Hochschule für Technik und Wirtschaft
Dresden

Fakultät Informatik/Mathematik

Bachelorarbeit

im Studiengang Informatik

Thema:
Demonstration eines RESTful Webservices am Beispiel eines Schachservers

Eingereicht von:	Christoph Kretzschmar
Eingereicht am:	02. September 2015
Betreuender Hochschulprofessor:	Prof. Dr.–Ing. Jörg Vogt
Zweitgutachter:	Prof. Dr.–Ing. Wilfried Nestler

Inhaltsverzeichnis

Abkürzungsverzeichnis

ABNF	Augumented Backus-Naur-Form
API	Application Programming Interface
DBMS	Database-Management-System
DI	Dependency Injection
EBNF	Extended Backus-Naur-Form
FEN	Forsyth-Edwards-Notation
HATEOAS	Hypermedia as the engine of application state
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JSON	JavaScript Object Notation
MAC	Media Access Control
MVC	Model-View-Controller
REST	Representational State Transfer
RPC	Remote Procedure Call
RTT	Round-Trip-Time
SAN	Standard Algebraic Notation
SOAP	Simple Object Access Protocol
URI	Uniform Resource Identifiers
URL	Unified Resource Locator
URN	Unified Ressource Name
WSDL	Web Services Description Language
WWW	World Wide Web
XML	Extensible Markup Language

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1. Einleitung

Ein Webservice ist eine Methode, um unterschiedliche Anwendungen an eine eigenständige Datengrundlage anbinden zu können. Dabei gibt es verschiedene Möglichkeiten Web-APIs zu implementieren. Technologien im Kontext eines Webservices sind dabei: Simple Object Access Protocol (SOAP), Remote Procedure Call (RPC) und die Representational State Transfer (REST) Architektur.

Im Kontext dieser Arbeit wird die REST Architektur verwendet, da es gegenüber SOAP oder RPC Vorteile in der Verwendung, insbesondere in der Selbstbeschreibung gegenüber dem Konsumenten der API, hat. Es ist im Vergleich zu SOAP oder RPC eine grobe Richtlinie, bzw. Architektur und kein speziell definiertes Netzwerkprotokoll. Dadurch sind dem Entwickler viele Freiheiten bei der Gestaltung der Kommunikationsmechanismen und der verwendeten Kommunikationsformate gegeben.

Bei SOAP wird das Interface für den Zugriff durch die Web Services Description Language (WSDL) beschrieben. Dadurch wird das verwendete Kommunikationsformat durch den Server festgelegt und entsprechend beschränkt. Gleiches gilt auch für RPC, da dort Aktor (Objekt), Aktion (Methode) und die zu verarbeitenden Parameter übergeben werden, welche durch den Server, vorgegeben werden. Bei SOAP und RPC sind die Kommunikationsmechanismen und -formate daher immer Applikationsspezifisch definiert.

Die REST Architektur grenzt sich dadurch ab, dass ein generisches Interface für die Kommunikation definiert wird. Dieses Interface ist bei allen RESTful Webservices gleich. Durch die von REST geforderte Aushandlung des Kommunikationsformates, kann ein Konsument der API bestimmen, wie die Information ausgetauscht werden soll.

Im Verlauf dieser Arbeit wird das Application Programming Interface (API) für einen Schachserver entworfen und implementiert, wobei die einzelnen Überlegungen und Schritte so festgehalten werden, dass diese sich auch für die Planung anderer Webservices eignen. Die, aus dieser Arbeit resultierende API, soll dabei die Vorgaben und Restriktionen, welche von der REST Architektur gestellt werden, erfüllen. Das Ziel der Arbeit ist es allgemeingültige Methodiken, für die Planung und Bewertung einer REST API, zu entwickeln.

Im Kapitel Grundlagen werden die Begriffe und Technologien, welche für das Verständnis der vorliegenden Arbeit notwendig sind, erklärt. Zudem werden die Restriktionen eines RESTful Webservices, anhand der Dissertation von Roy Thomas Fielding, erläutert.

Im darauf folgenden Kapitel werden die theoretischen Aspekte des Themas bearbeitet. Dabei werden verschiedene Ansatzwege für, der Meinung des Autors nach, wichtigen Themen einer API untersucht und entworfen. Die Implementierungsdetails werden im vierten Kapitel zusammengefasst, wobei auf die verwendeten Technologien und die Umsetzung des Entwurfs eingegangen wird. Das Kapitel fünf schließt die Arbeit mit Anregungen zu nicht behandelten Themen und möglichen Erweiterungen einer REST API ab.

Für das Verständnis dieser Arbeit werden die Grundbegriffe des Hypertext Transfer Protocol (HTTP) Protokolls vorausgestzt. Für den Entwurf wird vorausgesetzt, dass einfache Datenstrukturen, wie Listen oder assoziative Arrays, bekannt sind. Die Implementierung setzt weiterhin einfache Programmierkenntnisse voraus. Für das Verständnis der Entwurfs- und Implementierungsdetails sind Kenntnisse zum Schachspiel nützlich.

2. Grundlagen

Dieses Kapitel widmet sich den Grundlagen und erläutert die technischen Begriffe und Technologien, die für das Verständnis dieser Arbeit notwendig sind. Hierfür wird zuerst die Adressierung von Ressourcen im World Wide Web (WWW) erläutert. Danach wird das Hypertext Transfer Protocol vorgestellt, welches eingesetzt wird, um die Informationen zwischen dem Webservice und einem Konsumenten des Webservices zu ermöglichen. Auf diesen Grundlagen aufbauend, wird die REST Architektur vorgestellt und die daraus resultierenden Restriktionen beschrieben. Abschließend wird die, in dieser Arbeit verwendete, elektronische Schachnotation vorgestellt.

2.1. Uniform Resource Identifier

Dieser Abschnitt soll einen kurzen Überblick über das Konzept eines Uniform Resource Identifiers (URI) ermöglichen.

Die URI wird in [Ber94a] erstmals beschrieben und durch [Ber05] erweitert und genauer definiert. Die URI bildet dabei die Obermenge zu dem Unified Resource Locator (URL), beschrieben in [Ber94b] und dem Unified Ressource Name (URN), beschrieben in [Sol94].

Eine URL beschreibt, wo sich eine Ressource befindet und mit welchem Protokoll bzw. Schema auf diese zugegriffen werden kann. Beschrieben wird die URL durch:

`<schema>:<schema-spezischischer Teil>`

Beim Zugriff auf eine Ressource, welche durch das HTTP Protokoll übertragen werden soll, kann entsprechend der obigen Definition für eine URL folgendes Beispiel betrachtet werden `http://domain.tld/ressource`. Dabei beschreibt `http://` das verwendete Protokoll, `domain.tld` den Host, mit dem die Kommunikation stattfinden soll und `/ressource` gibt den Pfad der abzufragenden Ressource an. Ein weiteres Beispiel für eine URL wäre `mailto:recipient@domain.tld`. Dabei entspricht das Schema in diesem Fall keinem Netzwerkprotokoll, sondern einem Schema, um eine E-Mail zu adressieren.

Allgemein besteht eine URI aus einem Schema, einer Autorität (authority), einem Pfad (path), einer Abfrage (query) und einem Fragment Teil. Damit kann eine URI

durch folgenden Augumented Backus-Naur-Form (ABNF) Ausdruck grob definiert werden.[1]

```
1 URI = scheme ":" "//" authority [ "/" path ] [ "?" query ] [ "#"
    fragment ]
```

Listing 2.1: Vereinfachte URI Darstellung in ABNF

Im Verlauf der Arbeit werden relative URIs genutzt. Diese sind unabhängig vom Schema oder der Autorität. Dies ermöglicht es URIs, unabhängig von Protokoll oder verarbeitender Autorität, zu definieren. Die relativen URIs bestehen aus dem Pfad- und Abfrage-Teil, der in Listing 2.1 angegebenen ABNF.

2.2. Hypertext Transfer Protocol

Das Hypertext Transfer Protocol (HTTP) wird durch [Ber96] und [Fie99] beschrieben und definiert.

Eine Anfrage wird durch eine URI dargestellt. Dabei entspricht das Schema `http` und die Autorität entspricht dem Server, welcher die Anfrage bearbeiten soll. Der Pfad gibt an, auf welche Ressource zugegriffen werden soll. Optional kann eine Abfrage hinzugefügt werden, welche vom verarbeitenden Server mit beachtet werden soll.

Ein Beispiel für die URI einer Anfrage sieht wie folgt aus:

`http://domain.tld/pfad/ressource?param1=5&param2=6`

2.2.1. Content-Negotiation

Das HTTP Prokotoll bietet die Möglichkeit die für eine Kommunikation verwendeten Datenformate auszuhandeln. Dabei wird die Content-Negotiation genutzt, um zwischen den beiden Parteien, Server und Konsument, ein geeignetes Format auszuhandeln. Diese Verfahren ist zu jeder Zeit innerhalb des Protokolls möglich.

Das Datenformat (Content-Type), mit dem Server und Konsument die Daten austauschen, wird durch Angaben im Protokoll Header gesteuert. Im `Accept` Header gibt die jeweilige Gegenseite an, welche Datenformate akzeptiert werden. Durch den

[1] Eine exakte Definition kann [Ber05] entnommen werden.

`Content-Type` Header wird bei einer Antwort das zurückgegebene Datenformat angegeben. Eine Kommunikation ist möglich, wenn die unterstützten Datenformate des Servers und des Konsumenten übereinstimmen, bzw. eine Schnittmenge größer Null bilden. Ist ein Datenaustausch zwischen Server und Konsument nicht möglich, sendet der Server den Fehler-Statuscode 406.

2.2.2. Methoden

Für den Zugriff auf Ressourcen, welche durch eine URI referenziert werden, sind verschiedene Zugriffsmethoden definiert.

Die Methoden werden dabei in vier unterschiedliche Kategorien unterteilt. Dabei gibt es den Begriff der sicheren und unsicheren Methoden, wie auch den Begriff der idempotenten und nicht idempotenten Methoden.

Eine Methode gilt dann als sicher, wenn diese nur lesend auf die Ressource zugreift. Sobald die Ressource verändert wird, ist eine Methode als unsicher definiert.

Idempotenz beschreibt, dass eine Methode, die einen Seiteneffekt hat, wie z.B. das Verändern einer Ressource, bei identischen und aufeinanderfolgenden Anfragen in denselben Ressourcen auf dem Server resultieren. Entsprechend können idempotente Anfragen beliebig oft ausgeführt werden, ohne dass die resultierende Ressourcendarstellung sich unterscheidet.

Aus diesen Erklärungen folgt, dass eine sichere Methode immer idempotent ist. Jedoch sind nicht alle idempotenten Methoden auch sicher. Dafür sind alle nicht idempotenten Methoden unsicher.

Damit festgestellt werden kann, welche Methoden auf einem Endpunkt definiert sind, kann der Server diese, durch einen `Allow` Header, dem Konsumenten bei einer Anfrage signalisieren. Greif ein Konusment auf einen Endpunkt mit einer nicht unzerstüzten Methode zu, kann der Server diesen durch den Header auf die möglichen Zugriffsmethoden hinweisen. Weiterhin kann ein Konsument durch die OPTIONS Methode diese Information, gegenüber eines Endpunktes, anfragen.

Die genauen Beschreibungen der einzelnen Methoden befindet sich im Anhang A.1.

2.2.3. Statuscodes

Das Protokoll sieht es vor, dass der verarbeitende Server den Erfolg oder Misserfolg über das Ausführen, einer angefragten Aktion, dem Konsumenten mitteilen kann. Für diese Information existieren nummerische Statuscodes, welche in verschiedene Erfolgs- und Fehlergruppen eingeteilt sind. Weiterhin kann der Server den Konsumenten, durch die Kombination mit entsprechenden HTTP Headern, auf ein weiteres Vorgehen hinweisen. Nach dem erfolgreichen Erstellen einer Ressource durch eine POST Anfrage kann der Server, z.B. durch den Statuscode 201 signalisieren, dass die Ressource erfolgreich erstellt wurde. Des Weiteren kann durch den Header `Location` signalisiert werden, wie die URI der erstellten Ressource lautet. Ein Konsument kann nun dieser URI nachgehen und die Repräsentation beispielsweise in den lokalen Cache aufnehmen.

Im Anhang A.2 befinden sich die im Standard definierten Statuscodes.

2.3. Fieldings REST Definition

Der Begriff des Representational State Transfer (REST) wurde von Roy Thomas Fielding in seiner Dissertation (siehe [Fie00]) beschrieben. Fielding, einer der Autoren des HTTP 1.1 Standards (siehe [Fie99]), hat in seiner Arbeit anhand von netzwerkarchitektonischen Bewertungen und Gegebenheiten das HTTP Object Model weiterentwickelt und darauf basierend Restriktionen definiert.

Werden diese Restriktionen eingehalten, so können die darauf aufbauenden Webservices, aufgrund der zugrundeliegenden Architekturbedingungen, sehr gut skaliert werden. In vielen APIs, werden jedoch häufig einige dieser Beschränkungen nicht beachtet (vgl. [Fie08]), wodurch diese, der Definition nach, keine REST API darstellen, sondern nur eine Web-API mit REST Charakter.

Auf der Grundlage der Kapitel fünf und sechs seiner Dissertation [Fie00] und dem Blogeintrag [Fie08] ergeben sich die folgenden Limitierungen und Definitionen, welche in dieser Sektion zusammengefasst werden.

2.3.1. Adressierbarkeit

Ein RESTful Webservice ist durch eine eindeutige URI erreichbar. Zudem muss für jede, vom Webservice verwaltete, Ressource eine eindeutige URI existieren. Das verwendete Netzwerkprotokoll muss daher mindestens die Möglichkeit dieser Adressierungen umsetzen. Das HTTP Protokoll erfüllt diese Anforderung und eignet sich daher für eine REST API.

2.3.2. Standardkonformität

In [Fie08] wird im zweiten Punkt der Auflistung definiert, dass nur standardisierte Verfahren, Datentypen und Protokolle eingesetzt werden sollen. Dies soll es vereinfachen den Webservice zu verwenden. Sollte in einem Standard eine Definition nicht eindeutig sein, so muss diese vom Webservice, bzw. dessen Dokumentation eindeutig definiert werden. Im HTTP Protokoll wird z.B. die PATCH Methode, im Kontext der Verwendung, bzw. des Inhaltes der Anfrage, nicht genau definiert. Diese Definition muss von dem Webservice und der zugehörigen Dokumentation ausgeführt werden.

2.3.3. Zustandslosigkeit

Ein RESTful Webservice darf keinen Zustand zwischenspeichern. Ist ein Zustand gewünscht, so muss dieser vom Konsumenten des Webservices verwaltet werden. Jede Anfrage muss daher, unabhängig zu einer ggf. vorherigen Anfrage, vom Webservice verarbeitet werden können. Entsprechend müssen immer alle notwendigen Informationen, zur Verarbeitung der Anfrage, transferiert werden.

Eine SessionID oder ein Cookie widersprechen diesem Prinzip, da der Webservice sich an den Zustand, der durch eine Session oder einen Cookie impliziert wird, erinnern müsste. Entsprechend können diese Methoden, z.B. zum Identifizieren eines Konsumenten, nicht verwendet werden.

Ein großer Vorteil der Zustandslosigkeit ist eine gute Skalierbarkeit der Anwendung. Bei einer Lastverteilung muss kein Austausch zwischen den einzelnen Servern erfolgen. Diese können unabhängig voneinander die eingehenden Anfragen der Konsumenten beantworten.

2.3.4. Zugriffsmöglichkeiten

Der Zugriff auf verschiedene Ressourcen wird am Beispiel von HTTP definiert. Dieses Protokoll besitzt eine Trennung zwischen lesenden und verändernden Zugriffsarten auf eine, durch eine URI definierte, Ressource. Durch die definierten HTTP Methoden ist ein einheitliches Interface, für den Zugriff auf diese, gewährleistet.

2.3.5. Datenformate

Das Datenformat, der übertragenen Informationen, soll sich den Anforderung des Konsumenten anpassen können und nicht fest vom Webservice definiert werden. Das HTTP Protokoll erfüllt diese Bedingung durch die eingebaute Content-Negotiation (vgl. Abschnitt 2.2.1). Der Webservice sollte dabei standardisierte Formate, wie z.B. HTML, JSON oder XML unterstützen (vgl. Abschnitt 2.3.2).

Da die Kommunikation vorrangig über die Ressourcenrepräsentation, also dem Datenformat stattfindet, muss im Design des Webservices besonders darauf geachtet werden, geeignete Formate zu verwenden. Bei standardisierten Formaten werden die Verarbeitungsregeln vom jeweiligen Standard definiert.

Wichtig ist hierbei die Repräsentation gegenüber dem Konsumenten des Webservices. Wie die Ressourcen intern verwaltet werden, bzw. welche Darstellung für die persistente Speicherung gewählt wird, ist nicht zu beachten.

2.3.6. Selbstbeschreibung

Ein RESTful Webservice soll sich selbst beschreiben können, dies beinhaltet, unter anderem, die Dokumentation der Endpunkte. Dabei wird zusätzlich davon ausgegangen, dass eine Navigation vom Startpunkt des Webservices durch Übergänge, welche vom Webservice in der jeweiligen Antwort des aktuellen Zustandes vorgegeben werden, durch den kompletten Webservice ermöglicht werden. Die Möglichkeiten werden vom Konsumenten, anhand der bekannten Medientypen und Kommunikationsmechanismen, limitiert. Beide können durch Code-On-Demand, sofern der Konsument oder der Webservice dies unterstützen, erweitert werden. Der Zustand wird dabei immer vom Konsumenten des Webservices verwaltet und ggf. zwischengespeichert.

Das Beschreiben der Übergänge, in der Antwort des Webservices, wird auch Hypermedia as the engine of application state (HATEOAS) genannt.

2.4. Elektronische Schachnotation

Damit eine Schachpartie abgebildet werden kann, muss eine geeignete Darstellung definiert werden. Für diese Arbeit wurde sich für die, in diesem Abschnitt behandelten, Formate entschieden, da diese standardisiert sind und explizit für diesen Anwendungsfall durch [Edw94] definiert wurden.

2.4.1. Forsyth-Edwards-Notation

Die Forsyth-Edwards-Notation (FEN) beschreibt den Zustand einer aktuellen Schachpartie. Die FEN kann dabei durch folgende Extended Backus-Naur-Form (EBNF), wie in [Wik15a] definiert, dargestellt werden.

```
FEN ::= Figurenstellung '␣' Am Zug '␣' Rochade
        '␣' en passant '␣' Halbzüge '␣' Zugnummer
Figurenstellung ::= Reihe '/' Reihe '/' Reihe
        '/' Reihe '/' Reihe '/' Reihe '/' Reihe '/' Reihe
Reihe ::= ( Figur | Leerfelder )+
Figur ::=  'p' | 'r' | 'n' | 'b' | 'q' | 'k'
        | 'P' | 'R' | 'N' | 'B' | 'Q' | 'K'
Leerfelder ::= '1' | '2' | '3' | '4'
        | '5' | '6' | '7' | '8'
Am Zug ::= 'w' | 'b'
Rochade ::= 'K' ['Q'] ['k'] ['q'] | 'Q' ['k'] ['q']
        | 'k' ['q'] | 'q' | '-'
en passant ::= '-'
        | ( ('a'|'b'|'c'|'d'|'e'|'f'|'g'|'h') ('3'|'6') )
Halbzüge ::= '0' | positiveGanzeZahl
Zugnummer ::= positiveGanzeZahl
```

Listing 2.2: Forsyth-Edwards-Notation in EBNF

In der Grammatik werden die Figuren für den weißen Spieler in Großbuchstaben dargestellt. Für den schwarzen Spieler werden die Figuren durch Kleinbuchstaben beschrieben. Leere Felder werden durch die Anzahl der leeren Felder bis zur nächsten Figur oder dem Ende des Spielbretts dargestellt.

Tabelle 2.1.: Schachfiguren in der Forsyth-Edwards-Notation

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

Eine Startsituation einer klassischen Schachpartie kann demnach durch folgende Notation beschreiben werden:

```
rnbqk bnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
```

2.4.2. Standard Algebraic Notation

Die Standard Algebraic Notation (SAN), ermöglicht es einzelne Züge einer Schachpartie darstellen zu können. Diese Arbeit betrachtet die Darstellung der Figuren in der englischen Notation, da diese sich mit den Bedeutungen der Figuren der FEN (vgl. Tabelle 2.1) gleicht.

Eine Bewegung wird durch den Großbuchstaben der jeweiligen Figur, unabhängig vom aktuellen Spieler, und dem Feld, auf welches diese sich bewegt, angegeben. Wird ein Bauer bewegt, so wird dessen Buchstabe weggelassen. Wenn sich beispielsweise ein Läufer von C3 nach E5 bewegt, so wird dies notiert als `Be5`. Bewegt sich jedoch ein Bauer von C4 nach C5, so wird nur `c5` angegeben.

Wird eine Figur von einem Gegenspieler geschlagen, so wird dies mit einem `x` in der Notation erfasst. Auch ein en passant kann über diese Notation erfasst werden. Dabei kann zusätzlich noch ein `e.p` an die Notation angehangen werden. Damit bei einem Bauern festgestellt werden kann, welcher den aktuellen Zug ausführen soll, wird das Feld, von welchem der Bauer schlägt, mit angegeben. Würde ein Bauer z.B. eine Figur von E4 auf D5 schlagen, so würde das durch `exd5` dargestellt werden. Bei einem en passant von E4 auf D6 würde das Geschehen durch `exd6e.p.` dargestellt werden.

Es kann vorkommen, dass ein Zug uneindeutig ist, da mehrere Figuren auf dem Spielbrett diesen ausführen können. Dieses Problem wird dadurch gelöst, dass nach der Beschreibung der zu bewegenden Figur noch dessen Feld beschrieben wird, von welchem die Figur kommt. Sind beide auf der gleichen Linie, so wird die Zahl der Reihe verwendet, auf welchem diese stehen. Sind beide nicht auf der gleichen Linie, so wird der Buchstabe der Linie der Figur verwendet. Zwei Beispiele sollen dies

verdeutlichen: Es befindet sich jeweils ein Springer auf G1 und D2. Beide können sich auf das Feld F3 bewegen. Da die Linie beider Figuren unterschiedlich ist, wird der Buchstabe der Linie verwendet. Daraus resultieren die Möglichkeiten: `Ngf3` und `Ndf3`. Würden sich die Springer auf G1 und G5 befinden, wäre die Linie gleich und die Zahl der Reihe würde für die Identifizierung genommen, dem Beispiel entsprechend `N5f3` und `N1f3`.

Erreicht ein Bauer das andere Ende des Spielbretts, so kann dieser zu einer anderen Figur umgewandelt werden. Dies kann in der Notation, durch ein Hinzufügen der gewünschten Figur an das Ende des Ziels, dargestellt werden. Soll ein Bauer zu einer Königin umgewandelt werden kann dies durch `e8Q` angegeben werden.

Eine Rochade kann in der Notation auch erfasst werden. Dabei wird eine Rochade auf der Seite des Königs durch `O-O` dargestellt. Für die Rochade auf der Seite der Königin wird `O-O-O` verwendet. Das Zeichen entspricht dabei dem großen Buchstaben `O` und nicht der Zahl `0`.

Führt ein Zug zu einem Schach, so wird ein + an den entsprechenden Zug angefügt. Für ein Schachmatt wird ein ++ oder # an das Ende des Spielzuges angefügt.

3. Entwurf eines RESTful Webservices

In diesem Kapitel werden die wichtigsten Eckpunkte einer API definiert und entworfen. Dabei werden zuerst die Ressourcen des Services identifiziert und definiert. Anschließend wird eine Möglichkeit für den Zugriff auf diese Ressourcen geschaffen. Im darauf folgenden Abschnitt wird die öffentliche Repräsentation der Ressourcen, an den definierten Zugriffsendpunkten, entwickelt. Im vierten Abschnitt wird die Umsetzung von HATEOAS, innerhalb des Webservices, beschrieben. Der letzte Abschnitt erläutert die Möglichkeiten zur Absicherung der API, durch das Vorstellen von Authentifizierungs- und Autorisierungsmechanismen.

Eine Ressource, innerhalb des Webservices, beschreibt die, vom Service, verwalteten Datenstrukturen. Diese können identisch mit der Repräsentation auf der Seite des Server sein oder nur einer limitierten öffentlichen Darstellung entsprechen. Die Transformation zwischen der öffentlichen Darstellung, die der Konsument erhält, hin zur physischen Repräsentation, die auf dem Server gespeichert wird, muss dabei transparent sein.

3.1. Definieren der verwalteten Ressourcen

Einer der wichtigsten Schritte in der Planung eines RESTful Webservices ist das korrekte Identifizieren der verwalteten Ressourcen.

Als erste Ressource eines Schachspieles kann das Schachbrett bzw. die Partie, das aktuelle Spiel, identifiziert werden. Eine Partie besteht in der einfachsten Definition aus maximal zwei Kontrahenten und dem Spielbrett.

Das Spielbrett wird dabei durch 2 Komponenten dargestellt werden. Die erste Komponente ist die Startsituation, die in der FEN dargestellt wird. Die Zweite beschreibt die Züge, welche von den Kontrahenten vorgenommen wurden und wird in der SAN dargestellt.

Für die Kontrahenten einer Partie wird eine Spieler Ressource eingeführt. Ein Spieler besitzt einen Namen und ein Passwort. Das Passwort kann für eine mögliche Authentifizierung verwendet werden (vgl. Abschnitt 3.5).

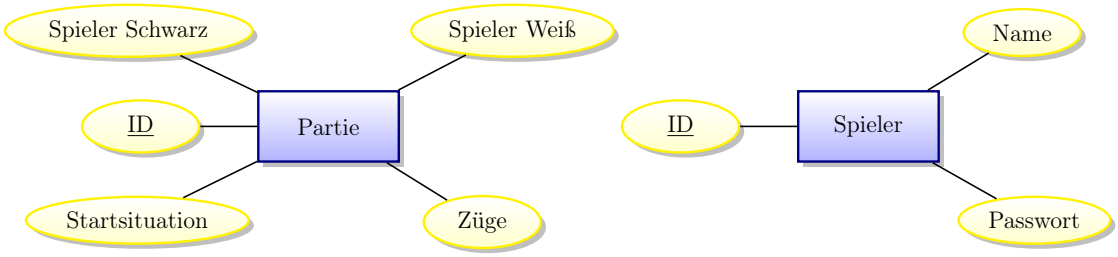
Um eine eindeutige Identifizierung der Ressourcen zu ermöglichen, wie im Abschnitt 2.3.1 gefordert, wird für jede Ressource eine numerische Identifikation eingeführt.

Diese wird beim Anlegen einer neuen Ressource, jeweils um 1, erhöht. Dadurch ergibt sich eine einzigartige, fortlaufende Nummer, die eine Ressource eindeutig identifizieren kann.

Die Spieler können einer Partie als weißer, bzw. schwarzer Spieler, anhand dieser einzigartigen Identifizierung, zugewisen werden.

Aus dieser Beschreibung ergibt sich die interne Ressourcendarstellung (siehe Abbildung 4.2), so wie diese vom Webservice verwaltet und gespeichert wird.

Abbildung 3.1.: Ressourcen des Schachservers



3.2. Zugriffsmöglichkeiten auf die verwalteten Ressourcen

Nach der Definition der Ressourcen müssen nun die möglichen Zugriffe und damit die Verwendungsmöglichkeiten der entstehenden API gestaltet werden.

Eine Konsument der API soll in der Lage sein sich alle Spieler und aktuellen Partien anzeigen lassen zu können. Zudem soll es möglich sein ,die Details einer Ressourcen zu holen, zu verändern und im Webservice wieder zu speichern.

Für den Zugriff, auf die Ressourcen, müssen daher Zugriffspunkte definiert werden. Die Zugriffspunkte des Webservices werden durch URIs definiert. Ausgehend von einem initialen Startpunkt innerhalb der API, über die Listenansichten der jeweiligen Ressourcen hin zu den Detailansichten der Ressourcen selbst.

Zusätzlich wird die Art, wie mit diesen Zugriffspunkten und dahinter liegenden Ressourcen interagiert werden kann, definiert. Dabei werden die Methoden des HTTP Protokolls (vgl. Abschnitt 2.2.2) verwendet, um eine einheitliche und standardisierte Definition der Interaktionsmöglichkeiten vornehmen zu können.

Das verwendete Datenformat, der Kommunikation mit den einzelnen Endpunkte, wird durch die, im HTTP Standard definierte, Content-Negotiation (vgl. Abschnitt 2.2.1) geregelt.

3.2.1. Spielerverzeichnis

Für den Zugriff auf alle Spieler wird der Listenendpunkt `/users/` definiert. Dieser ist vergleichbar zu einem Verzeichnis in einem Dateisystem.

Wird dieser Endpunkt mit der GET Methode aufgerufen, so werden alle Spieler, repräsentiert durch eine URI zur jeweiligen Detailansicht, zurückgegeben. Bei einer erfolgreichen Anfrage, wird der Statuscode 200, mit der jeweiligen Listenrepräsentation, an den Konsumenten gesendet. Bei einem Serverfehler wird der Statuscode 500 ausgeliefert. Dadurch wird der Konsument der API darauf hingewiesen, dass dieser es ggf. zu einem späteren Zeitpunkt erneut versuchen oder sich an den Betreiber des Webservices wenden sollte.

Um das Anlegen von Spielern zu ermöglichen, wird die POST Methode für den Endpunkt definiert. Dabei wird durch die Content-Negotiation ausgehandelt, wie die Details der Ressource übertragen werden sollen. Für das Anlegen eines Spielers müssen dabei zwei (vgl. Abbildung 4.2) Details angegeben werden. Der Name des Spielers und dessen Passwort. Beim Anlegen des Spielers, durch diese Methode, wird zudem die ID des Spielers, vom verarbeitenden Prozess auf dem Server, vergeben.

Nach einem erfolgreichen Anlegen eines Spielers antwortet der Server mit dem HTTP Statuscode 201[2], einer aktuellen, öffentlichen Repräsentation der angelegten Ressource, sowie der URI zu der Detailansicht, der angelegten Ressource, im HTTP `Location` Header.

Sollte beim Erstellen des Spielers ein Fehler aufgetreten sein, so wird im Falle eines Benutzungsfehlers des Konsumenten der Statuscode 409, mit einem entsprechendem Fehler, im ausgehandelten Format, zurückgegeben. Liegt der Fehler auf Seiten des Webservices, so wird der Statuscode 500 zurückgegeben.

Damit ein Konsument der API in der Lage ist, ohne externe Dokumentation, den Webservice zu bedienen, wird zuletzt die OPTIONS Methode für diesen Endpunkt definiert. Dieser enthält die Dokumentation der möglichen HTTP Methoden, deren

[2] Anhang A.2

Verwendung und möglicher Rückgaben. Dabei wird auch dieses Dokumentationsformat, durch die Content-Negotiation, ausgehandelt.

3.2.2. Spielerdetails

Damit ein Zugriff auf die Ressourcen, welche in der Listenansicht angelegt werden können, möglich ist, wird als nächstes der Endpunkt für die Detailansicht definiert. Durch diesen kann eine Ressourcen im Webservice identifiziert und adressiert werden.

Da die Spieler als einzigartige Identifikation eine ID besitzen, wird diese genutzt, um den Endpunkt zu beschreiben `/users/{id}`. Dabei ist `{id}` ein Platzhalter für eine beliebige Zahl.

Auch für diesen Endpunkt kann der Vergleich zu einem Dateisystem verwendet werden. Dabei entspricht dieser Endpunkt einer Datei, welche im Verzeichnis `/users/` abgelegt ist.

Soll der Inhalt der Datei angezeigt werden, so kann dies mit der GET Methode erreicht werden. Diese gibt für den Endpunkt die Detailansicht, bzw. die öffentlichen Ressourcendarstellung, zurück. Sofern eine Ressource an dem Endpunkt existiert wird diese, durch den Statuscode 200 in der Antwort und der Rückgabe der öffentlichen Ressourcenrepräsentation, dargestellt. Existiert die Ressource noch nicht im System, so wird der Statuscode 404 verwendet, um zu signalisieren, dass die Ressource nicht gefunden werden konnte. Bei einem Serverfehler wird der Statuscode 500 gesendet.

Um eine Ressource bearbeiten zu können werden die Methoden POST und PUT verwendet. Beide Methoden können die Felder der Ressource modifizieren. Analog zum Dateisystem entspricht ein POST dem Bearbeiten und Speichern einer Datei.

PUT hingegen entspricht dem Überschreiben einer Datei, welche ggf. einen komplett anderen Inhalt hat. Eine Besonderheit der PUT Methode ist, dass diese auch auf noch nicht existierende Spieler angewendet werden kann. Dadurch ist es möglich, dass der Konsument die ID des Spielers bestimmt und diese Vergabe nicht dem Server überlässt, wie es beim Erstellen durch einen POST, auf die Listenansicht, geschieht.

Wurde durch einen PUT der Spieler neu im System angelegt, so wird der Statuscode 201 verwendet. Wurde eine bestehende Ressource durch einen PUT überschrieben oder durch einen POST bearbeitet, so wird dies mit dem Statuscode 200 beantwortet. In beiden Fällen wird außerdem die öffentliche Repräsentation, der gespeicherten Ressourcen, zurückgegeben. Im Fehlerfall verhalten sich beide Methoden analog zum POST der Listenansicht.

Sollte beim Erstellen oder Ändern der Partie ein Fehler aufgetreten sein, so wird im Falle eines Benutzungsfehlers des Konsumenten der Statuscode 409 mit einem entsprechendem Fehler im ausgehandelten Format zurückgegeben. Liegt der Fehler auf Seiten des Webservices, so wird der Statuscode 500 zurückgegeben.

Bei der POST Methode muss weiterhin darauf geachtet werden, dass der Statuscode 404 zurückgegeben werden kann, wenn die Ressource nicht existiert. Dies liegt daran, dass POST die erhaltenen Daten verarbeitet und in die Repräsentation übernimmt. Nicht gesetzte Werte werden dabei nicht verändert. Bei PUT werden immer alle Werte der Ressource, ob gesetzt oder nicht, mit den erhaltenen Daten ersetzt.

Zuletzt kann ein Spieler, durch die DELETE Methode, gelöscht werden. Dabei wird im Erfolgsfall, der Löschaktion, der Statuscode 200 und die letzte bekannte öffentliche Repräsentation zurückgegeben. Eine Spielerressource kann nur gelöscht werden, wenn diese auch existiert. Entsprechend wird für den Fall, dass die Ressource nicht existiert ein 404 zurückgegeben. Im Fall der Löschaktion ist dies gleichzusetzen mit dem Statuscode 200, da die Intention bzw. das Ergebnis identisch mit dem erfolgreichen Löschen ist. Sollte die Ressource nicht gelöscht werden können, so wird der Statuscode 500 zurückgegeben.

Die Dokumentation, durch die OPTIONS Methode, ist analog zur Listenansicht.

3.2.3. Partieverzeichnis

Das Partieverzeichnis verhält sich analog zum Spielerverzeichnis. Einzig die verwaltete Ressource ist nicht ein Spieler, sondern eine Partie. Die URI des Endpunktes wird als `/matches/` definiert.

3.2.4. Partiedetails

Die Partiedetails verhalten sich, bis auf eine Ausnahme, analog zu den Spielerdetails. Die verwaltete Ressource ist nicht der Spieler, sondern eine Partie. Entsprechend ist die URI des Endpunktes als `/matches/{id}` definiert.

Zusätzlich wird die PATCH Methode für diesen Endpunkt definiert. Dabei wird der Medientyp `text/san`[3] für diese Methode akzeptiert. Dieser entspricht der elektronischen Schachnotation für einen Zug, innerhalb einer Partie. Der Zug wird der Liste, mit den einzelnen Zügen, der, durch die URI identifizierten, Partie hinzugefügt.

Ist der übergebene Zug in der aktuellen Partie gültig, so wird mit dem Statuscode 200 und der aktuellen auf dem Server hinterlegten, öffentlichen Repräsentation der Ressource, im ausgehandelten Format, geantwortet. Ist der Zug ungültig, so wird der Statuscode 409 mit entsprechender Fehlerbeschreibung, im ausgehandelten Format, zurückgegeben. Bei sonstigen Fehlern antwortet der Server mit dem Statuscode 500.

3.2.5. Startpunkt

Von diesem Endpunkt aus, soll der Konsument in der Lage sein die, in dieser Sektion definierten, Listenendpunkte, also das Spieler- und Partieverzeichnis, zu erreichen. Entsprechend gibt dieser Endpunkt bei einem erfolgreichen Aufruf durch einen Konsumenten den Statuscode 200 zurück. Der Inhalt der Antwort sind die URIs zu den Listenendpunkten. Die Antwort ist damit analog zu den Listenansichten bis auf den Punkt, dass URIs zu anderen Listenansichten und keine Detailansichten zu Ressource zurückgegeben werden.

Die Dokumentation dieses Endpunktes, anhand der OPTIONS Methode, beinhaltet allgemeine Informationen über den Webservice selbst.

3.2.6. Zusammenfassung

Durch die in diesem Abschnitt getroffenen Definitionen, werden die Punkte der Adressierbarkeit und der Zugriffsmöglichkeiten eines RESTful Webservices erfüllt (vgl. Abschnitt 2.3). Durch die Nutzung von standardisierten HTTP Methoden für

[3] siehe Anhang A.3.6

den Zugriff auf, durch URIs definierte Endpunkte, wird der Punkt der Standardkonformität (vgl. Abschnitt 2.3.2) erfüllt. Auch die Definition der PATCH Methode, im Kontext des entwickelten Webservices, wird der Standardkonformität gerecht.

Tabelle 3.1.: Übersicht der Zugriffsmöglichkeiten

Ressource	**Typ**	**URI**	**Unterstüzte HTTP Methoden**
	Listenansicht	/	GET
Spieler	Listenansicht	/users/	GET, POST, OPTIONS
Spieler	Detailansicht	/users/{id}	GET, PUT, POST, DELETE, OPTIONS
Partie	Listenansicht	/matches/	GET, POST, OPTIONS
Partie	Detailansicht	/matches/{id}	GET, PUT POST, PATCH, DELETE, OPTIONS

3.3. Strukturierung der öffentlichen Darstellung der Ressourcen

In diesem Abschnitt wird die Evolution des Datenformates beschrieben, die vom entstehenden Webservice zur Kommunikation verwendet werden soll. Dieses Datenformat wird abseits eines Formates wie der JavaScript Object Notation (JSON) oder der Extensible Markup Language (XML) definiert. Ziel ist es dabei, dass die Darstellung der Ressourcen, sofern eine entsprechende Abbildung ermöglicht wird, durch verschiedene Datenformate dargestellt werden kann.

Es werden dabei nur die öffentlichen Ressourcendarstellungen modelliert, die ein Konsument des Webservices, über die in Tabelle 3.1 definierten URIs, anfordern kann. Da der Startpunkt keine Ressource verwaltet, sondern nur einen Einstiegspunkt in die API darstellt, wird dieser nicht betrachtet.

Während der Entwicklung, der öffentlichen Datenrepräsentationen, wird zudem eine Bewertung der Formate, anhand folgender Eckpunkte, vorgenommen:

Die **Komplexität** definiert, wie einfach sich das Format implementieren und verwenden lässt. Das **Laufzeitverhalten** wird anhand der Anzahl der notwendigen Anfragen und der groben Dauer des Datenaustausches, zwischen Webservice und Konsument, beschrieben. Die **Cachability** gib an, wie gut sich das Format in einem Cache, einem lokalen oder zentralen Zwischenspeicher, abbilden lässt.

Für die Bewertung wird weiterhin ein Komplexbeispiel genutzt und in allen Formaten entsprechend der Eckpunkte bewertet. Das gewählte Beispiel orientiert sich dabei an einem theoretischen Schachprogramm, welches nach dem Start in der ersten Ansicht alle möglichen Partien und deren Kontrahenten anzeigt.

Die bestmögliche Bewertung wird durch ++ dargestellt. Das schlechteste Ergebnis einer Bewertung wird durch −− dargestellt. Ein ○ entspricht einer neutralen Bewertung.

Im ersten Format wird das Verfahren, wie die Bewertung vorgenommen wird, ausführlich erläutert. In den fortführenden Evolutionen der Formate werden nur die notwendigen Daten definiert und die Vergleichswerte errechnet, bzw. bei einem nicht numerischen Wert diskutiert.

3.3.1. Format 1

Format 1 kombiniert die, in 3.1 und 3.2, definierten Ressourcen und Endpunkte miteinander. Dabei werden sensitive Informationen aus der öffentlichen Darstellung entfernt. Bei den Ressourcen der Schach API ist das Passwort des Spielers die einzige sensitive Information. Dieses wird daher niemals bei einer Anfrage mit zurückgegeben, sondern nur für serverseitige Zwecke (z.B. Authentifizieren/Autorisieren) verwendet.

Die Verweise, innerhalb einer Ressource, auf eine andere Ressource werden durch die einzigartige URI, der jeweiligen Ressource, dargestellt. Dies ermöglicht es in einer Partie auf einen existierenden Spieler, innerhalb des Webservices, zeigen zu können. Das Vorgehen ist dabei vergleichbar zu einem Fremdschlüssel in einem Database-Management-System (DBMS). Der Webservice muss dabei sicherstellen, dass wenn eine Ressource gelöscht wird, die Referenz in allen existierenden Ressourcen entfernt wird. Dadurch soll gewährleistet werden, dass die API nur gültige URIs an einen Konsumenten zurück gibt. Gleiches gilt für das Hinzufügen von Verweisen innerhalb einer Ressource.

Die vom Konsumenten angegebenen Verweise sollen dabei auf Ihre Gültigkeit geprüft und bei nicht bestehen dieser Prüfung vom Server, durch einen Fehler, abgewiesen werden. Dies kann dabei durch den Statuscode 409, welcher bereits für die erstellenden und verändernden Prozesse auf die verschiedenen Ansichten definiert wurde (vgl. Abschnitt 3.2), geschehen.

Die Kombination der oben genannten Punkte ergibt das in Tabelle 3.2 dargestellte öffentliche Format für die jeweiligen Ressourcenendpunkte.

Tabelle 3.2.: Ressourcendarstellung: Format 1

URI	Darstellung
/users/	- URI zum 1. Spieler - URI zum 2. Spieler - ... - URI zum n-ten Spieler
/users/{id}	- Name des durch {id} identifizierten Spielers
/matches/	- URI zur 1. Partie - URI zur 2. Partie - ... - URI zur n-ten Partie
/matches/{id}	- URI zum weißen Spieler - URI zum schwarzen Spieler - Startsituation - Züge

Die Komplexität, um die erste Version zu verwenden, ist sehr gering und einfach zu beschreiben. Für jede Ressource gibt es zwei Endpunkte. Die Listenansicht und die Detailansicht (vgl. Abschnitt 3.2). Über die Listenansicht können die jeweils existierenden Ressourcen angezeigt werden und deren URI, welcher auf die Detailansicht der jeweiligen Ressource zeigt, ermöglicht es auf die Informationen der Ressource zugreifen zu können. Durch die getroffene Definition, dass Subressourcen durch einen URI zur Detailansicht dargestellt werden, können dadurch alle Informationen durch dieses Format abgebildet werden.

Der Zugriff auf eine Ressource erfolgt dabei immer über eine eigenständige Anfrage an den Webservice. In dessen Antwort sind die kompletten Informationen der angeforderten Ressource enthalten.

Um einen Vergleich zu ermöglichen wird nun das Komplexbeispiel betrachtet.

Zuerst wird die Anzahl der notwendigen Anfragen betrachtet. Für jede Partie muss die Detailansicht geladen werden, damit die URIs der Kontrahenten einer Partie ermittelt werden können. Mit diesen Verweisen müssen dann die Detailansichten der Spieler geladen werden, um deren Namen auslesen zu können. Daraus geht hervor, dass die Anzahl der notwendigen Anfragen durch die im System existierenden Partien und deren Kontrahenten bestimmt wird.

Wird nun die Anzahl der Partien durch n und die Anzahl der abzufragenden Spieler durch $n \cdot 2$ beschrieben, so ergibt sich die Formel $n+n \cdot 2$. Um die URIs der Partien zu erhalten wird zudem die initiale Anfrage auf die Listenansicht der Partien benötigt: $n + n \cdot 2 + 1$.

Bei einer Verallgemeinerung diese Formel entspricht die Anzahl der Partien der Anzahl der Ressourcen, dargestellt durch n. Die in einer Ressource existierenden Subressourcen, im Beispiel die Kontrahenten, werden nun allgemein durch m dargestellt. Mit der initialen Anfrage ergibt sich daraus folgende Formel:

$$\text{Anzahl der Anfragen} = n + m + 1 \tag{3.1}$$

$$\text{Anzahl der Anfragen} = n + (n \cdot 2) + 1 \tag{3.2}$$

Wird das Beispiel auf 50 Partien konkretisiert dann folgt daraus, dass 151 Anfragen an das System gesendet werden müssen.

$$\text{Anzahl der Anfragen} = 50 + (50 \cdot 2) + 1 \tag{3.3}$$

$$\text{Anzahl der Anfragen} = 151 \tag{3.4}$$

Beim Betrachten der Gesamtlaufzeit wird davon ausgegangen, dass eine HTTP1.1 Verbindung zum Webservice besteht, welche so konfiguriert ist, dass Anfragen über diese Verbindung sequentiell erfolgen müssen und die Verbindung nach der ersten Kommunikation bestehen beleibt. Dadurch wird im folgenden nur die Gesamtlaufzeit betrachtet, welche von der Anzahl der Abfragen und der Verzögerung zwischen Anfrage und Antwort, der Round-Trip-Time (RTT), der aktuellen Verbindung abhängig ist. Aufgrund der heutigen Übertragungsgeschwindigkeiten, auch im mobilen Sektor, wird die Dauer der Übertragung in der Gesamtlaufzeit nicht betrachtet.

Bei einer durchschnittlichen Verzögerung von 55ms (vgl. [Wik15b]) kann folgende Gesamtlaufzeit errechnet werden:

$$\text{Gesamtlaufzeit} = \text{RTT} \cdot \text{Anzahl der Anfragen} \tag{3.5}$$

$$\text{Gesamtlaufzeit} = 55ms \cdot 151 \tag{3.6}$$

$$\text{Gesamtlaufzeit} = 8305ms \approx 8,31s \tag{3.7}$$

Usability Studien (siehe [Sch03] und Kapitel fünf in [Nie93]) haben ergeben, dass eine Zeit zwischen 2 und 4 Sekunden vom Nutzer bereits als aktive Ladezeit wahrgenommen wird, wobei 8 bis 10 Sekunden den Nutzer bereits bewegen würden einer anderen Tätigkeit aktiv nachzugehen.

Die Gesamtlaufzeit kann in diesem Beispiel verringert werden, indem von den sequentiellen Anfragen auf, beispielsweise 8, parallele Anfragen gewechselt wird und somit die vom Nutzer empfundene Ladezeit auf eine Sekunde reduziert werden kann.

Wird das Beispiel jedoch auf über 100 Partien oder wesentlich komplexere Datenstrukturen skaliert, kann dies nicht mehr effizient durch eine erhöhte Anzahl an gleichzeitigen Anfragen ausgeglichen werden. Weiterhin wurde im Beispiel von einer stabilen DSL 2000 Verbindung ausgegangen. Im mobilen Sektor könnte die Verzögerung in der Kommunikation, je nach Netzausbau, weiter steigen und damit die Gesamtlaufzeit erhöhen.

Zuletzt muss noch beachtet werden, dass aufgrund der 1:1 Relation von Anfrage zu Ressource, diese direkt in einem Cache abgebildet werden können. Entsprechend müsste eine Anwendung nach einer möglichen Initialisierung eines Cachesystems nur noch die benötigten Ressourcen nachladen. Das kann wiederum die nötigen Anfragen an den Server reduzieren und damit die Gesamtlaufzeit optimieren. Für das Prüfen der Aktualität des Caches müssten jedoch wieder entsprechend viele Anfragen, wie Cache-Einträge existieren, an den Server gesendet werden.

Zusammengefasst: Durch die einfache Konstruktion des Formates ist die API einfach zu implementieren und zu benutzen. Subressourcen werden über eine eigene Anfrage vom Webservice nachgeladen. Eine Antwort des Webservice beinhaltet immer nur die von der Ressource spezifizierten Informationen und Verweise auf weitere Subressourcen. Dies ist bei komplexen Datenstrukturen oder bei Auflistungen einer der größten Nachteile. Das liegt daran, dass die Anzahl der Anfragen mit den zur

Verfügung stehenden Ressourcen, Subressourcen und abzufragenden Informationen skaliert. Ein wenig kann dabei durch das Cachen der Anfragen entgegenwirkt werden, jedoch ist dies nur bis zu einem bestimmten Punkt sinnvoll einsetzbar, da das initiale Bauen des Caches mindestens einmalig die, dem Format zugrundeliegende, Gesamtlaufzeit benötigt.

Tabelle 3.3.: Bewertung Format 1

Kriterium	**Bewertung**
Komplexität	++
Laufzeitverhalten	−−
Cacheability	++

3.3.2. Format 2

Einer der größten Nachteile des vorherigen Formates, ist die Anzahl der notwendigen Anfragen. Diese Formatevolution wird versuchen, die Anzahl der Anfragen zu reduzieren.

Um eine Reduktion der Anfragen zu ermöglichen, müsste der Webservice wissen, welche zusätzlichen Informationen übertragen werden sollen. Dies kann erreicht werden, indem die, in Abschnitt (vgl. Abschnitt 3.2) definierten URIs um Abfrageparameter erweitert werden. Diese werden vom Server ausgewertet, wobei die gewünschten Informationen mit in die Antwort aufgenommen werden.

Als Parameter wird für diese API `?embed-*` gewählt. Dabei steht $\star$ für ein Feld, welches einen Verweis auf eine Subressource definiert. In der Detailansicht einer Ressource können dadurch Subressourcen eingebunden werden. Wird eine Ressource über einen Parameter eingebunden, so wird dessen URI durch die Detailrepräsentation ersetzt.

Durch diese Definitionen ist es möglich alle Subressourcen in einer Anfrage einbinden zu können. Bei den Listenansichten gibt es jedoch noch keine Möglichkeit, die eigene Ressource direkt zu laden, da die obige Definition einen Feldnamen voraussetzt. Damit diese Erweiterung auch für Listenansichten verwendbar ist, wird der Parameter `?embed` eingeführt. Dieser spezifiziert kein Feld und ersetzt daher die URIs zu den Detailansichten mit deren Repräsentationen. Um weitere Sonderfälle zu vermeiden wird weiterhin definiert, dass bei einem `?embed-*` der Abfrageparameter `?embed` implizit angenommen wird.

Damit die eingebundenen Ressourcen dargestellt werden können, wird die Rückgabe um ein assoziatives Array oder eine gleichwertige Datenstruktur erweitert. Dadurch können die Ressourcen ineinander verschachteln werden.

Ein assoziatives Array wird durch `[]` beschrieben und beinhaltet die öffentliche Datenrepräsentation der Ressource.

Aus der Erweiterung ergibt sich das, in Tabelle 3.4 dargestellte, öffentliche Format für die jeweils betroffenen Ressourcenendpunkte. Für die Listendarstellung wird Beispielhaft jeweils nur das n-te Element angegeben.

Tabelle 3.4.: Ressourcendarstellung: Format 2

URI	Darstellung
/users/?embed	- [Name des n-ten Spielers] - ...

/matches/
?embed-weiß&embed-schwarz
	- [
	  - Weiß: [
	  - Name des weißen Spielers
	  ]
	  - Schwarz: [
	  - Name des schwarzen Spielers
	  ]
	  - Startsituation der n-ten Partie
	  - Züge der n-ten Partie
	]
	- ...

/matches/{id}
?embed-weiß&embed-schwarz
	- Weiß: [
	  - Name des weißen Spielers
	]
	- Schwarz: [
	  - Name des schwarzen Spielers
	]
	- Startsituation
	- Züge

Da `/users/{id}` keine Subressourcen enthält entspricht dieser Endpunkt dem ersten Format.

Durch die Erweiterung der API, um einen weiteren Datentypen, steigt die Komplexität weiter an. Durch die zur Verfügung stehenden Abfrageparameter können unbedingte Selektionen in der API durchgeführt werden. Dadurch können alle vom System verwalteten Ressourcen mit einer Anfrage angefordert werden. Bei komple-

xeren Datenstrukturen, mit mehreren Verweisen auf Subressourcen, könnte jedoch eine hohe Schachtelungstiefe der Ressourcen auftreten.

Jedoch ist dieses Format nun nicht mehr Abhängig von der Anzahl der Anfragen. Die Anzahl kann, je nachdem welche Informationen selektiert werden, auf 1 reduziert werden.

Werden diese Erkenntnisse auf das Beispiel der 50 Partien angewendet (vgl. Format 1), können folgende Datenpunkte definiert und errechnet werden:

$$\text{Anzahl der Anfragen} = 1 \tag{3.8}$$

$$\text{Gesamtlaufzeit} = \text{RTT} \cdot \text{Anzahl der Anfragen} \tag{3.9}$$

$$\text{Gesamtlaufzeit} = 55ms \cdot 1 \tag{3.10}$$

$$\text{Gesamtlaufzeit} = 55ms \approx 0,06s \tag{3.11}$$

Die Anzahl der Anfragen ist in dieser Formatdefinition unwichtig, da nur eine benötigt wird. Damit ist allein die Verzögerung zwischen dem Konsumenten und dem Webservice entscheidend, also wie schnell die Informationen abgefragt werden können. Je mehr Daten über die Parameter selektiert werden, umso größer wird das zu übertragende Datenpaket. Entsprechend würde sich, bei weitaus komplexeren Datentypen, eine Analyse der Übertragungsdauer lohnen. Für den in dieser Arbeit behandelten Schach Webservice lohnt sich diese Analyse nicht, da sich die zu übertragenden Daten im Kilobyte-Bereich bewegen, welche über heutigen Datenverbindungen schnell übertragen werden können.

Aufgrund der monolithischen Anfragen, bzw. Antworten lassen sich diese jedoch schlecht in einem Cache abbilden. Das liegt daran, dass wenn sich eine Ressource im System ändert, das komplette Datenpaket erneut und komplett angefragt werden müsste. Die Validität eines einzelnen Cache-Eintrages ist nicht gegeben, da dieser mehre Subressourcen beinhalten kann. Das Speichern der Subressourcen im Cache ist zudem nicht separat möglich, da die URI beim Einbinden verloren geht. Weiterhin wird ein Spieler, der an mehr als nur einer Partie teilnimmt, in den jeweiligen Partiedarstellungen immer erneut eingebunden. Entsprechend müssten, wenn ein

Spieler seinen Namen ändert, alle Partien invalidiert werden. Daraus folgt, dass sich ein sinnvoller Cache nur schwer, wenn überhaupt, aufbauen lassen würde.

Zusammenfassung: Dieses Format löst das Problem der Abhängigkeit der Gesamtlaufzeit, indem die Anzahl der Anfragen auf 1 reduziert werden kann. Daher bestimmt die Komplexität, die Menge der angefragten Informationen und die resultierende Gesamtgröße einer Antwort auf eine Anfrage, die Laufzeit des Formates. Aufgrund der dadurch entstanden spezifischen Anfragen und den daraus resultierenden komplexen Antworten wird die Möglichkeit, einen Cache sinnvoll implementieren zu können, minimiert. Entsprechend wird durch dieses Format einer der größten Vorteile eines REST Webservices außer Kraft gesetzt. Zudem steigt die Komplexität des Formates durch die Abfrageparameter und der daraus resultierenden Ressourcenschachtelung.

Tabelle 3.5.: Bewertung Format 2

Kriterium	**Bewertung**
Komplexität	–
Laufzeitverhalten	++
Cacheability	––

3.3.3. Format 3

In der letzten Formatevolution wird die Datenstruktur aus dem vorherigen Format so verändert, dass ein vergleichbares Format entsteht, welches jedoch die Cacheability der REST Architektur besser unterstützt.

Um die Cacheability wiederherzustellen, muss ein Format definiert werden, welches es ermöglicht, die zusätzlichen Informationen sinnvoll zu in einem Cache abzulegen, ohne dabei von den Abfrageparametern abhängig zu sein. Dabei soll jedoch die Flexibilität der Parameter erhalten bleiben, bzw. die Anzahl der Anfragen weiterhin so gering wie möglich gehalten werden. Die grundlegende Idee dieses Formates wurde durch einen RFC Draft, welcher das JSON+HAL Format[4] definiert, inspiriert.

Die beste Cacheability hatte das erste definierte Format, da dies die Ressourcendarstellungen 1:1 auf die URIs abbilden konnte. Diese Abbildung wird mit den Definitionen aus Format 2 verschmolzen.

[4] `http://tools.ietf.org/html/draft-kelly-json-hal-07`

Tabelle 3.6.: Ressourcendarstellung: Format 3 - Spielerverzeichnis

URI	Darstellung
/users/?embed	- Link zum n-ten Spieler - ... - _embedded: ['Link zum n-ten Spieler': [- Name], ...] - ...

Tabelle 3.7.: Ressourcendarstellung: Format 3 - Partieverzeichnis

URI	Darstellung
/matches/ ?embed-weiß &embed-schwarz	- URI zum n-ten Spiel - ... - _embedded: ['URI zur n-ten Partie': [- URI zum weißen Spieler - URI zum schwarzen Spieler - Startsituation - Züge], 'URI zum weißen Spieler der n-ten Partie': [- Name], 'URI zum schwarzen Spieler der n-ten Partie': [- Name], ...]

Um dies zu ermöglichen und gleichzeitig die Komplexität zu minimieren werden Beschränkung definiert. Die Ressourcendarstellung, so wie diese in Format 1 definiert ist, muss dabei immer eingehalten werden. Entsprechend dürfen die Verweise auf Subressourcen nicht durch deren Repräsentation ersetzt werden. Damit diese Subressourcen dennoch eingebunden werden können, wird ein Feld definiert, welches alle Darstellungen der eingefügten Subressourcen anhand Ihrer jeweiligen URI enthält. Dieses Feld wird dabei als `_embedded` definiert. Der Name des Feldes ist nicht bindend, sondern soll nur darauf hinweisen, dass alle in diesem Feld abgelegten Werte einer Subressource entsprechen. Eine Implementierung in einem spezifischem Datenformat kann hierbei eigene Namensdefinitionen treffen.

Beispiele der Formatdefinition sind den Tabellen 3.6, 3.7 und 3.8 zu entnehmen.

Tabelle 3.8.: Ressourcendarstellung: Format 3 - Partiedetails

URI	Darstellung
/matches/{id} ?embed-weiß&embed-schwarz	- Weiß: URI zum weißen Spieler - Schwarz: URI zum schwarzen Spieler - Startsituation - Züge - _embedded: ['URI zum weißen Spieler': [- Name], 'URI zum schwarzen Spieler': [- Name]]

Aufgrund der einfachen Datenstruktur und das Fehlen von Subressourcen entspricht die Darstellung der Detailansicht des Spielers dem ersten Format.

Durch den Umbau der Struktur, auf das Einfügen der Informationen durch ein separates Feld, wurden die Vorteile der Formate 1 und 2 kombiniert.

Die Komplexität der Verwendung der API sinkt im Vergleich zum zweiten Format, da eine endlose Schachtelung komplexer Strukturen nicht mehr möglich ist. Durch das `_embedded` Feld wurde dabei die Schachtelungstiefe für eingebettete Ressourcen auf 1 gesenkt. Das Format ist jedoch, aufgrund der Abfrageparameter, Komplexer als das erste Format.

Im Format 2 wurde die Anzahl der theoretisch notwendigen Anfragen auf 1 reduziert, diese Eigenschaft und die daraus resultierende Laufzeit wird von der aktuellen Formatevolution beibehalten. Weiterhin ist eine Dopplung von Daten in diesem Format innerhalb einer Anfrage ausgeschlossen, da die Subressourcen, egal wie oft diese Referenziert werden, anhand ihrer einzigartigen URI eingebunden werden. Eine Schachtelung von Ressourcen wird dadurch ebenfalls verhindert.

Die Möglichkeit einzelne Ressourcen zu Cachen und zu Invalidieren ist vergleichbar zum ersten Format, da in dieser Formatversion die URI aus dem `_embedded` Feld als identifizierender Wert für die Ressource genommen werden kann. Entsprechend lässt sich auch jede Ressource anhand dieser URI eigenständig überprüfen, invalidieren und erneut Anfragen.

Zusammenfassung: Das dritte und letzte Format kombiniert die einfache Verwendung des ersten und die komplexen Abfragen und somit die Verminderung der not-

wendigen Anfragen des zweiten Formates. Dabei wird durch das Definieren eines Feldes für die, durch die Abfrageparameter eingebundenen, Ressourcen nicht nur die Komplexität verringert, sondern gleichzeitig das Cacheproblem gelöst.

Tabelle 3.9.: Bewertung Format 3

Kriterium	**Bewertung**
Komplexität	+
Laufzeitverhalten	++
Cacheability	+

3.3.4. Übersicht der Formate

Tabelle 3.10.: Übersicht der Darstellungsformate

Kriterium	**Format 1**	**Format 2**	**Format 3**
Komplexität	++	−	+
Laufzeitverhalten	−−	++	++
Cacheability	++	−−	+

3.4. Erweiterung der Ressourcendarstellung um HATEOAS

Im Abschnitt 2.3.6 wird die Fähigkeit einer REST API sich selbst beschreiben zu können definiert. Dabei wird neben der Eigendokumentation der API, welche innerhalb der Schach API durch die OPTIONS Methode gewährleistet wird, auch das Definieren der möglichen Übergänge innerhalb der API von einem zum nächsten Endpunkt gefordert.

Damit die API dazu in der Lage ist, müssen die möglichen Transitionen vom aktuellen Endpunk zu allen adjazenten Endpunkten definiert werden. Im Abschnitt 3.2 wurden zum Verständnis unterschiedliche Vergleiche zu einem Dateisystem herangezogen.

Auch bei den Transitionen innerhalb der API kann dieser Vergleich wiederverwendet werden. Wird von der obersten Ebene eines Verzeichnisbaumes ausgegangen, so ist der Ausgangspunkt in der API definiert als `/`, dem Startpunkt des Webservices. Die erste Ebene der Endpunkte, die Listenansichten, entsprechen dabei den Verzeichnissen `/users/` und `/matches/`. Damit von der oberen Ebene in diese Verzeichnisse

gewechselt werden kann, müssen diese bekannt sein. Um dies zu ermöglichen müssen diese URIs in der obersten Ebene mit angezeigt werden.

In einem Verzeichnis werden, nach der Grunddefinition der Listenansichten, bereits die Dateien, also die jeweiligen Detailansichten, der API mit ihren URIs `/users/{id}` und `/matches/{id}` ausgegeben. Von diesem Endpunkt wäre es jedoch nun nicht mehr möglich durch eine Antwort des Webservices zurück zur obersten Ebene zu gelangen. Damit auch dies möglich ist, wird die jeweils übergeordnete Ebene, innerhalb der Antwort, anhand der URI angegeben.

Allgemeiner kann definiert werden, dass in einer Antwort des Webservices alle Möglichkeiten, die eine Ebene tiefer und eine Ebene höher liegen angezeigt werden sollen. Dadurch wird es einem Konsumenten ermöglicht, sich innerhalb der API anhand von Transitionen zu bewegen. Bei komplexeren Datenstrukturen können auch tiefere Ebenen oder Verbindungen in die Antwort mit eingebettet werden. So könnte ein Webservice der beispielsweise Bestellungen verwaltet, innerhalb einer Kundenrepräsentation auf die Übersicht aller, für diesen Kunden getätigten, Bestellungen verweisen.

Weiterhin sollte eine Ressource, welche durch die API zurückgegeben wird, auf sich selbst Verlinken. Dies erleichtert den erneuten Zugriff auf diese Ressource und definiert zudem eine einheitliche Möglichkeit an die URI einer Ressource innerhalb einer Antwort zu gelangen.

Die URIs werden in die Ressourcenrepräsentation mit aufgenommen und können so von einem Konsumenten ausgelesen und verarbeitet werden. Entsprechend erweitert sich das gewählte Format, um die URIs auf die jeweiligen Ressourcen selbst.

Die Verweise auf die Detailansichten, können dabei mit Ihren Platzhaltern, für die ID der Ressource, in die Liste aufgenommen werden. Zweck ist es, dass z.B. ein Konsument die URI für eine PUT Anfrage verwenden kann und den Platzhalter, durch einem selbst gewählten Wert, ersetzen kann.

In der Detailansicht eines Ressourcenendpunktes, beziehen sich die Eigen-Links immer auf die Ressource selbst. Wird eine Ressource als Subressource durch einen Abfrageparameter eingebunden, so sollten diese URIs erhalten bleiben. Dabei könnten alle Verweise, welche sich auf diese Ressource beziehen, also in einer tieferen Ebene liegen, auch erhalten bleiben. Diese Entscheidung, der Einbindung tieferer

Tabelle 3.11.: Ressourcendarstellung: HATEOAS

URI	Erweiterung
/	- / - /users/ - /matches/
/users/	- /users/ - / - /users/{id}
/users/{id}	- /users/{id} - /users/
/matches/	- /matches/ - / - /matches/{id}
/matches/{id}	- /matches/{id} - /matches/

Ebenen, obliegt den genauen Implementierungsdetails der API. Für eine einheitlichere Ressourcendarstellung wäre dies jedoch Vorteilhaft.

3.5. Zugriffsschutz des Webservices

Der Zugriffsschutz, im Kontext eines Webservices, bezieht sich auf die möglichen Interaktionen mit Ressourcen, durch einen Konsumenten. Dieser Schutz umfasst die Themen der Authentifizierung eines Konsumenten gegenüber der API, durch eindeutige Merkmale und die Autorisierung. Dabei definiert die Autorisierung die möglichen Interaktionen, welche ein authentifizierter Konsument, innerhalb des Webservices anwenden darf.

3.5.1. Keine Authentifizierung

Eine Möglichkeit des Zugriffsschutzes, ist es diesen nicht durch eine Authentifizierung zu schützen. Eine Alternative der Authentifizierung könnte darin bestehen, dass nur vertrauenswürdige Konsumenten, anhand ihrer netzwerkarchitektonischen Merkmale den Zugriff auf den Webservice erhalten. Dazu könne z.B. die Internet Protocol (IP) oder Media Access Control (MAC) Adresse des Konsumenten verwendet werden. Diese Konsumenten können dann alle Ressourcen, ohne Limitierungen: Erstellen, Verändern und Löschen.

Diese Methode des Zugriffsschutzes eignet sich daher nur für Webservices, welche sich an eine sehr begrenzte Anzahl von Konsumenten richtet, die vorzugsweise unter der Kontrolle des Betreibers, des Webservices, stehen.

3.5.2. HTTP - Basic Auth

Das HTTP Protokoll definiert eine Möglichkeit sich gegenüber einem Server Authentifizieren zu können. Das Verfahren wird in [Fra99] beschrieben.

Wird ein geschützter Endpunkt von einem Konsumenten aufgerufen, so wird den HTTP Header `WWW-Authenticate` darauf hingewiesen, dass eine Authentifizierung an diesem Endpunkt erfolgen muss. Diese Authentifizierung benötigt einen Benutzernamen und ein Passwort für die Identifikation. Die Informationen werden mit einem `:` getrennt, kodiert[5] und in einer erneuten Anfrage der Ressource im HTTP Header `Authorization` übertragen. Damit der Server feststellen kann, um welche

[5]Siehe [Fra99] für die Art der Kodierung

Form der Authentifizierung es sich handelt, wird ein `Basic` vor die kodierten Daten geschrieben.

Ein komplettes Beispiel für `Spieler 1:Geheim` würde in folgender Definition resultieren:

```
Authorization: Basic U3BpZWxlciAxOkdlaGVpbQ==
```

In dem Prozess der Authentifizierung wird der Benutzername und das Passwort an den Webservice übergeben. Diese werden mit den vom Webservice gespeicherten Daten vergleichen. Stimmen diese überein wird der Konsument gegenüber dem System authentifiziert. Der Benutzername entspricht dabei dem Namen eines Spielers und das Passwort entspricht dem, bei der Erstellung angegebenen, Passwort des Spielers.

Da die Informationen zur Identifikation des Benutzers im Klartext übertragen werden, sollte dieses Verfahren nur über eine gesicherte Verbindung, wie z.B. HTTPS verwendet werden. Sonst könnten diese Informationen von einem Dritten mitgelesen werden. Dieser könnte sich daraufhin unrechtmäßig gegenüber dem Webservice authentifizieren.

3.5.3. Token basierte Authentifizierung

Eine Token basierte Authentifizierung nutzt für das Identifizieren eines zugriffsberechtigten Konsumenten einen Token, der beispielsweise aus einer, zwischen dem Server und Konsumenten ausgehandelten, Zeichenkette bestehen kann. Ein Beispiel für eine solche Authentifizierung ist im OAuth Standard [E H10] spezifiziert. Dabei wird von einem Webservice ein Token generiert, mit welchem ein Konsument sich gegenüber der geschützten Endpunkte des Webservices authentifizieren kann.

Um das Token zuordnen zu können, muss dieses an einem Benutzer gespeichert werden. Im Fall der Schach API ist ein möglicher Benutzer für die Authentifizierung ein Spieler. Für diesen könnte das im Abschnitt 3.1 definierte `Passwort` Feld für das Token verwendet werden. Sobald der Konsument auf einen geschützten Endpunkt, im Kontext des Spielers zugreifen möchte, muss dieser das Token bei einer Anfrage mit übertragen. Wird ein Spieler gefunden, dem das Token zugeordnet werden kann, so wird dieser, für den Kontext der Anfrage, authentifiziert.

Die Methode ist für größere Webservices geeignet, da Tokens beliebig erteilt und zurückgezogen werden können. Zudem könnten einzelne Token verschiedene Autorisierungen bzw. Rechte gegenüber den existierenden Endpunkte besitzen, sofern diese Granularität von dem Webservice unterstützt wird.

Auch bei dieser Authentifizierung werden die sensitiven Daten, das Token, im Klartext übertragen. Entsprechend sollte eine geschützte Verbindung zum Austausch des Token verwendet werden.

3.5.4. Autorisierung

Nachdem der Konsument sich mit einem Verfahren authentifiziert hat, muss noch festgelegt werden, auf welche Ressourcen dieser zugreifen darf und was passieren soll, wenn der Konsument keine Berechtigung auf eine Ressource hat. Die Prüfung der Berechtigungen setzt eine vorherige, erfolgreiche Authentifizierung voraus. Entsprechend sollten nicht authentifizierte Benutzer darauf hingewiesen werden, dass diese sich, bevor Sie diesen Endpunkt verwenden können, am Webservice authentifizieren müssen.

Sollte ein Konsument noch nicht authentifiziert sein und auf einen geschützten Endpunkt zugreifen, so antwortet die API mit einem Statuscode 401, um darauf hinzuweisen, dass der Konsument sich erst Authentifizieren muss, bevor der Zugriff, unter Umständen, gewährt werden kann.

Liegt eine Authentifizierung vor, der Konsument darf aber nicht auf den Endpunkt zugreifen, da der, hinter der Authentifizierung stehende, Spieler keine Zugriffsberechtigung für den geschützten Endpunkt besitzt, so muss der Webservice mit einem Statuscode 403 antworten und die Anfrage verwerfen.

3.5.5. Schützenswerte Endpunkte

In der Schach API gibt es 2 Endpunkte, welche geschützt werden sollten, damit die darin enthaltenen Informationen nicht beliebig von Dritten verändert werden können.

Die Detailinformationen eines Spielers sollten nur dann verändert werden können, wenn der Konsument gegenüber dem Webservice als der Spieler authentifiziert wurde, dessen Informationen geändert werden sollen. Entsprechend soll z.B. Spieler 1

nicht in der Lage sein die Daten von Spieler 2 zu verändern. Jedoch darf ein authentifizierter Spieler seine eigenen Detailinformationen beliebig verändern. Dazu zählen die PUT, POST und DELETE Methode der Detailansicht der Spieler.

Weiterhin wird es nur den in einer Partie eingetragenen Spielern gestattet neue Züge zu dieser hinzuzufügen. Entsprechend werden diese Spieler gegenüber der Detailansicht einer Partie, in welcher diese eingetragen sind, dazu Autorisiert die PATCH Methode zu verwenden.

Sollte in einer Partie noch ein Spieler fehlen, so kann sich ein Spieler dieser hinzufügen. Dabei darf für diesen Fall ein Spieler die Partie durch einen POST verändern. Jedoch muss durch eine Validierung, bzw. Autorisierung, sichergestellt werden, dass nur das Feld des schwarzen, respektive weißen Spielers verändert wird. Weiteres Bearbeiten, Ersetzen oder Löschen einer Partie wird einer verwaltenden Entität überlassen.

Diese verwaltende Entität kann bisher nicht in der Schach API abgebildet werden. Die einfachste Möglichkeit diese abzubilden würde z.B. darin bestehen, dass ein zusätzliches Feld eingeführt wird. Das zusätzliche Feld könnte dabei eine beliebig geartete Information, beispielsweise einen boolschen Wert, beinhalten. Anhand des eingeführten Feldes wäre der Webservice in der Lage einen Spieler als verwaltende Entität identifizieren zu können.

Alle Methoden und Endpunkte, die in dieser Sektion nicht erwähnt wurden, können ohne vorherige Authentifizierung oder Autorisierung verwendet werden.

3.5.6. Zusammenfassung

Da der Webservice, entsprechend der REST Prinzipien, zustandslos ist muss beachtet werden, dass die Authentifizierung und auch die Autorisierung bei jedem Aufruf einer (geschützten) Ressource stattfinden muss.

Entsprechend der zugrundeliegenden Anforderung des Webservices kann zwischen verschiedenen Authentifizierungsverfahren gewählt werden. Dabei sollte darauf geachtet werden, dass die Authentifizierungsdaten nach Möglichkeit nicht von Dritten abgefangen werden können.

Die vorgestellten Verfahren zur Authentifizierung setzen, aufgrund der Möglichkeit dass die Authentifizierungsinformationen abgefangen werden könnten, eine gesicherte Verbindung voraus.

4. Implementierung

Im vorherigen Kapitel wurde die Schach API beschrieben und mehrere Designentscheidungen vorgestellt. Weiterhin wurden verschiedene Lösungsansätze vorgeschlagen, welche in der folgenden prototypischen Implementierung umgesetzt werden.

Die Implementierung basiert auf dem PHP–Framework Silex[6], welches eine minimale Auswahl an Funktionen bietet um eine, auf HTTP basierende, API zu entwickeln. Dabei wird neben den verschieden HTTP Methoden auch die Content-Negotiation unterstützt.

Für die Umsetzung des Prototyps wurde sich für das dritte Format entschieden, da dies die meisten Vorteile gegenüber der anderen Formate bietet. Das Format wurde dabei auf die Datenformate `text/xml` und `application/json` abgebildet.

4.1. Struktur

Die Struktur des Prototypen orientiert sich am Model-View-Controller (MVC) Pattern. Das Model entspricht dabei der intern verwendeten Datenstruktur. Der View ist die, für den Benutzer sichtbare Schicht. Dabei enthält der View nahezu keine Logik, außer diese dient der formatierten Ausgabe der, vom Controller übergebenen, Daten. Der Controller ist das Bindeglied zwischen dem Model und dem View. Zudem verarbeitet der Controller die Anfragen (Requests) des Konsumenten und gibt diesem die Antwort (Response) zurück.

Alle benötigten Klassen werden durch Dependency Injection (DI) innerhalb der Anwendung zugänglich gemacht. Dabei dient das Applikationsobjekt als DI-Container. In diesem werden beim Aufrufen der Applikation alle benötigten Dienste Initialisiert. Das Registrieren, der im System existierenden Dienste, übernimmt dabei ein ServiceProvider, genauer der ApiServiceProvider.

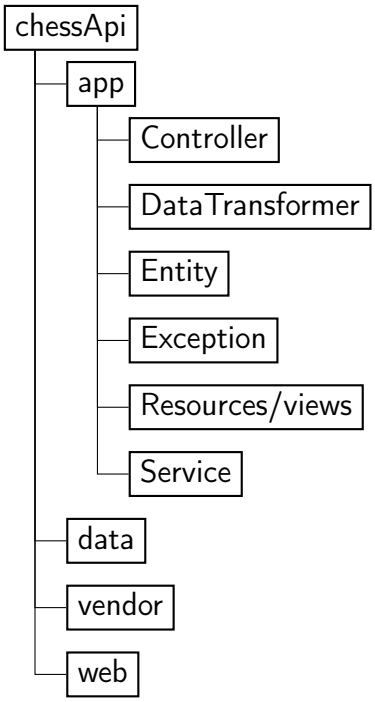
Der ApiServiceProvider registriert unter Anderem den ControllerProvider. Dieser ist dafür zuständig, die Methoden der Controller den Routen zuzuweisen. Eine Route besteht mindestens aus einer URI, einer gültigen HTTP Methode und einem Controller-Methode-Tupel. Stimmt eine Anfrage mit einer Route überein, so wird

[6] `http://silex.sensiolabs.org/`

die Methode des Controllers aufgerufen. Neben den Routen werden zudem Event-Listener registriert. Diese können bei, vom Silex Framework festgelegten, Events einen Request gesondert behandeln oder einen Response eines Controllers modifizieren. Die registrierten Listener ermöglichen unter Anderem die serverseitige Content-Negotiation, das automatische generieren des HTTP Accept Headers und ein nahezu automatisiertes HATEOAS für die vom Prototypen registrierten Routen.

Im folgenden wird auf den Aufbau der Verzeichnisstruktur eingegangen und auf die Klassen und Dateien, die in den jeweiligen Verzeichnissen liegen.

Abbildung 4.1.: Verzeichnisstruktur - Übersicht



4.1.1. Einstiegspunkt

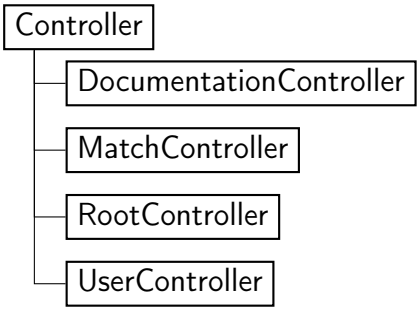
Der Einstiegspunkt der Applikation, auch Frontend-Controller genannt, befindet sich unter `schachApi/web/index.php`. Die Aufgabe des Frontend-Controllers ist die Initialisierung der Applikation und das Beginnen der Abarbeitung.

4.1.2. Controller

Die Controller der Applikation unter `chessApi/app/Controller` definieren die Aktionen und Routen für die, vom System verwalteten, Ressourcen. Beim Start der Applikation werden diese Routen eingelesen und anhand der Zuordnung der `getRoutes` Funktion des Controllers an die Methoden des Controllers gebunden. Die Struktur der Zuordnung wird hinreichend im Quelltext der Controller dokumentiert und wird hier nicht weiter ausgeführt.

Für den Prototyp wurden insgesamt 4 Controller umgesetzt.

Abbildung 4.2.: Verzeichnisstruktur - Controller



Der erste Controller ist der `RootController`. Dieser setzt den Einstiegspunkt in die API um. Von da aus kann ein Konsument dann entsprechend des Entwurfs (vgl. sec:entwurf-hateoas) die `listAction` des `UserControllers` und des `MatchControllers` erreichen. Der `DocumentationController` übernimmt alle OPTIONS-Anfragen, welche gegen alle Endpunkte der Schach API ausgeführt werden können. Dabei liest dieser, für das generieren der Dokumentation, die Metadaten der Route aus. Diese werden in den Controllern in der `getRoutes` Funktion definiert und bei der Generierung der Route, durch die `registerController` der `ControllerProvider` Klasse, als Metadaten angehangen.

4.1.3. DataTransformer

Für die Umsetzung der öffentlichen Datenstruktur werden DataTransformer verwendet. Diese wandeln die interne Darstellung in die öffentliche Darstellung um. Dabei werden alle Attribute einer Ressource, die in der öffentlichen Darstellung enthalten sein sollen, durch die `toArray` Funktion, als assoziatives Array, zurückgegeben.

4.1.4. Entity

Eine Entity beschreibt die interne Darstellung einer Ressource. Im Fall der Schach API, die Partie (Match) und den Spieler (User). Die Verwaltung der Ressourcen erfolgt über einen, jeweils eigenen, Manager. Dieser kümmert sich um das Laden und persistente Speichern einer Ressource.

4.1.5. Resources/views

In diesem Verzeichnis befinden sich die Templates, welche von der Template-Engine Twig verarbeitet werden und so eine HTML-Darstellung ermöglichen. Dabei hat jeder Controller seine eigenen Templates in den, jeweils gleichnamigen, Unterordnern. Die Namen der Templates entsprechen dem Namen der jeweiligen Methode des Controllers.

Das Ganze an einem Beispiel: Wird im `UserController` die Methode `listAction` gerufen, so wird die `list.html.twig` aus dem `User` Verzeichnis mit den, von der `listAction` zurückgegebenen, Parametern von Twig gerendert und als Response an den Konsumenten geschickt.

4.1.6. Service

Die Services enthalten Klassen und Funktionen, die unabhängig vom MVC Pattern agieren können. Im Fall der Schach API befinden sich hier zudem Klassen, die auch unabhängig von der Problemdomäne verwendet werden können.

So wird z.B. ein allgemein gültiges `ManagerInterface` für Entitäten definiert, obwohl die jeweiligen EntityManager, welche unter Entity liegen. Weiterhin wird hier der `FileManager` umgesetzt, welcher das Laden und Speichern von Dateien im lokalen Dateisystem übernimmt. Zuletzt gibt es noch einen `AutoIncrementManager`, welcher ein Dateibasiertes Auto-Increment umsetzt und so das Generieren einer einzigartige Identifikationen auf Dateisystem-Ebene ermöglichen. (Ein `EntityManager` nutzt den `AutoIncrementManager` und den `FileManager` zum persistieren der Ressourcen)

Weitere Dienste umfassen die Validierung der Schachzüge, ein Interface für die Implementierung von weiteren Schach-Engines und ein Interface für KI Systeme. Letzteres wird durch die `ChenardEngine` implementiert.

4.1.7. Das data-Verzeichnis

Alle Daten, die der Prototyp speichert, werden in diesem Verzeichnis abgelegt. Dies beinhaltet die Auto-Increment Dateien der Ressourcen, wie auch die internen Darstellungen der persistierten Ressourcen.

4.1.8. Die externen Bibliotheken

Im `schachApi/vendors` Verzeichnis befinden sich die, vom Prototypen verwendeten, externen Funktionsbibliotheken. Diese wurden verwendet, um ein schnelle Prototypen zu ermöglichen. Zudem wurden, soweit wie möglich, stabile und etablierte Bibliotheken verwendet. Dieser Abschnitt gibt einen Überblick, was diese Bibliotheken ermöglichen und wozu diese verwendet werden. Genauere Informationen zur Verwendung der Bibliotheken sind deren Dokumentation zu entnehmen.

Wie bereits erwähnt verwendet der Prototyp das Silex Framework. Silex ist dabei die Hauptkomponente, da es die Abstraktion einer Anfrage (Request) und einer Antwort (Response) für die Applikation abstrahiert. Weiterhin übernimmt es die Rolle des DI-Containers und ermöglicht es Klassen durch einen ServiceProvider zu registrieren.

Die Hal Bibliothek von Nocarrier setzt das in 3.3.3 erwähnte RFC zu JSON-HAL um. Diese Bibliothek wird verwendet um die, im dritten Format, definierte Struktur, zusammen mit HATEOAS, einfacher umsetzen zu können.

Als Template Engine wird Twig verwendet. Diese ermöglicht das separieren gemäß dem MVC Prinzips und wird dafür verwendet um die Daten, welche von einer Controller-Aktion zurückgegeben werden als HTML zu rendern, sofern der Konsument HTML akzeptiert.

Die Chess-Game Bibliothek wird verwendet, um eine Validierung der Schachspielzüge vornehmen zu können.

4.2. Verweise

Die Darstellung von URI Verweisen wird in den, vom Prototyp unterstützten, Datenformaten (JSON, XML) unterschiedlich dargestellt. Soweit es möglich war, wurde versucht diese so einheitlich wie möglich zu gestalten.

Bei der XML Darstellung verweisen alle Ressourcen durch das `href` Attribut auf sich selbst, sobald eine URI, welche die Ressource beschreibt, existiert. Verweise zu Subressourcen oder den durch HATEOAS definierten Links, werden durch das, eigentlich für HTML definierte, `<link>` Element dargestellt. Das `<link>` Element fordert dabei eine Relation, definiert durch das `rel` Attribut. Die Relation gibt an, welche Daten sich hinter dem Link verbergen, bzw. was ein Konsument erwarten kann. Für alle Endpunkte des Webservices wurden daher einzigartige Namen vergeben, welche als Relation verwendet werden können (siehe Tabelle 4.1).

Tabelle 4.1.: Implementierung: URI Relationen

URI	**Relation**
/	root_index
/users/	user_list
/users/{id}	user_detail
/matches/	match_list
/matches/{id}	match_detail

Durch das `href` Attribut kann eine Ressource auf sich selbst, gemäß der HATEOAS Definitionen (vgl. Abschnitt 3.4), verweisen. Weiterhin ist es auch möglich Verweise auf weitere URIs innerhalb des Webservices anzugeben. Sollte eine URI von einem Parameter abhängig sein, wie z.B. die Detailansichten der Spieler oder Partien, wird die URI mit dem Attribut `templated` versehen. Dieses soll einen Konsumenten der API darauf hinweisen, dass die Parameter noch durch die richtigen Werte ersetzt werden müssen, um auf den Endpunkt zugreifen zu können.

Das JSON Datenformat definiert für Verweise keine spezifischen Konstrukte (vgl. [Cro06]). Daher wurde die Festlegung getroffen, dass Verweise in einem Objekt gesammelt werden, das den Bezeichner `_links` erhalten hat. Zudem wird ein Verweis, ähnlich zum `link` Element des XML Formates, durch ein eigenes Objekt dargestellt. Dieses enthält dabei die URI im `href` Attribut des jeweiligen Objektes. Die Relation wird durch den identifizierenden Schlüsselwert in `_links` für die Objek-

te dargestellt. Entsprechend werden mehrere Objekte gleicher Relation, durch eine Liste, gruppiert dargestellt.

Bei den Detailansichten einer Ressourcen, in den jeweiligen Formaten, werden die Attribute der Ressource, neben den Verweisen, direkt eingebunden. Beispiele der jeweiligen Formate können dem Anhang A.4.1 entnommen werden.

4.3. Kommunikation mit den Endpunkten

Für eine Kommunikation vom Konsumenten zu den, durch POST und PUT angesteuerten, API Endpunkten wurde sich für den, im Anhang A.3.4 definierten, Datentypen `application/x-www-form-urlencoded` entschieden. Dadurch können die Endpunkte durch ein HTML Formular angesteuert werden. Auch die Verwendung mit CLI Tools wie `curl` wird dadurch vereinfacht.

Damit eine Ressource nicht immer von neuem erzeugt werden muss, werden auch die XML und JSON Repräsentationen unterstützt. Dadurch können komplexere Konsumenten die öffentliche Darstellung in deren internen Darstellungen nutzen und bei Änderungen diese direkt an den Webservice schicken, um dadurch die Repräsentation auf dessen Seite zu aktualisieren.

Die formalen Anforderungen der Endpunkte bezüglich dem Verhalten bei erfolgreichen und fehlerhaften Anfragen (vgl. Kapitel 3) wurden alle umgesetzt.

4.4. Einbinden der Subressourcen

Der Prototyp setzt das Einbinden von Subressourcen gemäß der Richtlinien der gewählten Formatdefinition um. Ähnlich den Verweisen (vgl. Abschnitt 4.2) auf die Subressourcen ist die Implementierung für XML anders als für JSON.

In XML wird eine eingebundene Ressource direkt durch einen `resource` Tag dargestellt. Damit diese Ressourcen von anderen unterschieden werden können, ist die Relation der Ressource als `embedded` definiert. Die Repräsentation der eingebundenen Ressourcen entspricht der Darstellung der Detailansicht. Lediglich die Verweise auf höher und tiefer liegende Ressourcen werden nicht mit eingebunden. Die Eigenverlinkung jedoch bleibt bestehen.

Die Darstellung in JSON kann sich näher an die Beschreibungen der Formatdefinition halten. Entsprechend werden die eingebundenen Ressourcen, in einem `_embedded` Feld und durch Ihre URI, in einem assoziativem Array dargestellt.

Beispielrückgaben können dem Anhang A.4 entnommen werden.

4.5. Zugriffschutz

Der Prototyp setzt für den Zugriffsschutz den Punkt "Keine Authentifizierung" um. Entsprechend können alle Konsumenten jegliche Ressourcen verändern. Dies soll die Verwendung des Webservices für das Praktikum (siehe Anhang A.5) erleichtern.

4.6. Künstliche Intelligenz

Vom Prototypen wird die Schach-Engine Chenard[7] als künstliche Intelligenz unterstützt. Diese kann über die PATCH Methode einer Partie, um einen Zug für den aktuellen Spieler gebeten werden. Der von der Chenard Engine ermittelte Zug wird dabei direkt in die Historie der Züge eingefügt.

Um die künstliche Intelligenz in der PATCH Methode integrieren zu können wurde der `text/san` Medientyp in der Umsetzung des Prototyps, um die Definition erweitert, dass eine Eingabe von `KI ZeitInMillisekunden` dazu führt, dass die Schach-Engine den nächsten Zug durchführen soll.

[7] `https://github.com/cosinekitty/chenard`

5. Fazit

Der theoretische Teil dieser Arbeit bearbeitete und betrachtete die notwendigen Definitionen, welche getroffen werden müssen, um einen RESTful Webservice umsetzen zu können. Dabei wurden Methoden entwickelt, welche dabei Helfen die Ressourcen in einem System zu identifizieren, die möglichen Zugriffe auf diese zu definieren und die Repräsentation gegenüber eines Konsumenten der API zu erarbeiten. Die verwendeten Bewertungskriterien wurden allgemein definiert und können bei der Planung anderer Webservices wiederverwendet werden.

Das in der Einleitung definierte Ziel, Methodiken für die Entwicklung eine API zu entwerfen, wurden im Kapitel drei der Arbeit umgesetzt.

Weiterhin wurde durch die Arbeit gezeigt, dass es möglich ist einen Schachserver, als RESTful Webservice zu implementieren.

5.1. Thesen

Aus den Überlegungen dieser Arbeit können folgende Thesen abgeleitet werden:

1. Durch die Entwicklung eines geeigneten öffentlichen Darstellungsformates für die Ressourcen eines RESTful Webservices kann die Menge der notwendigen Kommunikationen optimiert werden.
2. Eine Autorisierung muss immer über eine gesicherte Verbindung stattfinden, da aufgrund des fehlenden Zustandes die Authentifizierungsinformationen immer mitgesendet werden müssen.
3. Der fehlende Zustand eines REST Webservices ist nur auf die Kommunikation zwischen Konsument und Webservice bezogen. Die verwalteten und gespeicherten Repräsentationen können jedoch beliebige Zustände einer Applikation darstellen.

5.2. Ausblick

Die Möglichkeiten, welche ein Webservice zur Verfügung stellt, werden in realen API stetig weiterentwickelt. Dabei sollten jedoch die Konsumenten, welche die vorhandene API nutzen nicht durch diese Änderungen beeinträchtigt werden. Entsprechend

sollte die Stabilität eines Webservices und dessen Möglichkeiten gegenüber einem Konsumenten gewährleistet werden können. Dies könnte durch eine Versionierung der API gewährleistet werden. Diese kann dabei durch mehrere Möglichkeiten erfolgen, jedoch müsste dabei analysiert werden, wie sich die einzelnen Möglichkeiten auf die Eigenschaften der API auswirken.

Sobald der Webservice eine größere Anzahl von Ressourcen verwaltet müsste voraussichtlich eine optimierte Darstellung für die Listenansicht entwickelt werden. Dabei könnte z.B. die Menge der übertragenen Elemente durch einen geeigneten Mechanismus beschreiben oder limitiert werden.

Diese Erweiterungen bieten, neben den Thesen, die Möglichkeiten für weitere Forschungsziele und Analysen im Themengebiet der RESTful Webservices.

Literatur

[Ber05] T. Berners-Lee. *RFC3986 Uniform Resource Identifier (URI): Generic Syntax*. Internet Engineering Task Force (IETF), Jan. 2005. URL: http://tools.ietf.org/html/rfc3986.

[Ber94a] T. Berners-Lee. *RFC1630 Universal Resource Identifiers in WWW*. Internet Engineering Task Force (IETF), Juni 1994. URL: http://tools.ietf.org/html/rfc1630.

[Ber94b] T. Berners-Lee. *RFC1738 Uniform Resource Locators (URL)*. Internet Engineering Task Force (IETF), Dez. 1994. URL: http://tools.ietf.org/html/rfc1738.

[Ber95] T. Berners-Lee. *RFC1866 Hypertext Markup Language - 2.0*. Internet Engineering Task Force (IETF), Nov. 1995. URL: http://www.ietf.org/rfc/rfc1866.txt.

[Ber96] T. Berners-Lee. *RFC1945 Hypertext Transfer Protocol – HTTP/1.0*. Internet Engineering Task Force (IETF), Mai 1996. URL: http://tools.ietf.org/html/rfc1945.

[Cro06] D. Crockford. *RFC4627 The application/json Media Type for JavaScript Object Notation (JSON)*. Internet Engineering Task Force (IETF), Juli 2006. URL: http://www.ietf.org/rfc/rfc4627.txt.

[E H10] Ed. E. Hammer-Lahav. *The OAuth 1.0 Protocol*. Internet Engineering Task Force (IETF), Apr. 2010. URL: https://tools.ietf.org/html/rfc5849.

[Edw94] Steven J. Edwards, Hrsg. *Portable Game Notation Specification and Implementation Guide*. März 1994. URL: http://www6.chessclub.com/help/PGN-spec (besucht am 01. 09. 2015).

[Fie00] Roy Thomas Fielding, Hrsg. *Architectural Styles and the Design of Network-based Software Architectures*. 2000. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (besucht am 01. 09. 2015).

[Fie08] Roy Thomas Fielding, Hrsg. *REST APIs must be hypertext-driven*. Okt. 2008. URL: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven (besucht am 01. 09. 2015).

[Fie99] R. Fielding. *RFC2616 Hypertext Transfer Protocol – HTTP/1.1*. Internet Engineering Task Force (IETF), Juni 1999. URL: http://tools.ietf.org/html/rfc2616.

[Fra99] J. Franks. *RFC2617 HTTP Authentication: Basic and Digest Access Authentication*. Internet Engineering Task Force (IETF), Juni 1999. URL: http://www.ietf.org/rfc/rfc2617.txt.

[Mur01] M. Murata. *RFC3023 XML Media Types*. Internet Engineering Task Force (IETF), Jan. 2001. URL: http://www.ietf.org/rfc/rfc3023.txt.

[Nie93] Jakob Nielsen. *Usability Engineering*. 1993, S. 362.

[Sch03] Hans Karl Schmitz, Hrsg. *Webseite verbessern durch Kontaktdesign*. Feb. 2003. URL: http://kontaktdesign.de/website-verbessern/funktionalitaet/antwortzeit.htm (besucht am 01. 09. 2015).

[Sol94] K. Sollins. *RFC1737 Functional Requirements for Uniform Resource Names*. Internet Engineering Task Force (IETF), Dez. 1994. URL: http://tools.ietf.org/html/rfc1737.

[Wik15a] Wikipedia, Hrsg. *Forsyth-Edwards-Notation*. Sep. 2015. URL: https://de.wikipedia.org/wiki/Forsyth-Edwards-Notation (besucht am 01. 09. 2015).

[Wik15b] Wikipedia, Hrsg. *Paketumlaufzeit - DSL2000*. Sep. 2015. URL: https://de.wikipedia.org/wiki/Paketumlaufzeit%20-%20DSL2000 (besucht am 01. 09. 2015).

Eidesstattliche Erklärung

Ich versichere, dass ich die Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Christoph Kretzschmar

Rosswein, den 15. November 2015

A. Anhang

A.1. HTTP - Methoden

Die Beschreibung der folgenden Methoden wurden Sinngemäß der Spezifikation des HTTP 1.1 Protokolls [Fie99] entnommen.

A.1.1. OPTIONS

Die OPTIONS Methode ist so definiert, dass diese Informationen darüber zurückgibt, welche Möglichkeiten gegenüber der übertragenen URI zur Verfügung stehen. Sofern bei dieser Methode eine Rückgabe von Daten innerhalb der Antwort erfolgt, so sollte diese weiterführende Informationen über die Benutzung, der hinter der URI liegenden Ressource, enthalten. Hierbei ist nicht festgelegt in welchem Format diese Informationen übertragen werden sollen.

Über diese Methode kann man aufgrund dieser Eigenschaften eine API sehr gut dokumentieren. Der Zugriff auf die Dokumentation erfolgt dabei über die jeweilige URI der Ressource. Als Antwort kann man dann ein beliebig definiertes Format wählen, welches der Konsument der API verstehen muss, sofern dieser auf die Dokumentation zugreifen möchte. Für den Zugriff durch einen menschlichen Konsumenten eignet sich z.B. HTML und für die maschienenlesbare Form JSON oder XML.

Diese Methode ist sicher und damit idempotent, da die Ressourcendarstellung nicht verändert wird. Wenn man es ganz genau nimmt, wird bei dieser Methode die Ressource selbst gar nicht beachtet sondern nur Metainformationen zur Verwendung der Ressource.

A.1.2. GET

Die GET Methode definiert den lesenden Zugriff auf eine Ressource. Sollte die verwendete URI dabei jedoch auf einen Daten verarbeitenden Prozess verweisen, so wird dessen Rückgabe in die Antwort geschrieben. Damit ist auch diese Methode sicher und damit idempotent.

A.1.3. HEAD

Diese Methode verhält sich analog zu GET, jedoch wird hierbei die Ressourcenrepräsentation nicht mit übermittelt. Jedoch werden alle Header des Protokolls so übertragen, wie bei einer GET Anfrage. Entsprechend kann diese Methode dazu verwendet werden um Metainformationen abzufragen. Wie die GET Methode ist diese idempotent und sicher.

A.1.4. POST

Bei der POST Methode gibt es verschiedene Möglichkeiten der Funktionsweise. Alle haben dabei gemeinsam, dass die Anfrage als Inhalt die zu verarbeitenden Daten enthält. Die Ressource auf welche die URI zeigt bestimmt dabei, welche Aktion vom Server mit den übergebenen Daten ausgeführt werden soll. Dabei muss diese Aktion nicht immer eine neue Ressource mit einer eigenen URI im System generieren.

Eine Möglichkeit der POST Methode legt z.B, eine neue Ressource in einer Ressourcenliste an. Würde mann nun auf diese neue Ressource, welche keine Liste darstellt, wiederum mit POST zugreifen, so wird diese Ressourcendarstellung z.B. Aktualisiert oder erweitert. Hierbei ist das ganze vergleichbar mit einem HTML Formular. Das eine Formular kann z.B. einen Benutzer erstellen, ein anderes wiederum diesen Benutzer bearbeiten. Beide arbeiten dabei mit der POST Methode, welche gegen unterschiedliche Endpunkte angewendet wird.

Aufgrund möglicher Seiteneffekte der von einer POST Methode ausgelösten Aktion ist diese Methode nicht idempotent und dadurch auch nicht sicher.

A.1.5. PUT

Durch die PUT Methode kann eine Ressource direkt innerhalb einer API angelegt oder verändert werden. Dabei soll die in der Anfrage enthaltene Repräsentation der resultierenden Ressource entsprechen. Die PUT Methode entspricht dabei dem Ersetzen einer Ressource auf einem durch eine URI definierte Ressource. Existiert die Ressource bereits, so wird diese, mit der in der Anfrage enthaltenen Darstellung, überschrieben. Sollte die Ressource jedoch noch nicht existieren, so wird die in der Anfrage enthaltene Darstellung unter dieser URI angelegt und gespeichert.

Dies ist auch der fundamentale Unterschied zu der POST Methode. Bei der POST Methode wurde ein Ressourcenendpunkt angegeben, welcher die übergebenen Daten verarbeitet hat und dann in der richtigen Ressource gespeichert hat. Entsprechend legt der Server fest wo und wie die Ressource zu speichern ist. Bei PUT definiert jedoch der Konsument wo eine Ressource abgelegt werden soll. Anhand der Content Negotiation wird dabei sichergestellt, das beide Seiten wissen, welche Entität auf der URI abgelegt werden kann und welche nicht.

Da der Inhalt gleicher Anfragen und die daraus resultierende Ressource immer gleich ist, kann diese Methode als idempotent betrachtet werden. Jedoch könnte eine bestehende Ressource überschrieben werden wodurch diese Methode nicht sicher ist.

A.1.6. DELETE

Mit der DELETE Methode kann ein Konsument das Löschen einer Ressource beantragen. Jedoch kann nicht garantiert werden, dass die Ressource sofort gelöscht wurde, selbst wenn die Antwort vom Server erfolgreich ist. Dies liegt daran, da man mögliche Interaktionen vor einem Löschen im Standard beachtet hat. Jedoch sollte nach einer erfolgreichen Antwort des Servers auf die Anfrage des Löschens einer Ressource diese nach Möglichkeit unzugänglich gemacht werden.

Die DELETE Methode ist idempotent. Dies erklärt sich dadurch, dass eine Ressource die gelöscht werden sollte, aber ggf. durch eine vorherige DELETE Anfrage bereits gelöscht wurde, noch immer eine gelöschte und damit nicht mehr vorhandene Ressource darstellt. Da die Ressource durch das löschen Verändert wird ist diese Methode nicht sicher.

A.1.7. PATCH

Die PATCH Methode bietet die Möglichkeit eine Ressource anhand eines nicht näher definierten Formates zu verändern. Sofern diese Methode verwendet wird, muss dieses Format hinreichend dokumentiert werden. Dabei muss dieses Format nicht der Repräsentation der Ressource entsprechend, welche verändert werden soll. Das Format soll eine Vorlage bieten, wie die Daten zu verändern sind.

Sofern ein Ressourcenendpunkt diese Methode unterstützt muss diese einen `Allow-Patch` Header bei jeder Anfrage mit versenden. Der Inhalt des Headers ist eine Komma

separierte Liste von Medienformaten, welche von dieser Methode als Eingabe unterstützt werden.

Da diese Methode die Ressource anhand eines nicht näher definierten Vorganges bearbeitet, ist diese Methode nicht idempotent und auch nicht sicher.

A.2. HTTP - Statucodes

Die in diesem Anhang verwendeten Tabellen, basieren auf der Quelle `http://webmasterparadies.de/webmasterwissen/130-http-status-codes-und-fehlermeldungen.html`.

A.2.1. 1xx

Tabelle A.1.: 1xx Statuscodes

Statuscode	**Bezeichnung**	**Erläuterung**
100	Continue	Häufig bei großen Anfragen an den Servern der Fall. Die Anfrage wurde noch nicht zurückgewiesen und der Client kann fortfahren.
101	Switching Protocols	Wird verwendet wenn der Server mit dem Wechsel zu einem anderen Protokoll einverstanden ist.
102	Processing	Wird verwendet um Timeouts zu vermeiden

A.2.2. 2xx

Tabelle A.2.: 2xx Statuscodes

200	OK	Die Anfrage wurde erfolgreich bearbeitet und das Ergebnis der Anfrage wird in der Antwort übertragen.
201	Created	Die Anfrage wurde erfolgreich bearbeitet. Die angeforderte Ressource wurde vor dem Senden der Antwort erstellt.
202	Accepted	Die Anfrage wurde akzeptiert wird aber zu einem späteren Zeitpunkt ausgeführt.
203	Non-Authoritative Information	Die Anfrage wurde bearbeitet das Ergebnis ist aber nicht unbedingt vollständig und aktuell.
204	No Content	Die Anfrage wurde erfolgreich durchgeführt die Antwort enthält jedoch keine Daten.
205	Reset Content	Die Anfrage wurde erfolgreich durchgeführt der Client soll das Dokument neu aufbauen und Formulareingaben zurücksetzen.
206	Partial Content	Der angeforderte Teil eines Downloads wurde erfolgreich übertragen.
207	Multi-Status	Die Antwort enthält ein XML-Dokument das mehrere Statuscodes zu unabhängig voneinander durchgeführten Operationen enthält.

A.2.3. 3xx

Tabelle A.3.: 3xx Statuscodes

300	Multiple Choice	Die angeforderte Ressource steht in verschiedenen Arten zur Verfügung.
301	Moved Permanently	Die angeforderte Ressource steht ab sofort unter der im „Location“-Header-Feld angegebenen Adresse bereit. Die alte Adresse ist nicht mehr gültig.
302	Found	Die angeforderte Ressource steht vorübergehend unter der im „Location“-Header-Feld angegebenen Adresse bereit (in HTTP/1.0 „Moved Temporarily“). Die alte Adresse bleibt gültig. Wird in HTTP/1.1 je nach Anwendungsfall durch die Statuscodes 301 bzw. 307 ersetzt. 302-Weiterleitung ist aufgrund eines Suchmaschinen-Fehlers dem URL-Hijacking in Kritik geraten. Webmaster sollten generell von der Verwendung eines solchen Redirects absehen.
303	See Other	Die Antwort auf die durchgeführte Anfrage lässt sich unter der im „Location“-Header-Feld angegebenen Adresse beziehen.
304	Not Modified	Die durchgeführte Anfrage führt zur selben Antwort wie zur vom Client übermittelten Zeit im „If-Modified-Since“-Header-Feld oder sie passt zu dem im „If-None-Match“-Header-Feld gesendeten Entity-Tag. Sie wurde deshalb nicht mit übertragen.
305	Use Proxy	Die angeforderte Ressource ist nur über einen Proxy erreichbar.
306	(reserviert)	"306 wird nicht mehr verwendet ist aber reserviert. Es wurde für Switch Proxy verwendet."
307	Temporary Redirect	Die angeforderte Ressource steht vorübergehend unter einer anderen Adresse bereit. Die alte Adresse bleibt jedoch gültig.

A.2.4. 4xx

Tabelle A.4.: 4xx Statuscodes Teil 1

400	Bad Request	Die Anfrage-Nachricht war fehlerhaft aufgebaut.
401	Unauthorized	Die Anfrage kann nicht ohne gültige Authentifizierung durchgeführt werden.
402	Payment Required	(reserviert)
403	Forbidden	Die Anfrage wurde mangels Berechtigung des Clients nicht durchgeführt.
404	Not Found	Die angeforderte Ressource wurde nicht gefunden. Dieser Statuscode kann ebenfalls verwendet werden um eine Anfrage ohne näheren Grund abzuweisen.
405	Method Not Allowed	Die Anfrage darf nur mit anderen HTTP-Methoden (z. B. GET statt POST) gestellt werden.
406	Not Acceptable	Die angeforderte Ressource steht nicht in der gewünschten Form zur Verfügung.
407	Proxy Authentication Required	Analog zum Statuscode 401 ist hier zunächst eine Authentifizierung des Clients gegenüber dem verwendeten Proxy erforderlich.
408	Request Timeout	Innerhalb der vom Server erlaubten Zeitspanne wurde keine Anfrage des Clients empfangen.
409	Conflict	Die Anfrage wurde unter falschen Annahmen gestellt.
410	Gone	Die angeforderte Ressource wird nicht länger bereitgestellt.
411	Length Required	Die Anfrage kann ohne ein „Content-Length“-Header-Feld nicht bearbeitet werden.

Tabelle A.5.: 4xx Statuscodes Teil 2

412	Precondition Failed	Eine in der Anfrage übertragene Voraussetzung traf nicht zu.
413	Request Entity Too Large	Die gestellte Anfrage war zu groß um vom Server bearbeitet werden zu können.
414	Request-URI Too Long	Die URI der Anfrage war zu lang. Ursache ist oft eine Endlosschleife aus Redirects.
415	Unsupported Media Type	Der Inhalt der Anfrage wurde mit ungültigem oder nicht erlaubtem Medientyp übermittelt.
416	Requested range not satisfiable	Der angeforderte Teil einer Ressource war ungültig oder steht auf dem Server nicht zur Verfügung.
417	Expectation Failed	Das im „Expect“-Header-Feld geforderte Verhalten des Servers kann nicht erfüllt werden.
421	There are too many connections from your internet address	Verwendet wenn die Verbindungshöchstzahl überschritten wird
422	Unprocessable Entity	Die Anfrage wurde wegen semantischer Fehler abgelehnt.
423	Locked	Die angeforderte Ressource ist zurzeit gesperrt.
424	Failed Dependency	Die Anfrage konnte nicht durchgeführt werden weil sie das Gelingen einer vorherigen Anfrage voraussetzt.
425	Unordered Collection	In den Entwürfen von WebDav Advanced Collections definiert aber nicht im "Web Distributed Authoring and Versioning (WebDAV) Ordered Collections Protocol".
426	Upgrade Required	Der Client sollte auf Transport Layer Security (TLS/1.0) umschalten.

A.2.5. 5xx

Tabelle A.6.: 5xx Statuscodes

500	Internal Server Error	Unerwarteter Serverfehler
501	Not Implemented	Die Funktionalität, um die Anfrage zu bearbeiten, wird von diesem Server nicht bereitgestellt.
502	Bad Gateway	Der Server konnte seine Funktion als Gateway oder Proxy nicht erfüllen, weil er seinerseits eine ungültige Antwort erhalten hat.
503	Service Unavailable	Der Server steht, zum Beispiel wegen Überlast oder Wartungsarbeiten, zurzeit nicht zur Verfügung.
504	Gateway Timeout	Der Server konnte seine Funktion als Gateway oder Proxy nicht erfüllen, weil er innerhalb einer festgelegten Zeitspanne keine Antwort von seinerseits benutzten Servern oder Diensten erhalten hat.
505	HTTP Version not supported	Die benutzte HTTP-Version wird vom Server nicht unterstützt oder abgelehnt.
506	Variant Also Negotiates	Die Inhaltsvereinbarung der Anfrage ergibt einen Zirkelbezug.
507	Insufficient Storage	Die Anfrage konnte nicht bearbeitet werden, weil der Speicherplatz des Servers dazu zurzeit nicht mehr ausreicht.
509	Bandwidth Limit Exceeded	Die Anfrage wurde verworfen, weil sonst die verfügbare Bandbreite überschritten werden würde.
510	Not Extended	Die Anfrage enthält nicht alle Informationen, die die angefragte Server-Extension zwingend erwartet.

A.3. Medienformate

Um mit der API kommunizieren zu können, müssen der Webservice und der Konsument ein geeignetes Format über die Content-Negotiation aushandeln. Da eine große Anzahl von Datenformaten existiert[8] muss diese bei der Planung der API betrachtet werden. Entsprechend werden die folgenden Formate definiert, welche innerhalb des Webservices verwendet werden.

Formate müssen über den HTTP `Accept` Header von Webservice und Konsument zur Wahl angeboten und über den `Content-Type` Header verwendet werden. Der Webservice sollte dabei, soweit möglich, alle für eine Aktion angebotenen Formate gleichwertig verarbeiten. Entsprechend darf die Menge der Seiteneffekte einer Funktion nicht vom gewählten Format abhängig sein.

A.3.1. Extended Markup Language - XML

Das Datenformat `text/xml` wird durch [Mur01] definiert. Der Inhalt einer XML kann weiterhin durch eine Schema Datei definiert werden. Diese gibt dabei an, welche Felder erlaub sind, bzw. welcher Inhalt in der XML Repräsentiert werden soll.

A.3.2. JavaScript Object Notation - JSON

Das Datenformat `application/json` wird durch die [Cro06] definiert. Es definiert dabei eine Repräsentation eines JavaScript Objektes.

A.3.3. Hypertext Markup Language HTML

Das Datenformat `text/html` Format wird definiert durch [Ber95]. Setzt ein Webservice dieses Format um, so kann er dadurch entweder den Webservice und dessen Gegebenheiten Dokumentieren und erläutern. Weiterhin ist es möglich einen minimalen Konsumenten für die API über dieses Format zu definieren.

[8] `http://wiki.selfhtml.org/wiki/Referenz:MIME-Typen`

A.3.4. URL Codierte Daten

Das Datenformat `application/x-www-form-urlencoded` wird in [Ber95] im Kapitel 8.2.1 beschrieben. Dabei werden die über dieses Format übertragen Daten ähnlich dem Abfrage (query) Teil einer URL/URI dargestellt (vgl. 2.1).

A.3.5. Forsyth-Edwards-Notation

Das Datenformat `text/fen` wird nicht durch einen Standard beschrieben. Die Definition ergibt sich dabei aus der EBNF, welche im Grundlagenkapitel 2.4.1 zu finden ist.

A.3.6. Standard Algebraic Notation

Das Datenformat `text/san` wird nicht durch einen Standard beschrieben. Die Definition dieses Formates entspricht der Definition der Standard Algebraic Notation aus dem Grundlagenkapitel 2.4.2

A.4. Implementierungsbeispiele

Die Implementierungsbeispiele wurden dem Prototypen entnommen. Dabei existierten zwei Benutzer und eine Partie im System.

A.4.1. XML

```
<?xml version="1.0"?>
<resource href="/users/">
<link rel="user" href="/users/1"/>
<link rel="user" href="/users/2"/>
<link rel="root_index" href="/"/>
<link rel="user_detail" href="/users/{id}" templated="1"/>
</resource>
```

Listing A.1: Implementierungsbeispiel - Benutzerverzeichnis in XML

```
<?xml version="1.0"?>
<resource href="/users/1">
<link rel="user_list" href="/users/"/>
<id>1</id>
<name>Bernd</name>
</resource>
```

Listing A.2: Implementierungsbeispiel - Benutzerdetails in XML

```
<?xml version="1.0"?>
<resource href="/matches/">
<link rel="match" href="/matches/1"/>
<link rel="root_index" href="/"/>
<link rel="match_detail" href="/matches/{id}" templated="1"/>
</resource>
```

Listing A.3: Implementierungsbeispiel - Partieverzeichnis in XML

```
<?xml version="1.0"?>
<resource href="/matches/1">
<link rel="match_list" href="/matches/"/>
<id>1</id>
<white>/users/1</white>
<black>/users/2</black>
<start>rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1</start>
<history>e4</history>
```

```
<history>d6</history>
</resource>
```

Listing A.4: Implementierungsbeispiel - Partiedetails in XML

```
<?xml version="1.0"?>
<resource href="/matches/">
<link rel="match" href="/matches/1"/>
<link rel="root_index" href="/"/>
<link rel="match_detail" href="/matches/{id}" templated="1"/>
<resource rel="embedded" href="/matches/1">
  <id>1</id>
  <white>/users/1</white>
  <black>/users/2</black>
  <start>rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1</
     start>
  <history>e4</history>
  <history>d6</history>
</resource>
<resource rel="embedded" href="/users/1">
  <id>1</id>
  <name>Bernd</name>
</resource>
<resource rel="embedded" href="/users/2">
  <id>2</id>
  <name>Max</name>
</resource>
</resource>
```

Listing A.5: Implementierungsbeispiel - Einbinden von Subressourcen in XML

A.4.2. JSON

```
{
  "_links": {
    "self": {
      "href": "/users/"
    },
    "user": [
      {
        "href": "/users/1"
      },
```

```
      {
        "href": "/users/2"
      }
    ],
    "root_index": {
      "href": "/"
    },
    "user_detail": {
      "href": "/users/{id}",
      "templated": true
    }
  }
}
```

Listing A.6: Implementierungsbeispiel - Benutzerverzeichnis in JSON

```
{
  "id": 1,
  "name": "Bernd",
  "_links": {
    "self": {
      "href": "/users/1"
    },
    "user_list": {
      "href": "/users/"
    }
  }
}
```

Listing A.7: Implementierungsbeispiel - Benutzerdetails in JSON

```
{
  "_links": {
    "self": {
      "href": "/matches/"
    },
    "match": [
      {
        "href": "/matches/1"
      }
    ],
    "root_index": {
      "href": "/"
    },
```

```
    "match_detail": {
      "href": "/matches/{id}",
      "templated": true
    }
  }
}
```

Listing A.8: Implementierungsbeispiel - Partieverzeichnis in JSON

```
{
  "id": 1,
  "white": "/users/1",
  "black": "/users/2",
  "start": "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1",
  "history": [
    "e4",
    "d6"
  ],
  "_links": {
    "self": {
      "href": "/matches/1"
    },
    "match_list": {
      "href": "/matches/"
    }
  }
}
```

Listing A.9: Implementierungsbeispiel - Partiedetails in JSON

```
{
  "_embedded": {
    "/matches/1": {
      "id": 1,
      "white": "/users/1",
      "black": "/users/2",
      "start": "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0
          1",
      "history": [
        "e4",
        "d6"
      ]
    },
    "/users/1": {
```

```
      "id": 1,
      "name": "Bernd"
    },
    "/users/2": {
      "id": 2,
      "name": "Max"
    }
  },
  "_links": {
    "self": {
      "href": "/matches/"
    },
    "match": [
      {
        "href": "/matches/1"
      }
    ],
    "root_index": {
      "href": "/"
    },
    "match_detail": {
      "href": "/matches/{id}",
      "templated": true
    }
  }
}
```

Listing A.10: Implementierungsbeispiel - Einbinden von Subressourcen in JSON