der anderen Klasse nichts vermerkt. Somit kann man von dieser Klasse zwar die Werte, die verbunden sind, aufrufen. Von der anderen Klasse aus kann man aber die zugehörigen Daten aus der anderen Tabelle einzelner Einträge nicht abrufen. Damit das gemacht werden kann, muss die Beziehung bidirektional gemacht werden.

„Owning side of the relationship“: Mit der owning side ist diejenige Tabelle gemeint, welche die ID der verbundenen Entität als Fremdschlüssel enthält. Wenn man keine owning side definiert, werden in beiden Tabellen Fremdschlüssel der jeweils anderen Entität abgespeichert. Dies ist Duplikation der Information, was man nicht will in einer Datenbank. Dies kann gelöst werden, indem man in der Tabelle, in der man den Fremdschlüssel nicht eintragen will (auf der „non owning side“), im entsprechenden Feld bei der `@OneToOne`-Annotation, das Attribut `mappedBy`= Name des Felds des Fremdschlüssels in der anderen Tabelle anfügt. [26]

Java Persistence Query Language (JPQL) kann in JPA gebraucht werden, um Queries zu definieren. Es ist ähnlich wie SQL, aber die in den Queries adressierten Entitäten sind nicht die Tabellen in der Datenbank, sondern die Klassen im Java-Code, die mit Entity annotiert sind. Diese Queries werden dann von Hibernate in SQL-Queries umgewandelt, die mit der Datenbank interagieren. [26]

Man kann in Hibernate auch Native Queries, dh. Queries in SQL direkt auf der Datenbank durchführen. Dies kann mit der Entity Manager methode `createNativeQueries` getan werden. Ein Anwendungsfall ist z.B. wenn man viele Tabellen updaten möchte, dies ist mit JPQL nicht oder nur sehr umständlich möglich. Es ist dabei zu beachten, dass der Application Context (die Instanzen der Java Klassen, welche die Datenbank repräsentie-

ren) nicht upgedated werden. Es muss also ein `em.refresh()` durchgeführt werden, um die Werte aus der Datenbank in den Application Context zu laden. [26]

5.2.4. Main-Klasse

Die Main-Klasse der Applikation heisst „KeileStatsApplication“. Sie besteht aus wenig Code. Sie umfasst eine `main`-Methode, in welcher einzig die statische Methode „`SpringApplication.run(KeileStatsApplication.class, args)`“ aufgerufen wird. Mit dieser Methode wird die Applikation gebootstrapt. Die zweite wichtige Komponente ist die Annotation „`@SpringBootApplication`“. Das ist eine Composite-Annotation, die drei weitere Annotationen umfasst.

„`@SpringBootConfiguration`“, die eine spezialisierte Form der `@Configuration`-Annotation ist und angibt, dass in dieser Klasse javabasierte Konfigurationsinformationen enthält. [9, S. 15]. „`@EnableAutoConfiguration`“ ermöglicht Spring Boot die Applikation automatisch zu konfigurieren. `@ComponentScan` ermöglicht component scanning. Das heisst dass andere Klassen mit `@Component`-Annotationen annotiert werden können und als Komponenten der Applikation im Spring application context registriert werden können. [9, S. 15]

Die Annotation `@Component` einer Klasse sagt dem Framework, dass diese Klasse eine Bean ist, die instantiiert werden soll. Mit der `@SpringBootApplication`-Annotation wird Spring mitgeteilt, dass es im Package, in das sich die Klasse mit dieser Annotation befindet, durchsuchen soll danach, welche Beans zu kreieren sind, d.h. nach der `@Component`-Annotation, somit wird Spring also mitgeteilt wo es nach Beans und ihren Dependencies suchen soll. Der Application Context, der von der Methode `run` der Klasse `SpringBootApplication` zurückgegeben wird, managed das Implementieren und erstellen dieser Beans und die Injektion der dependencies. [26]

5.3. Persistence Layer

5.3.1. JPA und Hibernate

Hibernate Types

Hibernate hat seine eigenes Set von Datentypen, welches die Java-Datentypen in die SQL-Datentypen der Datenbank übersetzt. [1][S. 198]

Mit dem Attribut „`cascade: save-update`“ speichert Hibernate ein neu instantiiertes Objekt, das einem anderen Objekt, welches bereits in der Datenbank persistent gespeichert ist, zugeordnet wird, z.B. als Feld, automatisch, ohne dass „`save()`“ aufgerufen werden muss.

Hibernate implementiert einen sogenannten „`transaction-scoped object identity`“. Das heisst, dass garantiert ist, dass die Instanz des Objekts, das in der Datenbank gespeichert oder aus ihr gelesen wird, über eine Transaktion hinweg, ein und dieselbe Instanz ist. Nicht dass, im Gegensatz dazu, bei einer Applikation ohne „`identity scope`“, wenn innerhalb einer Transaktion zwei mal eine Zeile aus der Datenbank gelesen wird, nicht garantiert ist, dass im Arbeitsspeicher nur eine Instanz des Objekts erstellt wird [1]

Jedes Persistence-Tool wie Hibernate verfügt über eine „persistence manager“-API, mit auf die Datenbank zugegriffen, Einträge upgedated, gespeichert oder gelöscht werden können (CRUD-Operationen). Weiter können auch Abfragen auf die Datenbank erstellt werden, Transaktionen erstellt und gesteuert werden sowie der Cache auf Transaktionslevel gesteuert werden. Das Hauptinterface dafür bei Hibernate ist das „Session“-Interface. [1] Mit Spring Boot 2.2.0 und Hibernate 5.2 muss jedoch das Session-Interface nicht zwingend genutzt werden, es reicht, wenn man ein Repository-Interface erweitert.

Wenn die „save()“-Methode aufgerufen wird, wird nicht direkt ein „INSERT“-SQL-Statement aufgerufen. Das Objekt wird nur in der Session als persistent markiert. Es werden möglichst wenige Datenbank-Aufrufe getätigt. Erst wenn die Session abgeschlossen ist und sie „committed“ wird (in neueren Versionen automatisch im Hintergrund), wird eine Datenbankverbindung aufgebaut und die SQL-Statements ausgeführt [1].

„automatic dirty checking“ heisst, dass Hibernate wenn ein persistentes Objekt mit einem GET-Request aus der Datenbank in eine Session geladen wird, Hibernate automatisch Änderungen am Objekt verfolgt und merkt, wenn das Objekt „dirty“ ist, das heisst geändert wurde und nicht mehr mit der Datenbank übereinstimmt. Hibernate speichert die Änderungen automatisch, wenn die Session „committed“ wird [1][S. 129].

Mit dem „delete()“-Befehl wird ein persistentes Objekt zu einem transienten Objekt gemacht, d.h. es wird von der Datenbank getrennt. Wenn es von keinem anderen Objekt mehr in der Applikation referenziert wird, wird es mit dem Garbage-Collector gelöscht. Der SQL-DELETE-Befehl wird erst ganz am Schluss ausgeführt, wenn die Session mit der Datenbank synchronisiert wird am Ende der Transaktion [1][S. 130]

Ein „detached“-Objekt (d.h. ein Objekt, das eine Datenbank-Identität hat, aber das nicht mehr mit einer Session mit der Datenbank verbunden ist, d.h. bei Änderungen würde nicht mehr automatisch der Status in der Datenbank angepasst) [1][S. 130], kann transient gemacht werden, indem man das persistente Objekt aus der Datenbank löscht. Das detached Objekt muss dabei nicht mehr mit einer Session wieder verbunden werden, um es aus der Datenbank zu löschen und transient zu machen. [1][S. 130].

Hibernate ist ein Tool, das eine sogenannte „transitive persistence“ offeriert. Das bedeutet, dass die Persistenz-Operationen im Hintergrund unsichtbar ablaufen - jedenfalls für transiente und detached Objekte. Die Klassen der Applikation sind mit keinem Hibernate-Code, abgesehen von den Meta-Informationen, verunreinigt und es müssen nur ein zwei wenige generische JPA-Interfaces, eingesetzt werden um die gewünschten persistenten Objekte zu bearbeiten. Es muss fast kein Datenbankzugriffs-Code geschrieben werden, abgesehen von simplen CRUD-Methoden. [1][S. 131]

Hibernate implementiert ein sogenanntes „association-level cascade style“-model um dafür zu sorgen, dass assoziierte Objekte, dependencies, eines persistenten Objekts ebenfalls in der Datenbank gespeichert werden, insbesondere, wenn es sich um detached oder transiente Objekte handelt. Dabei wird die Art, wie Hibernate mit dem Objekt-Graph, der aus transienten, detacheden sowie persistenten Objekten besteht, umgeht, und die zu speichernden Objekte persistent macht, mittels den Meta-Informationen zu den Assoziationen zwischen den Objekten festgelegt. [1][S. 133]

Das Ziel ist dabei, den besten cascading-Stil für jede Assoziation festzulegen, damit die Anzahl Sessions, um Objekte in der Datenbank abzulegen, minimiert wird. [1][S. 153]

Das „cascade“-Attribut in Hibernate bezieht sich auf die Assoziation zwischen zwei Objekten (z.B. „many-to-one“, „one-to-many“ etc.). Wenn z.B. bei einer many-to-one Assoziation

eines Objekts zu einem anderen cascade = „none“eingestellt ist, wird bei einer Speicherung einer Änderung in der Datenbank dieses Objekts, nicht das Objekt auf der anderen Seite der Assoziation auch geladen und kontrolliert, ob es auch eine Änderung zu speichern gibt. [1][S. 136].

Damit Hibernate transiente von detachten Objekten unterscheiden kann, wird für das „id“-Attribut der Entität der java.lang.Long-type verwendet. Dadurch, dass die id automatisch von Hibernate generiert wird, ist bei Objekten, die neu erstellt werden, der id-typ noch null, bei solchen, die bereits persistent waren, nicht. [1]

Unter „fetching strategy“wird verstanden, welcher Teil, des Objektgraphen aus der Datenbank geladen werden soll [1][S. 140]

Eine der grössten Herausforderungen von ORM ist, effizienten Zugriff auf die relationale Datenbank zu gewährleisten (bei Applikationen, welche die Daten als Objektgraphen erhalten wollen). In grösseren, verteilten, multiuser, enterprise und web-Applikationen ist ein ineffizienter Datenzugriff nicht akzeptabel[1][S. 143]

Fetching-Strategien sind z.B. „eager fetching“. Das heisst, dass das assoziierte Objekt einer Entität, die aus der Datenbank geladen wird, wird dann geladen, wenn die Entität geladen wird, mit einem outer join. Es wird nur eine Abfrage auf die Datenbank gemacht. „lazy fetching“: Das assoziierte Objekt wird nicht aus der Datenbank geladen, respektive erst, wenn darauf das erste Mal zugegriffen wird. Es resultiert eine neue Abfrage auf die Datenbank. Es ist eine guter Default-Wert, um eine anständige Performance zu erhalten. „batch fetching“bedeutet, dass alle in der Session enthaltenen Objekte geladen werden. Das ist eine gute Strategie für unerfahrene Nutzer, die nicht zu genau über die genauen SQL-Abfragen nachdenken möchten, aber trotzdem anständige Performance erreichen wollen. [1][S. 144].

Für Collections wird als Default-Wert für das fetching „lazy “empfohlen. Eager fetching eignet sich nicht für Kollektionen [1][S. 148]

5.3.2. Database Layer

Mit dem Spring Framework und Spring Boot 2, respektive der Java Persistence API und ihrer verbreitetsten Implementation Hibernate ist es ganz einfach, eine breite Palette von Datenbank- Technologien zu verwenden. Sie unterstützen die meisten Datenbanken. Man braucht einzig die gewünschten dependencies im der Maven Project Object Model Konfigurationsdatei herunterzuladen und in der Konfigurations-Datei „application-properties“, falls notwendig , die Zugangsdaten (Datenbankname, username, passwort) anzugeben und gegebenenfalls den SQL-Dialekt der Datenbanktechnologie zu spezifizieren, und die JPA und Spring konfigurieren die Verbindung zur Datenbank voll automatisch. Aus reinem Interesse, die open source Datenbank „PostgreSQL “einmal zu verwenden und kennen zu lernen, wurde sie für diese Arbeit gewählt.

Konfiguration der Datenbankverbindung

Die wichtigsten Elemente der Konfiguration von Hibernate sind die Konfiguration der Datenbankverbindung, die Transaktions-Modalitäten sowie die Mapping-Metainformationen [1, S. 58]

Damit Hibernate die korrekten Abfragen generieren kann, da jede Datenbanktechnologie einen leicht unterschiedlichen SQL-Dialekt hat, muss der SQL-Dialekt, respektive die verwendete Datenbank in Spring Boot angegeben werden. Dies wird in einer Datei „spring.jpa.properties.hibernate.dialect“ gespeichert, die via Konfigurationsdatei („application.properties“) des Maven/Spring Boot-Projekts eingelesen wird. Hibernate kann so flexibel mit unterschiedlichen Datenbank-Technologien mit sehr geringem Konfigurationsaufwand verbunden werden. [21]

5.4. Dokumentation mit Swagger

Swagger ist ein Tool zum Dokumentieren und Testen von API's. Mit Spring und Spring Boot kann mit Hilfe von Maven sehr einfach eine Swagger-Representation der API generiert werden. Hierfür müssen zwei dependencies ins pom.xml-file integriert werden, die Swagger2-dependency und die SwaggerUI-dependency.

```

1 <!-- https://mvnrepository.com/artifact/io.springfox/springfox-swagger-ui -->
2 <dependency>
3   <groupId>io.springfox</groupId>
4   <artifactId>springfox-swagger-ui</artifactId>
5   <version>2.9.2</version>
6 </dependency>
7
8 <!-- https://mvnrepository.com/artifact/io.springfox/springfox-swagger2 -->
9 <dependency>
10  <groupId>io.springfox</groupId>
11  <artifactId>springfox-swagger2</artifactId>
12  <version>2.9.2</version>
13 </dependency>

```

Listing 5.2: Maven Dependencies Swagger

Dann muss eine Konfigurationsklasse erstellt werden, die es Swagger ermöglicht, die nötigen Informationen aus dem Projekt, die vorhandenen Endpoints, zu lesen und daraus das Swagger User-Interface zu generieren.

```

1 package ch.keilestats.api.application.config;
2
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.Configuration;
5
6 import springfox.documentation.builders.RequestHandlerSelectors;
7 import springfox.documentation.service.ApiInfo;
8 import springfox.documentation.spi.DocumentationType;
9 import springfox.documentation.spring.web.plugins.Docket;
10 import springfox.documentation.swagger2.annotations.EnableSwagger2;
11
12 import static springfox.documentation.builders.PathSelectors.regex;
13
14 /*Class to configure and enable automatic Swagger2 documentation for the API*/
15 @EnableSwagger2
16 @Configuration
17 public class SwaggerConfig {
18
19   /* Docket on which Swagger2 will run on */

```

```
20 @Bean
21 public Docket keileAPI() {
22     return new Docket(DocumentationType.SWAGGER_2).select()
23         .apis(RequestHandlerSelectors.basePackage("ch.keilestats.api.application
24             ")) .paths(regex("/api.*"))
25         .build().apiInfo(metaInfo());
26 }
27 // Titletext to be displayed in the Swagger html-file
28 private ApiInfo metaInfo() {
29
30     ApiInfo apiInfo = new ApiInfo("Keile Stats API",
31         "API for saving and reading " + "statistics of a just-for-fun Icehockey
32         Team", "1.0",
33         "Terms of Service", "", "", "");
34
35     return apiInfo;
36 }
```

Listing 5.3: Swagger Configuration Class

Über die URI <http://localhost8080/swagger-ui.html> kann nun auf die Swagger-Darstellung der API zugegriffen werden und diese damit getestet werden.

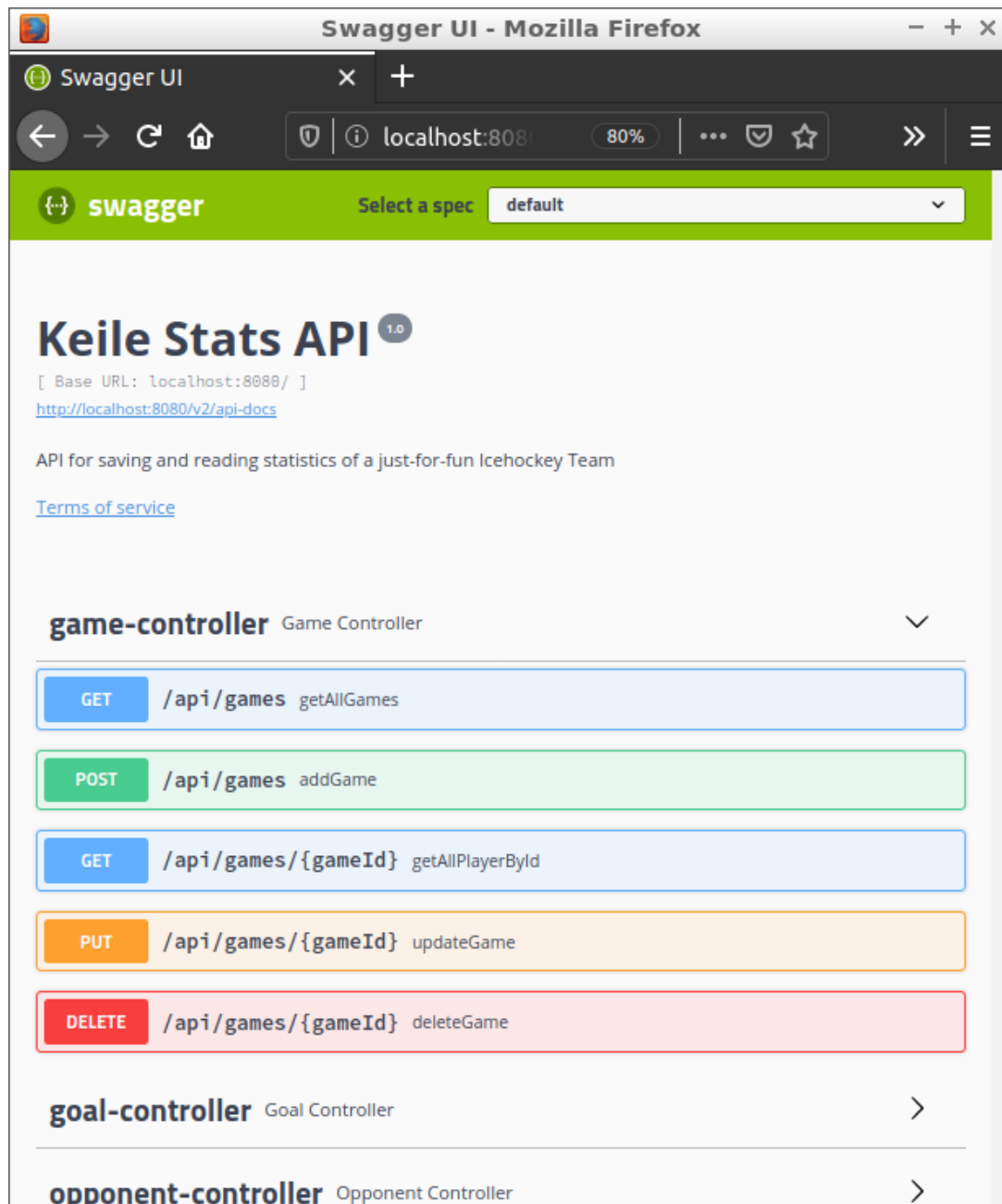


Abbildung 5.2.: Swagger Interface

6

Abschliessende Bemerkungen

6.1. Zusammenfassung

6.2. Probleme und Herausforderungen

Schwierigkeiten: post methode implementieren. Verstehen, wie repositories funktionieren. Verstehen, was genau im Hintergrund geschieht. Dies ist eine sehr grosse Herausforderung in der Arbeit mit dem Spring Framework und Spring Boot als Spring-Neuling und mit nur theoretischen groben Grundkenntnissen der Themen Software-Architektur mit Webservices, Http-Protokoll oder Remote Procedure Calls. Es ist so viel bereits automatisch konfiguriert, wenn man mit dem Spring initialiser und Maven ein Spring Boot Projekt erstellt, dass man zwar sehr schnell eine funktionierende Applikation erstellen kann. Für einen Einsteiger ist jedoch das Programmieren und das Debugging sehr anspruchsvoll, weil, wenn man die Architektur des Spring-Frameworks und Konfiguration und Funktionsweise nicht vertraut ist, man nicht sieht, was jetzt alles genau im Hintergrund abläuft und wie das Programm funktioniert. Es erfordert eine lange Auseinandersetzung mit der Grundstruktur und Grundfunktionsweise des Spring Frameworks und von Spring Boot, um, wenn etwas nicht funktioniert, einen Ansatzpunkt zu haben, wo genau das Problem liegt, auch wenn natürlich die Fehlermeldungen klar sind und helfen. Wenn man mit den Konzepten und Klassen, die in den Fehlermeldungen erwähnt sind, nicht vertraut ist, hilft das einem nichts, respektive es braucht ein aufwändiges Recherchieren und vertraut machen mit dem Spring Framework, bis man einigermaßen einen Anhaltspunkt hat, wo der Fehler liegt.

Zudem ist es sehr hilfreich und nötig, sich zuerst theoretische und praktische Grundlagen der Themenbereiche Webservices, Remote Procedure Calls oder HTTP anzueignen, um dann erst überhaupt sich damit zu beschäftigen, wie das Spring-Framework nun diese Herausforderungen angeht. Schwierigkeit auch, datenbankmodell via hibernate respektive Annotationen in java klassen auszudrücken. z.B. many to many relation kann mit annotationen angegeben werden. Schaffen, dass die korrekten Daten in die Datenbank eingegeben werden. Evtl. hätte man ein Datenmodell wählen können, das näher an den Rohdaten ist, statt den eher theoretischen Approach mit einem Entity-Relationship Model, der von der Datenbanktheorie her stammt zu wählen. D.h. eher von den Rohdaten ausgehen und daraus ein Domain-model ableiten, als von der theoretischen Datenbankmodellierung ausgehen, basierend auf der realen Welt entsprechende Entitäten.

Annotationen, die Boilerplate-Code helfen zu vermeiden, helfen für schnelle Entwicklung. Entstanden aus der Praxis, was man merkt. Nachteil ist fehlende Transparenz. Um zu lernen, wie die Software funktioniert, ist expliziter Boilerplate-Code hilfreicher, weil so klar ersichtlich ist, welche Funktionalitäten benötigt und verwendet werden. Beispielsweise bei den Datenbankabfragen, aber auch beim Handling von Http-Requests, sprich bei den Controllern, im Prinzip überall.

Diese Bachelorarbeit deckt ein breites Gebiet an Themen ab: das Spring Framework, ORM, Hibernate, Spring Boot, REST-Prinzipien und HTTP. Für alles ist ein gewisses Verständnis gefragt, um diese Arbeit gut umsetzen zu können. Die Arbeit kann aber nicht bei jedem dieser Aspekte in die Tiefe gehen, deshalb wurde meist so viel Kenntnisse und Verständnis zu den einzelnen Themen erarbeitet, die nötig waren um das praktische Ziel der Arbeit, die Erstellung von funktionierenden, mehr oder weniger sauber programmierten REST-Endpoints zu erreichen.

A

Abkürzungen

ORM	Objekt/Relationales Mapping
JDBC	Java Database Connectivity
JPQL	Java Persistence Query Language
MVC	Model-View-Controller
EJB	Enterprise Java Beans
ERM	Entity Relationship Model
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IANA	Internet Assigned Numbers Authority
JPA	Java Persistence API
POJO	Plain Old Java Object
POM	Project Object Model
REST	Representational State Transfer
DI	Dependency Injection
AOP	Aspect Oriented Programming
RMI	Remote Method Invocation
API	Application Programming Interface
MVC	Model-View-Controller
SOAP	Simple Object Access Protocol
ANSI	American National Standards Institute
CSS	Cascading Style Sheet
DBMS	Database Management System
DOM	Document Object Model
HL7	Health Level 7
HTML	Hypertext Markup Language
IoT	Internet of Things
IP	Internet Protocol
IPSec	Internet Protocol Security
JAXB	Java Architecture for XML Binding
JAX-RS	Java API for RESTful Web Services
JPQL	Java Persistence Query Language
JSON	JavaScript Object Notation
JSP	Java Server Pages
JSTL	Java Server Tag Library
NCSA	National Center for Supercomputing Applications

PHP	Hypertext Processor
QR	Quick Response
RFID	Radio Frequency Identification
ROA	Resource Oriented Architecture
SAX	Simple API for XML
SOA	Service Oriented Architecture
SQL	Structured Query Language
SPDY	SPeeDY, an open networking protocol
TCP	Transmission Control Protocol
URI	Unified Resource Identifier
URL	Uniform Resource Locator
VPN	Virtual Private Network
W3C	World Wide Web Consortium
WADL	Web Application Description Language
WSDL	Web Service Description Language
WoT	Web of Things
XML	eXtensible Markup Language
XSD	XML Schema Definition

B

Lizenz

Copyright (c) 2020 Marc Raemy.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

The GNU Free Documentation Licence can be read from [19].

Literaturverzeichnis

- [1] Christian Bauer and Gavin King. *Hibernate in Action*. Manning, 2005. 22, 28, 29, 35, 36, 37
- [2] Sun Microsystems Inc. JavaBeans. Oracle, 1997. 17
- [3] Rod Johnson. *Expert One-on-One J2EE Design and Development*. Wrox, 2002. 16
- [4] Rod Johnson, Jürgen Hoeller, et al. Spring 2.0.8, java/j2ee Application Framework - Reference Documentation, 2007. <https://docs.spring.io/spring/docs/2.0.x/spring-reference.pdf>. 16
- [5] Jacques Pasquier, Arnaud Durand, and Gremaud Pascal. Lecture notes advanced software engineering, October 2016. Departement für Informatik, Universität Freiburg (CH).
- [6] Michael Simons. *Spring Boot 2 - Moderne Softwareentwicklung mit Spring 5*. dpunkt.verlag, 2018. 33
- [7] Stefan Tilkov, Eigenbrodt Martin, Silvia Schreier, and Wolf Oliver. *REST und HTTP - Entwicklung und Integration nach dem Architekturstil des Web*. dpunkt.verlag, 2015. 24
- [8] Craig Walls. *Spring im Einsatz*. Hansen, 2012. 16, 17, 18, 20, 21, 23
- [9] Craig Walls. *Spring in Action*. Manning, 2019. 19, 20, 21, 22, 31, 32, 35

Webreferenzen

- [10] What is the Spring Framework really all about?, Java Brains. <https://www.youtube.com/watch?v=gq4S-ovWVlM> (Letzter Aufruf Juni 30, 2019).
- [11] Annotations. <https://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html> (Letzter Aufruf August 5, 2019).
- [12] Understanding the Basics of Spring vs. Spring Boot. <https://dzone.com/articles/understanding-the-basics-of-spring-vs-spring-boot> (Letzter Aufruf August 6, 2019).
- [13] Spring Boot Tutorial, How to Do in Java. <https://howtodoinjava.com/spring-boot-tutorials> (Letzter Aufruf August 27, 2019).
- [14] A Comparison Between Spring and Spring Boot. <https://www.baeldung.com/spring-vs-spring-boot> (Letzter Aufruf August 6, 2019).
- [15] Object Messages and Dependencies, YouTube-Kanal Knowledge Dose. <https://www.youtube.com/watch?v=W21CLd9zm9k> (Letzter Aufruf Sept 17, 2019).
- [16] Understanding Dependency Injection, YouTube-Kanal Java Brains. <https://dzone.com/articles/understanding-the-basics-of-spring-vs-spring-boot> (Letzter Aufruf August 6, 2019).
- [17] Difference between Spring and Spring Boot, Dzone. <https://dzone.com/articles/understanding-the-basics-of-spring-vs-spring-boot> (Letzter Aufruf Oktober 5, 2019).
- [18] API Endpoints Tutorial, YouTube Ethan Jarell. <https://www.youtube.com/watch?v=C470XGASGX0> (Letzter Aufruf August 28, 2019).
- [19] Free Documentation Licence (GNU FDL). <http://www.gnu.org/licenses/fdl.txt> (Letzter Aufruf July 30, 2005).
- [20] Steps toward the Glory of REST. <https://martinfowler.com/articles/richardsonMaturityModel.html> (Letzter Aufruf Juni 30, 2019).
- [21] Hibernate Framework Basic, Java Beginners Tutorial. <https://javabeginnerstutorial.com/hibernate/hibernate-framework-basic/> (Letzter Aufruf Dezember 12, 2019).
- [22] JPA and Hiberbate Tutorial for Beginners with Spring Boot and Spring Data JPA, YouTube-Kanal in28minutes. https://www.youtube.com/watch?v=MaIO_XdpdP8 (Letzter Aufruf November 8, 2019).

- [23] JSP and Servlets Tutorial : First Java Web Application in 25 steps, YouTube-Kanal in28minutes. <https://www.youtube.com/watch?v=Vvnliarkw48> (Letzter Aufruf September 22, 2019).
- [24] Spring framework tutorial for beginners with examples in eclipse | Why Spring Inversion of control. <https://www.youtube.com/watch?v=r2Q0Jz12qMQ> (Letzter Aufruf November 7, 2019).
- [25] Javadoc. <https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repository/JpaRepository.html> (Letzter Aufruf März 13, 2020).
- [26] Master hibernate and jpa with spring boot in 100 steps, udemy onlinekurs. <https://www.udemy.com/course/hibernate-jpa-tutorial-for-beginners-in-100-steps/> (Letzter Aufruf Januar 15, 2020). 22
- [27] Pom Reference, Apache Maven Documentation . <https://maven.apache.org/pom.html> (Letzter Aufruf August 27, 2019).
- [28] Spring Rest Docs - Documenting REST API, Example, YouTube-Kanal Java Techie. https://www.youtube.com/watch?v=ghn9p6d__Yc (Letzter Aufruf Februar 3, 2020).
- [29] REST principles explained. <https://www.servage.net/blog/2013/04/08/rest-principles-explained> (Letzter Aufruf Juni 1, 2019).
- [30] Building an Application with Spring Boot. <https://spring.io/guides/gs/spring-boot/> (Letzter Aufruf August 6, 2019).
- [31] How to create a Spring Boot project in Eclipse. <https://www.youtube.com/watch?v=WZzGhWSJ6h0> (Letzter Aufruf Juni 30, 2019).
- [32] Spring Boot API in 10 Minuten - Tutorial Deutsch. <https://www.youtube.com/watch?v=pkVrAmGb1XQ> (Letzter Aufruf August 6, 2019).
- [33] Spring Boot Anwendung mit MySQL Datenbank verbinden - Tutorial Deutsch. <https://www.youtube.com/watch?v=TJfwKT-mxOA> (Letzter Aufruf August 6, 2019).
- [34] Java Spring Online Dokumentation. <https://docs.spring.io/spring-boot/docs/current/reference/html/getting-started-first-application.html> (Letzter Aufruf September 20, 2019). 18, 32
- [35] Spring Tutorial, JavaTPoint. <https://www.javatpoint.com/spring-tutorial> (Letzter Aufruf Oktober 28, 2019).
- [36] REST API Documentation using Swagger2 in Spring Boot, YouTube-Kanal Tech Primers. <https://www.youtube.com/watch?v=HHyjWc0AS18> (Letzter Aufruf Februar 3, 2020).
- [37] What is REST API, YouTube-Kanal Telusko. <https://www.youtube.com/watch?v=qVTAB8Z2VmA> (Letzter Aufruf Juni 30, 2019).
- [38] Introduction to Servlets, YouTube-Kanal Telusko. <https://www.youtube.com/watch?v=CRvcn7GKrFO> (Letzter Aufruf Sept 19, 2019).
- [39] What is Spring Boot? Introduction, YouTube-Kanal Telusko. <https://www.youtube.com/watch?v=Ch163VfHtvA> (Letzter Aufruf September 23, 2019).

- [40] What is a RESTful API? Explanation of REST and HTTP, YouTube-Kanal Traversy Media. <https://docs.spring.io/spring/docs/2.0.x/spring-reference.pdf> (Letzter Aufruf August 28, 2019).
- [41] Domain Model, Wikipeida. https://en.wikipedia.org/wiki/Domain_model (Letzter Aufruf Februar 5, 2020).
- [42] JavaBeans, Wikipedia. <https://de.wikipedia.org/wiki/JavaBeans> (Letzter Aufruf Februar 1, 2020).
- [43] Apache Maven, Wikipedia. https://de.wikipedia.org/wiki/Apache_Maven (Letzter Aufruf August 27, 2019).