# Inhaltsverzeichnis

# KAPITEL 2

# Grundlagen

## 2.1 REST-API

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

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

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

In den nachfolgenden Abschnitten sollen die allgemeine Definition, die Vor- und Nachteile und die Qualitätsmerkmale einer API näher beleuchtet werden. Anschließend wird auf die Grundprinzipien von REST detaillierter eingegangen. Als letztes wird das Konzept HATEOAS, welches laut Roy Fielding ein Muss[1] für jede RESTful API ist, genauer erläutert.

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

### 2.1.1 Allgemeine Definition einer API

Laut [Spi16, S. 7] definiert Kai Spichale eine API mit den Worten von Joshua Bloch wie folgt: „Eine API spezifiziert die Operationen sowie die Ein- und Ausgaben einer Softwarekomponente.

1 siehe [Fie08]

Ihr Hauptzweck besteht darin, eine Menge an Funktionen unabhängig von ihrer Implementierung zu definieren, sodass die Implementierung variieren kann, ohne die Benutzer der Softwarekomponente zu beeinträchtigen“. Des weiteren unterteilt dieser die APIs in zwei Kategorien ein, in Programmiersprachen- und Remote-APIs. Des weiteren definiert er die APIs der Programmiersprachen als abhängig und die Remote als unabhängig gegenüber der Sprache und der Plattform.

### 2.1.2 Vorteile einer API

*Stabilität durch lose Kopplung*

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

*Portabilität*

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

*Komplexitätsreduktion durch Modularisierung*

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

*Softwarewiederverwendung und Integration*

Neben dem verbergen von Details zur Implementierung sollten APIs Funktionen einer Komponente leicht verständlich bereitstellen, um API-Nutzern eine einfache Integration bzw. Verwendung zu ermöglichen. Aus diesen sollten APIs auch dahingehend optimiert werden.

### 2.1.3 Nachteile einer API

*Interoperabilität*

Ein Nachteil, der allerdings nur Programmiersprachen-APIs betrifft, ist die Interoperabilität zu anderen Programmiersprachen. Beispielsweise kann ein Programm, welches in Go geschrieben

wurde, nicht auf die Java-API zugreifen. Als Problemlösung stehen hierbei aber die Remote-API bereit. Diese arbeiten mit Protokollen wie HTTP oder Advanced Message Queuing Protocol (AMQP), welche sprach- und plattformunabhängig sind[1].

*Änderbarkeit*

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

### 2.1.4 Qualitätsmerkmale

*Benutzbarkeit*

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

*Effizienz*

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

*Zuverlässigkeit*

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

1 siehe Kapitel 2.1.1

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

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

### 2.1.5 Grundprinzipien von REST

*Eindeutige Identifikation von Ressourcen*

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

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

*Verwendung von Hypermedia*

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

*Verwendung von HTTP-Standardmethoden*

Um die vom Server bereit gestellten Links ordnungsgemäß auszuführen, müssen neben den URIs auch einheitliche Schnittstellen bekannt sein. Das setzt voraus, dass alle Clients über Verwendung und Semantik der Schnittstellen Bescheid wissen. An dieser Stelle kommen die HTTP-Standardmethoden zum Einsatz. Die Schnittstellen von HTTP bestehen dabei im wesentlichen aus den Funktionen *GET*, *HEAD*, *POST*, *PUT* und *DELETE*, welche alle in der HTTP-Spezifikation[1] definiert sind. Die Methoden *PATCH* oder auch *LINK* waren dabei nicht von Anfang in der Spezifikation enthalten, sonder wurden erst in nachträglich hinzugefügt.

1 siehe [Fie]

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

*Unterschiedliche Repräsentationen von Ressourcen*

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

*Statuslose Kommunikation*

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

### 2.1.6 HATEOAS

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

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

- „Mit »State« ist der Zustand der Ressource gemeint, deren Zustandsübergänge durch die »Engine« definiert werden.“

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

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

## 2.2 Das Build-Tool Gradle

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

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

### 2.2.1 Eigenschaften von Gradle

*Declarative Dependency Management*

Gradle biete eine komfortable Möglichkeit mit Abhängigkeiten umzugehen. Denn viele Projekte verwenden andere Bibliotheken oder Frameworks, welche gegebenenfalls eigenen Abhängigkeiten aufweisen. Dadurch kann es sehr mühselig werden die ganze Abhängigkeitsstruktur mit alle Versionen zu verwalten. Es müssen nur die benötigten Abhängigkeiten definiert werden und Gradle kümmert sich um den Download dieser, inklusive aller Unterabhängigkeiten. Es ist wichtig zu wissen das Gradle nur das *Was* aber nicht das *Wie* gesagt werden muss.

*Declarative Builds*

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

*Build by Convention*

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

*Incremental Builds*

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

*Gradle Wrapper*

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

*Plugins*

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

### 2.2.2 Verwaltung von Projekten und Tasks

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

wobei jedes Projekt wiederum ein oder mehrere Tasks beinhaltet.

*Projekte*

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

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

*Tasks*

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

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

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

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

copyCompiledFilesIntoDist.dependsOn(build)
```

**Listing 2.1:** Beispiel: Gradle-Task

## 2.3 Schachnotationen FEN und SAN

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

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

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

**Abbildung 2.1:** Startposition eines Schachspiels in der FEN

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

a4

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

Nc3

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

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

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

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

## 2.4 Schachregeln

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

# Literatur


[ANO] ANONYM: „Why AMD?“ (), Bd. URL: http://requirejs.org/docs/whyamd.html (besucht am 13. 04. 2018) (siehe S. 37).

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

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

[Fie] FIELDING, ROY THOMAS u. a.: *Hypertext Transfer Protocol – HTTP/1.1*. URL: https://tools.ietf.org/html/rfc2616 (besucht am 12. 05. 2018) (siehe S. 7).

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

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

[Mas] MASHKOV, SERGEY: „DOM trees“. (), Bd. URL: https://github.com/Kotlin/kotlinx.html/wiki/DOM-trees (besucht am 13. 04. 2018) (siehe S. 37).

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

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

[Wat] WATSON, GRAY: „OrmLite - Lightweight Object Relational Mapping (ORM) Java Package“. (), Bd. URL: http://ormlite.com/ (besucht am 10. 04. 2018) (siehe S. 37).

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Listings



1 Quelle: [Wat]
2 verändert nach [Wat]
3 Quelle: [ANO]
4 Quelle: [Mas]

# 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 14. Mai 2018 Felix Dimmel

# A First chapter of appendix

## A.1 Parameters