Hochschule für Telekommunikation Leipzig (FH)

Abschlussarbeit zur Erlangung des akademischen Grades

Bachelor of Engineering

Thema: Entwicklung eines generischen RESTful URI Schemas zur Ausführung von Datenbankabfragen

Vorgelegt von: Christopher Rost

Geboren am: 06.09.1990
in: Querfurt

eingereicht am: 07. September 2015

Erstprüfer: Prof. Dr. Matthias Krause
Hochschule für Telekommunikation Leipzig

Zweitprüfer: Mario Leonhard
Gesundheitsforen Leipzig GmbH

Inhaltsverzeichnis

1. Einleitung **7**
1.1. Problemstellung und Motivation . . . 7
1.2. Systemidee und Zieldefinition . . . 8
1.3. Anforderungserhebung . . . 9

2. Technologien **10**
2.1. REST . . . 10
2.1.1. Definition . . . 10
2.1.2. Kommunikation . . . 10
2.1.3. Ressourcen und deren Repräsentationen . . . 11
2.1.4. URIs . . . 12
2.1.5. URI-Templates . . . 14
2.1.6. Hypermedia . . . 14
2.2. SQL . . . 16
2.2.1. Einordnung . . . 16
2.2.2. Relationale Datenbanksysteme . . . 17
2.2.3. Der Standard SQL . . . 17
2.2.4. Die SELECT Syntax . . . 18
2.2.5. JDBC . . . 21

3. Softwaredesign **23**
3.1. Identifikation der Architekturbausteine . . . 23
3.2. Interne Anwendungslogik . . . 24
3.3. Datenbankanbindung . . . 25
3.4. Architektur des RESTful Webservice . . . 26
3.4.1. Daten in lesbare Ressourcen umwandeln . . . 26
3.4.2. Identifikation durch URIs . . . 27
3.4.3. Auswahl und Entwurf der Repräsentationen . . . 29
3.4.4. Hypermediaverweise . . . 31
3.4.5. Fehlersituationen . . . 33
3.4.6. Dokumentation . . . 33
3.4.7. Anmerkungen zur REST-Konformität . . . 35
3.4.8. JAX-RS . . . 36

4. URI-Schema à la SQL **37**
4.1. Möglichkeiten der Übermittlung einer SQL-Abfrage . . . 37
4.2. Erlaubte Zeichen im Pfad-Segment der URI . . . 39
4.3. Entwicklung des Schemas . . . 39
4.4. Das URI-Schema und dessen generischer Charakter . . . 52
4.5. RAML Dokumentation des Schemas . . . 52

5. Realisierung **54**
5.1. JAX-RS Integration . . . 54
5.2. Generisches URI-Schema durch Annotationen und Sub-Ressourcen . . . 55
5.3. Abbildung der Ressourcen . . . 56
5.4. JOINs über rekursiven Aufruf der Sub-Resource . . . 57

5.5. Ressourcen zu Repräsentationen 59
5.6. Fehlerbehandlung 61

6. Zusammenfassung **63**

A. Anhang **68**
A.1. Ausschnitt I der RAML API-Dokumentation 68
A.2. Ausschnitt II der RAML API-Dokumentation 70
A.3. HTML-Repräsentation der RAML API Dokumentation 71
A.4. SQL SELECT Syntax in EBNF 72
A.5. RESTful URI-Schema zur Identifikation der Datenbankschema beschreibenden Ressourcen in EBNF 77
A.6. RESTful URI-Schema zur Übermittlung einer SQL Selecgt-Abfrage in EBNF . . 78
A.7. Java-Anwendung auf CD 84

Abkürzungsverzeichnis

ABNF Augmented Backus-Naur Form

ANSI American National Standards Institute

API Application Programming Interface

ADO Active Data Objects

DAO Data Access Objects

DBI DataBase Interface

DDL Data Definition Language

DML Data Manipulation Language

DCL Data Control Language

DBMS Datenbankmanagementsystem

EBNF Erweiterte Backus-Naur Form

WWW World Wide Web

HTTP Hypertext Transfer Protocol

HTML Hypertext Markup Language

IANA Internet Assigned Numbers Authority

JAX-RS Java API for RESTful Web Services

JDK Java SE Development Kit

JDBC Java Database Connectivity

JSON JavaScript Object Notation

MIME Multipurpose Internet Mail Extensions

ODBC Open Database Connectivity

RAML RESTful API Modeling Language

RDBMS Relationales Datenbankmanagementsystem

ROA Resource Oriented Architecture

SOAP Simple Object Access Protocol

SQL Structured Query Language

UML Unified Modeling Language

URI Universal Resource Identifier

URL Universal Resource Locator

XML Extensible Markup Language

YAML YAML Ain't Markup Language

1. Einleitung

1.1. Problemstellung und Motivation

> *Die gute alte SQL-Datenbank ist nach wie vor die informationstechnische Basis fast jeder Geschäftstätigkeit.* [16, S. 5]

Relationale Datenbankmanagementsysteme weisen durch ihr tabellenorientiertes Datenmodell und die standardisierte Abfragesprache SQL einen hohen Akzeptanzfaktor auf und gelten bis heute als unerlässlicher Bestandteil vieler Unternehmen. Sollen Anwendungen auf die darin enthaltenen Daten zugreifen, benötigen sie zur Kommunikation mit der Datenbank eine einheitliche Schnittstelle. Für Desktop- und Server-Anwendungen, die sich in Reichweite der Datenbank befinden, existieren Datenbankschnittstellen, wie Java Database Connectivity (JDBC), Open Database Connectivity (ODBC) und DataBase Interface (DBI), die die Verbindung zur Datenbank herstellen.

Nicht nur große Konzerne, wie Microsoft, Google und Amazon, sondern auch kleinere Unternehmen setzten heutzutage vermehrt auf Anwendungen, die unabhängig von der eingesetzten Hardware, im Browser eines Endgerätes ausgeführt werden können. „Immer mehr IT-Anwendungen werden nicht mehr für den Desktop-PC programmiert, sondern als Webanwendung konzipiert und entwickelt."[47] Da diese jedoch ausschließlich über Hypertext Transfer Protocol (HTTP) mit einer Serverinstanz kommunizieren können, benötigen sie einen Webservice, der alle erforderlichen Daten in geeigneter Form bereitstellt. Hierbei zeichnet sich die klassische „3-Tier Architektur" aus, die eine Software in 3 Schichten einteilt: die Client-Schicht, die Anwendungslogik und die Datenbank. Der Webservice, welcher die Anwendungslogik darstellt, wird im Normalfall für jeden Anwendungsfall und Datenbestand individuell implementiert, da sich die zu bereitstellenden Daten, deren Struktur und Beziehung untereinander von Fall zu Fall unterscheiden. Dieses Vorgehen kann sowohl zeitaufwändig als auch fehleranfällig sein und bedarf an fundierten Fachkenntnissen in den Bereichen Anwendungsentwicklung, Webservices und anderen Grundprinzipien des Web. Da sich nun auch der REST-konforme Architekturstil im Webservice-Bereich stark etabliert hat, ist es zudem wünschenswert, dass der zu entwickelnde Dienst dessen Konzepte berücksichtigt.

Man erkennt, dass mit dem Vorhaben, Daten einer relationalen Datenbank mittels eines Webservice bereitzustellen, ein nicht vernachlässigbarer Aufwand für Design, Entwurf und Entwicklung der Server-Komponente entsteht. Zudem muss den Entwicklern der Client-Anwendungen, die mit dem Webservice kommunizieren, notwendigerweise eine Dokumentation der API zur Verfügung gestellt werden. Der Entwickler benötigt zwingend Kenntnisse über alle akzeptieren HTTP-Methoden, das zugrundeliegende URI-Schema, die darüber erreichbaren Ressourcen und deren Repräsentationsformate, um nur einige zu nennen.

Um den beschriebenen Aufwand zu minimieren, wäre eine allgemein verwendbare serverseitige Anwendung notwendig, die als Schnittstelle zwischen relationaler Datenbank und Client-Webanwendung fungiert. Welche relationale Datenbank angebunden wird, sollte irrelevant sein. Das URI-Schema, über das der Client die Daten vom Webservice anfordert, sollte ebenfalls unabhängig von dem Datenbestand und dessen Struktur gültig sein und sich an bekannte Konventionen lehnen, damit der Aufwand zur Einarbeitung in dessen Struktur möglichst gering bleibt. Es muss sozusagen so generisch sein, dass es auf alle Tabellen und Relationen einer beliebigen Datenbank anwendbar ist.

Ein generisches URI-Schema, dessen Syntax sich beispielsweise an der SQL-Grammatik orientiert, würde in kurzer Zeit von den meisten Client-Anwendungsentwicklern verstanden werden, da SQL als Abfragesprache im informationstechnischen Umkreis allgemein bekannt ist bzw. sein

sollte. Den damit verbundenen Vorteil, haben schon verschiedene Unternehmen aufgefasst und programmatisch umgesetzt. Produkte, wie DreamFactory Service Platform, SlashDB, restSQL, Jongo, sandman, MySQL HTTP Plugin, Google Fusion Tables und SQL Share REST API, verfolgen das selbe Ziel: einen möglichst komfortablen Zugriff auf einen Datenbestand über das HTTP. Die Spezifikationen der genannten Produkte sind auf den jeweiligen Internetpräsenzen beschreiben und werden in dieser Arbeit nicht erläutert. Zusammenfassend kann man sagen, dass jedes Produkt seine eigene Herangehensweise an die Übermittlung der Information hat, welche Daten der Client aus der Datenbank benötigt. Dies ist an einigen Stellen jedoch so eingeschränkt, dass beispielsweise Relationen zwischen Tabellen nicht berücksichtigt werden oder nur einzelne Datensätze angefragt werden können. Die konkrete Möglichkeit, SQL-Abfragen an den Webservice zu übermitteln, bieten nur die Produkte „MySQL HTTP Plugin“ und „SQL Share REST API“. Beide nutzen für die Übermittlung der Anweisung den Query-String eines URIs (siehe Kapitel 2.1.4). In Kapitel 4.1 wird sich mit Vor- und Nachteilen dieser Variante auseinandergesetzt. Das „MySQL HTTP Plugin“ ist zudem nur auf MySQL Datenbanken anwendbar.

Ein Webservice, der SQL-Abfragen über sein generisches URI-Schema ohne die Verwendung von Query-Parametern entgegennimmt und dessen Architektur zudem weitestgehend nach den REST Prinzipien designt ist, existiert derzeit nicht. Das nachfolgende Kapitel beschreibt die Idee, die mit dem Fehlen einer solchen Lösung einhergeht.

1.2. Systemidee und Zieldefinition

Die Idee besteht darin, eine Anwendung zu entwickeln, die das Datenbankschema und die Datensätze aus einer beliebigen relationalen Datenbank abruft und über einen REST-konformen Webservice bereitstellt. Der Ort, Treiber und die Authentifizierungsinformationen der Datenbank kann der Nutzer über eine Konfigurationsdatei festlegen. Der Konsument des Webservice, sei es der Entwickler einer Client-seitigen Webanwendung oder ein Nutzer, der Datenbankinformationen über einen Browser abrufen möchte, kann mit seinen Kenntnissen der Abfragesprache SQL ohne aufwändige Einarbeitung einen URI erzeugen, mit dem die gewünschten Daten vom Webservice abgerufen werden können.

Ziel dieser Bachelor Thesis ist es, ein generisches URI-Schema eines solchen REST-konformen Webservices zu entwickeln, das sich stark an den syntaktischen Aufbau einer SQL Select-Anfrage orientiert und somit die Übermittlung einer Datenbankabfrage ermöglicht. Weiterhin soll ein Webservice konzipiert und implementiert werden, der dieses URI-Schema zum Anfordern der Daten einer angebundenen Datenbank umsetzt.

Anfangs wird auf die Grundlagen von REST und SQL eingegangen. Danach erfolgt der Software-Entwurf, dessen Schwerpunkt auf der Konzeption der REST-Architektur liegt. Dabei wird ein hoher Wert auf die durchgängige Berücksichtigung des Hypermedia-Konzeptes gelegt. Dadurch wird dem Client mit jeder Response mitgeteilt, welche weiteren Ressourcen er über welchen URI findet. Im Anschluss folgt der wesentliche Bestandteil dieser Arbeit, die Entwicklung des oben genannten URI-Schemas, womit eine von der vorliegenden Datenbank und dessen Tabellenstruktur unabhängige Möglichkeit zur Identifikation der Datensätze innerhalb der Datenbank geschaffen wird. Es werden ausschließlich einfache Select-Anfragen ohne Unterabfragen berücksichtigt, da die Umsetzung aller in SQL formulierbaren Anweisungen eine sehr hohe Komplexität aufweist und somit den Rahmen einer Bachelorarbeit übersteigen würde. Es wird jedoch angestrebt, eine Erweiterbarkeit und Skalierbarkeit der Abfragen zu ermöglichen.

1.3. Anforderungserhebung

Der nachfolgenden Auflistung sind alle Anforderungen an die zu entwickelnde Anwendung zu entnehmen. Diese beziehen sich auf die in der Systemidee genannten Features.

Konfigurierbarkeit	Authentifizierungsmerkmale, Treiber und Pfad zur Datenbank sollen vom Administrator der Anwendung konfiguriert werden können.
Datenbankanbindung	Eine gängige relationale Datenbank kann unabhängig von deren Umfang und Struktur angebunden werden.
Webserver	Innerhalb der Anwendung läuft ein leichtgewichtiger Webserver, der HTTP Requests empfangen und mit HTTP Responses antworten kann.
RESTful Webservice	Ein nach den REST-Architekturprinzipien entworfener Webservice ist an die Webserverinstanz gebunden.
API Dokumentation	Eine API Dokumentation soll verfügbar sein, die neben dem URI-Schema auch alle verwendeten HTTP- Methoden und -Statuscodes sowie Beispielantworten enthält.
Datenbankschema	Das Schema der angebundenen Datenbank (Tabellen, Attribute und deren Eigenschaften) kann über den Webservice ausgelesen werden.
URI-Schema	Das URI-Schema orientiert sich an der Datenbankabfragesprache SQL.
SQL Abfragen	SQL Abfragen können über den Webservice an die Datenbank gestellt werden. Die Ergebnismenge wird an den Sender innerhalb der Response zurückgeliefert.
MIME-Types	Die innerhalb der Response übermittelte Ergebnismenge der SQL-Abfrage steht sowohl in einem für den Menschen lesbaren, als auch für eine Maschine verarbeitbaren Format zur Verfügung.
Plattformunabhängigkeit	Die Anwendung ist uneingeschränkt auf den aktuellen Desktop Betriebssystemen (Windows, Linux, iOS) ausführbar.

2. Technologien

2.1. REST

In diesem Kapitel wird der Architekturstil REST definiert, Begrifflichkeiten im Zusammenhang mit REST erklärt und dessen Grundprinzipien erläutert.

2.1.1. Definition

Roy T. Fielding beschäftigte sich innerhalb seiner 2000 veröffentlichten Dissertation „Architectural Styles and the Design of Network-based Software Architectures“[6] mit Unklarheiten der Bedeutung einzelner Konzepte des World Wide Web (WWW) und Problemen aktuell im Web verwendeter Technologien und erarbeitete daraufhin verschiedene Lösungsansätze. Als Ergebnis erstellte er ein auf der HTTP-Architektur aufsetzendes Konzept eines abstrakten Architekturstils für verteilte Hypermedia-Systeme. Der somit entstandene **Re**presentational **S**tate **T**ransfer (REST) Architekturstil bietet eine Sammlung von architektonischen Richtlinien, mit dem Ziel die Skalierbarkeit von Interaktionen zwischen Komponenten, die Allgemeingültigkeit von Schnittstellen und die unabhängige Entwicklung von Komponenten im Web zu gewährleisten. Weiterhin sollen mit intermediären Komponenten Latenzen von Interaktionen verringert, Sicherheitsaspekte erhöht und veraltete Systeme verborgen werden. [Vgl. 6, Absatz 4.4]

Fieldings Architekturstil kann heutzutage als Meilenstein angesehen werden, da die Akzeptanz von REST stetig steigt [vgl. 43, S. 10]. Im Gegensatz zu anderen weit verbreiteten Architekturen, wie beispielsweise Simple Object Access Protocol (SOAP)-basierte Webservices, bietet REST eine deutlich stärkere Entkopplung zweier Systeme, nicht zuletzt durch die bewusst fehlende Festlegung auf ein bestimmtes Nachrichtenformat. Mit REST setzt man Standards des Web, wie HTTP, URIs und Hypermedia, so ein, „wie es sich deren Architekt vorgestellt hat“[43, S. 10]. Die Wahrscheinlichkeit, dass die vorgesehenen Vorteile dieser Standards optimal ausgenutzt werden, ist damit am höchsten.

Bei der Verwendung des Begriffes REST sollte man genau genommen drei Interpretationen unterscheiden: REST als abstrakter Architekturstil, der konkreten REST-konformen Implementierung HTTP und Webservices, die konform zu den REST-Prinzipien implementiert wurden [vgl. 43, S. 11]. Im Umfang dieser Arbeit werden diese jedoch synonym verwendet.

2.1.2. Kommunikation

Eine konkrete Ausprägung des Architekturstils REST ist das HTTP [9]. HTTP ist ein zustandsloses Protokoll, welches die Übertragung von Daten auf der Anwednungsschicht nach dem Request-Response-Verfahren ermöglicht. Es nutzt standardmäßig TCP als zuverlässiges Transportprotokoll. Die Vorteile von HTTP, wie Interoperabilität, klar definierte Methoden, Kompressions-, Caching- und Load-Balancing-Mechanismen sowie integrierte Authentifizierungs- und Verschlüsselungsmechnismen machen es attraktiv für verschiedene Dienste im Web. Nachfolgend werden Methoden sowie Statuscodes von HTTP und die Eigenschaften der zustandslosen Kommunikation näher beleuchtet, da diese im Zusammenhang mit REST-konformen HTTP und dem Thema dieser Arbeit eine wichtige Rolle spielen. Die Funktionsweise, der Nachrichtenaufbau und die Sicherheitsmechanismen sind im RFC7230 [**RFC2730**] erläutert.

HTTP-Methoden Für den REST Architekturstil legt Fielding formal fest, dass eine begrenzte Menge von Operationen existieren muss, die für alle Ressourcen gleichgültig sind [vgl. 6]. Diese

Operationen, auch Verben oder Methoden genannt, spielen für die Entwicklung von REST-konformen Anwendungen eine zentrale Rolle. In der HTTP Version 1.1 sind acht Standardmethoden definiert, GET, PUT, POST, HEAD, DELETE, OPTIONS, TRACE und CONNECT [s. 7], von denen nachfolgend nur die GET Operation kurz erläutert wird, da diese für das in dieser Arbeit behandelte Thema relevant ist. Detaillierte Beschreibungen der einzelnen Methoden sind dem RFC2616 [7] zu entnehmen.

Die **GET** Methode dient zur Anforderung einer durch die Request-URI identifizierte Entität (die Repräsentation einer Ressource). Nahezu jeder Aufruf einer sich im Web befindlichen Repräsentation erfolgt über einen GET-Request. Verfolgt man einen Link innerhalb eines Hypermedia-Dokuments im Browser, wird über HTTP ein GET-Request an den Server der sich im Hoheitsbereich des entsprechenden URIs befindet gesendet. GET ist *safe*, was bedeutet, dass der Nutzer mit dem Abruf von Daten durch einen GET-Request keinerlei Verpflichtungen eingeht. Da das mehrfache Aufrufen der selben Entität den gleichen Seiteneffekt bewirkt, ist GET zudem *idempotent*. Das Server-seitig gegebenenfalls beim Eintreffen eines GET-Requests ein Log-Eintrag getätigt wird, was als Zustandsänderung und somit als Seiteneffekt angesehen werden kann, hat keinen Einfluss auf die Idempotenz, da der Client keine Änderung des Zustandes angefordert hat. Weiterhin ist die Operation GET für das gesamte Web optimiert, wodurch sich beispielsweise die Semantik der GET-Methode durch das anfügen bestimmter Header ändert und somit unnötige Netzwerkauslastung vermieden wird. [Vgl. 43, S. 53 f.]

HTTP-Statuscodes Die *HTTP-Statuscodes*, oder auch *Response Status Codes*, sind im RFC7231 [10] definiert. Ein Statuscode ist eine dreistellige Zahl, welcher neben der HTTP-Protokollversion Bestandteil der Status-Line der HTTP Response ist. Die Status-Line ist wiederum Bestandteil jeder Response eines Servers auf einen HTTP-Request. Der Server kann mit Hilfe des Statuscodes dem Client mitteilen, ob die Verarbeitung der Anfrage erfolgreich war bzw. im Fehlerfall den Grund des Scheiterns spezifizieren. Die erste Stelle des Codes gibt dabei die Klasse des Status an, beispielsweise 2xx für eine erfolgreiche Operation. [Vgl. 10, S. 47 ff.]

Zustandslose Kommunikation Ein weiteres Kommunikationsprinzip von REST (und HTTP) ist die Zustand- oder Statuslose Kommunikation. Es besagt, dass sämtliche Anfragen an einen Dienst als unabhängige Transaktionen zu verarbeiten sind. Jede Anfrage muss ohne Bezug zu vorherigen behandelt werden. Dem Server darf der Client nur während der Verarbeitung einer Anfrage bekannt sein, danach muss er seine Existenz wieder vergessen. Der Zustand muss also entweder clientseitig gehalten oder in einen Ressourcenstatus konvertiert werden, was bedeutet, den Status als eigene Ressource zu kapseln. Der Vorteil einer zustandslosen Kommunikation ist die lose Kopplung des Clients an den Server und die damit verbundene vereinfachte Saklierbarkeit, da Anfragen von verschiedenen Serverinstanzen bearbeitet werden können. [Vgl. 43, S. 18 f.]

2.1.3. Ressourcen und deren Repräsentationen

Ressourcen sind ein zentrales Konzept von REST. Sie zeichnen sich durch genau zwei Eigenschaften aus: zum einen die eindeutige Identifikation und zum anderen das Vorhandensein einer oder mehrerer Repräsentationen, die dem Konsumenten der Ressource zur Verfügung stehen. Wird eine Ressource angefordert, wird sie nicht selbst, sondern nur eine ihrer Repräsentationen ausgeliefert. Eine Repräsentation ist die Darstellung einer Ressource und ist durch ein Format gekennzeichnet. [Vgl. 43, S. 35]

Man kann somit eine Ressource „als eine durch eine gemeinsame ID zusammen gehaltene Menge von Repräsentationen“[43, S. 35] definieren. Der Zweck einer Repräsentation, welcher auch die

namensgebende Rolle für REST spielt, ist die Übermittlung des Zustandes der Ressource oder einfacher gesagt: die Übermittlung von Informationen der zugrundeliegenden Ressource [vgl. 35, S. 141].

REST-konforme Architekturen werden von Ruby und Richardson in [34] auch als Resource Oriented Architecture (ROA) bezeichnet, da Ressourcen im Mittelpunkt eines Entwurfes stehen sollten. In [43] teilt Tilkov et al. Ressourcen in sieben Kategorien ein: Primärressourcen, Listenressourcen, Filter, Projektionen, Aggregationen, Aktivitäten und Konzepte. Primärressourcen sind Ressourcen, die sich schon beim frühen Entwurf als klassische Entitäten erkennen lassen [s. 43, S. 37 f.], beispielsweise eine Visitenkarte, eine Bestellung oder eine Grafik. Listenressourcen sind ebenfalls eigene Ressource und bestehen in der Regel aus einer Menge von Primärressourcen, wie beispielsweise eine Auflistung aller Bestellungen eines Kunden.

Die Aushandlung der Repräsentationsformate zwischen Client und Server mittels HTTP wird als **Content Negotiation** bezeichnet. Der Client kann in seiner Anfrage an der Webservice mit dem „Accept"-Header mitteilen, welche „Media Types" (auch bekannt als Multipurpose Internet Mail Extensions (MIME) Types) er präferiert und in der Antwort erwartet. Die IANA veröffentlicht eine Liste aller registrierten Medientypen im Web [11]. Die Codierung der Medientypen erfolgt nach RFC2046 [12]. Fehlt dieses Header-Feld im Request, wird dies als Akzeptanz aller Media Types interpretiert. Innerhalb der Response bzw. auch in einem Request mit Inhalt, wird über den „Content-Type"-Header angegeben, in welchem Repräsentationsformat die Daten im Message Body vorliegen. Kann der Server mit keiner Response antworten, die den im Accept-Header des Requestes angegebenen Medientypen entspricht, antwortet dieser mit einer durch den Statuscode 406 (not acceptable) gekennzeichneten Response. [Vgl. 7, S. 100 ff.]

Eines der populärsten Standardformate ist neben Extensible Markup Language (XML) [48] das JavaScript Object Notation (JSON) [40] Format [vgl. 43, S. 88]. Zu dessen Vorteilen zählen neben dem geringen Overhead und der damit verbundenen Leichtgewichtigkeit, auch die für den Menschen verständliche und übersichtliche Notation sowie die umfassende Unterstützung durch gängige Programmiersprachen [s. 19]. HAL [18], SIREN [39], Collection+Json [1] sind Beispiele für auf JSON aufsetzende Repräsentationsformate, die es um Hypermedia-Elemente erweitern, da JSON selbst keine standardisierte Darstellung von Links bietet [s. 43, S. 89].

Allgemein kann man nicht sagen, welche Repräsentationsformate zwingend von einem Webservice angeboten werden müssen, da dies von der jeweiligen Ressource und deren Semantik abhängt. Da sich jedoch der Nutzer der API voraussichtlich zunächst mit seinem Browser durch die URI-Struktur „klickt", ist es ratsam zumindest ein allgemeingültiges Format anzubieten. Der Medientyp „Hypertext Markup Language (HTML)" (`text/html`) gehört neben dem „einfachen Text" (`text/plain`) zu den am meisten akzeptierten Medientypen [vgl. 43, S. 101] und wird zusammen mit JSON auch in dem in dieser Arbeit entwickelten Webservice für jede Ressource angeboten.

2.1.4. URIs

> *Was macht aus einer Ressource eine Ressource? Sie muss mindestens einen URI haben.*[35]

Der Universal Resource Identifier (URI) ist zugleich Identifikator und Adresse einer Ressource. URIs sind dereferenzierbar. Deren Prinzipien wurden von Tim Berners-Lee als Axiome in seiner Ausarbeitung „Universal Resource Identifiers - Axioms of Web Architecture"[3] beschrieben. Ein URI identifiziert genau eine Ressource, eine Ressource kann jedoch von mehreren URIs identifiziert werden. Der Client kann in manchen Fällen einfacher auf eine Ressource verweisen, wenn

diese durch mehrere URIs benannt wurde. Um das damit verbundene Problem des erhöhten Wartungsaufwandes mehrerer URIs für eine Ressource bereitzustellen und sicherzustellen, dass diese auch alle auf die gleiche Ressource verweisen, kann man einen einzelnen URI als „kanonischen“ URI festlegen. Trifft ein Request auf eine der nicht kanonischen URIs ein, antwortet der Server mit dem Statuscode `303 (See also)` inklusive des kanonischen URIs. Der Client wird damit auf den kanonischen weitergeleitet. [Vgl. 35, S. 93 ff.]

Im allgemeinen Sprachgebrauch findet man die Verwendung des Begriffes Universal Resource Locator (URL). URLs sind eine Teilmenge von URIs, die speziell deren Standort und damit auch den Zugriffsmechanismus beschreiben. Sie zeigen, wo sich eine Ressource befindet. Der Begriff URI ist jedoch allgemeingültiger, da damit sowohl die Identifikation, der Name und Standort der Ressource gemeint ist. [Vgl. 4, S. 7]

Ein URI sollte eine klare Struktur aufweisen und so gut wie möglich vorhersagbar sein. Technisch gesehen, spielt die Struktur und Vorhersagbarkeit jedoch keine Rolle. Die generische Syntax eines URI ist im RFC3986 [4] definiert. Sie besteht aus einer hierarchischen Sequenz aus den Komponenten: Scheme, Authority, Path, Query und Fragment. Die Augmented Backus-Naur Form (ABNF) der Syntax ist wie folgt:

```
URI = scheme ":" hier-part [ "?" query ] [ "#" fragment ]
hier-part = "//" authority path
```

Nachfolgend wird kurz auf die einzelnen Komponenten eingegangen. Die Erläuterungen und Syntaxbeschreibungen sind größtenteils aus dem RFC3986 [4] entnommen, in dem auch detailliertere Informationen zu den einzelnen Bestandteile und deren Syntax nachgelesen werden können. Weiterhin wird in Kapitel 4.2 auf die innerhalb einer URI verwendbaren Zeichen des US-ASCII Zeichensatzes eingegangen.

Scheme Das Schema ist der erste Bestandteil jedes vollständigen URIs. Es definiert den Namensraum und somit den Typ des URI. Im Umfang dieser Arbeit wird ausschließlich das Schema *http* verwendet, was alle betrachteten URIs zu HTTP-URIs macht. Das Problem der Doppeldeutigkeit des Begriffes „URI-Schema“ wird im Kapitel 4.4 betrachtet.

Authority Die Komponente Authority beinhaltet den Namen des Hosts, optional von einer Portangabe gefolgt. Vor dem Hostnamen kann weiterhin die Angabe von Benutzername und Passwort erfolgen. Die entsprechnde Syntax lautet:

```
authority   = [ userinfo "@" ] host [ ":" port ]
userinfo    =  username [ ":" password]
```

Path Die Pfadkomponente enthält eine in hierarchischer Form angeordnete Sequenz von Pfad-Segmenten, die mit einem Slash (`"/"`) voneinander getrennt sind, ähnlich der Pfadangabe in einem Dateisystem. Bei der Angabe eine Pfades, beispielsweise innerhalb eines Link-Objektes eines HTML-Dokuments, unterscheidet man relative und absolute Pfade. Mit relativen Pfaden lassen sich Ressourcen, wie der Name es sagt, relativ zueinander adressieren. Dies bietet einige Vorteile, wie beispielsweise das Weglassen redundanter Pfadbestanteile oder die Möglichkeit, Schema- und Host-Komponente ohne Konsequenzen zu ändern. [Vgl. 43, S. 44]

Query Der Query-Anteil (auch Query-String) wird durch ein Fragezeichen (`?`) vom Pfadanteil getrennt. Er besteht aus einer Aneinanderreihung von Query-Parametern, welche als Key-Value-Paare definiert sind. Sie werden üblicherweise zur Übermittlung von HTML-Formulardaten oder

nicht-hierarchischen Daten verwendet. Im Kapitel 4.1 wird auf Vor- und Nachteile der Verwendung von Query-Parametern eingegangen.

Fragment Fragment-IDs sind analog zum Query ein optionaler Bestandteil eines URI. Das Auflösen der Fragment-ID erfolgt immer clientseitig. Üblicherweise wird sie zusammen mit dem HTML Anchor-Element (`<a ... />`) verwendet. Ist an einen URI das Fragment `#kapitel2` angefügt und in der HTML-Repräsentation der Ressource existiert ein Anchor-Element mit dem Attribut `name`, dessen Wert „`kapitel2`“ ist, scrollt der Browser nach dem Rendern der Seite zu diesem Element.

2.1.5. URI-Templates

Möchte man eine URI beschreiben, die sowohl aus fixen als auch variablen Anteilen besteht, verwendet man URI-Templates. Diese sind sehr umfangreich in einem eigenen RFC [14] definiert. In der Praxis und speziell im Bezug auf die Dokumentation einer REST-API, wird meist nur eine kleine Untermenge aller Mechanismen verwendet. Um eine Menge an URIs zu beschreiben, werden geschweiften Klammern gekennzeichente Platzhalter verwendet. [Vgl. 43, S. 46] Die URI

```
http://myrest.de/articles/{id}
```

kennzeichnet die Menge aller URIs, die aus dem Präfix `http://myrest.de/articles/` und einer ID bestehen. Oftmals ist es auch notwendig, mehr als ein variablen Anteil zu definieren, wie in folgendem Beispiel:

```
http://myrest.de/manufacturers/{man_id}/articles/{art_id}
```

Diese Schreibweise wird auch innerhalb der den Pfad betreffenden Annotationen des Jersey-Frameworks verwendet. Die damit beschriebene Struktur bzw. das Schema der URI wird auch in der Dokumentation des Webservices verwendet, der innerhalb dieser Arbeit entwickelt wird.

2.1.6. Hypermedia

Die Verknüpfung von Daten zu einem globalen Netz ist eine Kerncharakteristik des heutigen Internets. Der Mechanismus, der diese Verknüpfungen ermöglicht, heißt *Hypermedia*. Der Begriff ist eine Zusammensetzung aus *Hypertext*, der Verknüpfung eine Textes mit anderen Texten, und *Multimedia*. Anders als bei Hypertext, werden bei Hypermedia Texte mit multimedialen Daten verknüpft. Die zu verknüpfenden Daten sind somit keinesfalls nur Dokumente, sondern auch Bilder, Videos, Streams und andere Ressourcen. Betrachtet man die Interaktion des Menschen über einen Browser mit Webservern im Internet, spielt Hypermedia eine unentbehrliche Rolle. Da es fest zum Architekturstil des Web gehört, stellt es auch ein entscheidendes Prinzip des REST-Architekturstils dar.

Tilkov et al. hebt in [43, S. 71] hervor, wie essentiell der Hypermedia-Aspekt für einen Webservice ist, der die REST Konformität wahren möchte. Fielding selbst ist davon überzeugt, dass sich eine Anwendung nur als REST-konform bezeichnen darf, wenn sie Hypermedia voll umfänglich umsetzt. Er bezeichnet diese Architektureinschränkung als „hypermedia as the engine of application state“ [s. 6, S. 82], was sich sinngemäß als „Hypermedia als Motor des Anwendungsstatus“ übersetzen lässt. Oftmals findet man in Online-Quellen auch die für den englischen Wortlaut verwendete Abkürzung *HATEOAS*. Dessen Kernprinzipien sind die Verknüpfung von Ressourcen untereinander sowie die Steuerung des Anwendugszustandes. [Vgl. 43, S. 71 ff.]

Die Darstellung von Hypermedia-Elementen ist vom Repräsentationsformat der Ressource abhängig [vgl. 43, S. 81 f.]. Innerhalb eines XML Dokuments werden beispielsweise XLink-Attribute

[49] verwendet, wohingegen HTML Anchor-Elemente (siehe oben) nutzt. JSON bietet keine standardisierte Darstellung von Links, worauf in Kapitel 3.4.4 eingegangen wird. Unabhängig von der genauen Darstellung, sollte das Element eine Link-Relation beinhalten, da diese die Semantik des Links beschreibt [vgl. 43, S. 82 ff.]. Die Internet Assigned Numbers Authority (IANA) hat unter [23] eine Liste mit standardisierten Link-Relationen und deren Beschreibungen veröffentlicht. Unabhängig davon ist es in bestimmten Anwendungsfällen möglich (jedoch nicht ratsam) eigene Link-Relationen zu definieren.

Die nachfolgenden drei Abschnitte betrachten den Hypermedia-Aspekt unter verschiedenen Anwendungsfällen.

Hypermedia im Browser (Mensch-zu-Anwendung) Die zentrale Schnittstelle des Menschen zum Abruf und Einspeisung von Informationen von und in Instanzen des Web, ist der Browser. Seine Hauptaufgabe ist das durch den Benutzer ausgelöste Anfordern und die Interpretation von HTML-Dokumenten. Die Auszeichnungssprache HTML bietet zwei Elemente zur Umsetzung von Hypermedia, das Anchor-Element (`<a href... />`) und das HTML-Formular (`<form ... />`). Über das Anchor-Element wird dem Nutzer die Möglichkeit geboten, ein HTTP GET Request an den darin hinterlegten URI zu versenden. Es stellt somit die Verknüpfung zu der Ressource her, die durch diesen URI identifiziert wurde. Die Serverinstanz teilt somit gezielt dem Client mit, welchen Links er folgen kann. Das Formular-Element ermöglicht dem Nutzer die Eingabe von (meist textuellen) Daten. Aus diesen Daten wird ein URI konstruiert, an den ein GET oder POST Request versendet wird.[1] Mit einem Formular übermittelt der Server dem Client somit die Struktur, die der URI aufweisen muss. [Vgl. 43, S. 72]

HATEOAS im Browser (Mensch-zu-REST-Anwendung) Eine weitere Betrachtung von Hypermedia ist die Interaktion eines Menschen mit einen REST-konformen Webservice. Der Browser interagiert dabei ausschließlich mit den Repräsentationen von Ressourcen. Wird ein GET Request über den Browser an einen URI versendet, der eine Ressource des Webservices identifiziert, wird dessen HTML-Repräsentation zurückgeliefert. Der Nutzer kann den darin enthaltenen Links folgen oder Formulare verwenden, um die nächste Repräsentation einer Ressource anzufordern oder eigene Repräsentationen an den Server zu übermitteln. Der Client bewegt sich durch eine vom Server vorgegebene Menge an Repräsentationen. Welche jedoch explizit abgerufen werden, entscheidet der Client. Im Zusammenhang mit REST spielt auch die Steuerung des Anwednungszustandes eine zentrale Rolle. Jede Ressource hat zu jedem Zeitpunkt einen bestimmten Status auf den Server, den Ressourcenstatus. Zusammen mit dem Status der Ressourcenrepräsentation beim Client ergibt sich der Anwendungsstatus, was keinesfalls mit den Sitzungsstatus (siehe Abschnitt „Zustandslose Kommunikation“ in Kapitel 2.1.2) gleichzusetzen ist. Der Server kann mit Hypermedia-Elementen den Client mitteilen, welchen Zustand er als nächsten annehmen kann, d.h. die möglichen Statusübergänge werden mittels Hypermedia gesteuert. Beispielsweise enthält eine Ressource die Zusammenfassung eines Buchungsvorganges für einen Flug. Darin ist ein Link, der zur Stornierung dieser Buchung führt bzw. den Zustand von „gebucht“ in „storniert“ überführt. Ist er das Link-Element mehr vorhanden, ist keine Stornierung mehr möglich. Der Server kann somit gezielt die Interaktionsmöglichkeiten des Clients steuern. Nach Fieldings Interpretation, bewegt sich der Nutzer beim Interagieren mit „Webseiten“ somit von Anwendungsstatus zu Anwendungsstatus. [Vgl. 43, S. 75 f.]

[1] Über das Attribut `method` des HTML Form-Elementes kann festgelegt werden, ob die HTTP-Methode GET oder POST verwendet wird.

HATEOAS zwischen Webanwendungen (Anwendung-zu-REST-Anwendung) Betrachtet man nun die Kommunikation einer Client-Webanwendung mit einen REST-konformen Webservice, stellt man fest, dass es bei der Verwendung von Hypermedia keine wesentliche Rolle spielt, ob der Mensch oder eine Anwendung mit dem Dienst kommuniziert. Die Eigenschaften, die ein Webservice in Bezug auf Hypermedia aufweisen muss, um die REST Konformität zu wahren[2], sind identisch:

- die Menge aller Links innerhalb der Ressourcenrepräsentation spiegelt die Handlungsmöglichkeiten des Clients wider
- dem Client darf nicht zugemutet werden, URIs selbst zu bilden
- der Client benötigt keine externe Beschreibung, um den Webservice nutzen zu können
- der Server bietet Repräsentationsformate an, die der Client interpretieren kann
- die angebotenen Repräsentationsformate müssen Hypermedia unterstützen können

Entwickelt man die REST-API mit diesen Eigenschaften, indem man Hypermedia korrekt nach Fieldings Vorgaben [s. 6, 8] anwendet, ergeben sich folgende Vorteile[2]:

- Entkopplung des Clients vom Server
- Transparenz von Änderungen an der Server- bzw. URI-Struktur
- Server-seitige Steuerung des Anwendnungsflusses
- Metadaten müssen nicht ausgelagert werden, sie sind Bestandteil der Ressource (bzw. deren Repräsentation)

2.2. SQL

> *SQL ist heute der zentrale Zugriffsmechanismus auf alle relationalen Datenbanken. Unabhängig, wie sich eine Datenbank an der Oberfläche präsentiert, bildet letztendlich SQL die Schnittstelle für den Datenzugriff.* [52, S. 21]

2.2.1. Einordnung

Um Datensätze aus einer relationalen Datenbank abzufragen, benutzt man Abfragesprachen. Die Bekannteste deskriptive Abfragesprache stellt dabei Structured Query Language (SQL) dar, da sie aufgrund ihrer ISO Standardisierung von nahezu allen relationalen Datenbanksystemen unterstützt wird. Der aktuelle Standard (SQL:2008) ist so umfangreich und mächtig, dass er von bestehenden Implementierungen nur partiell abgedeckt wird. [Vgl. 30]

[2] Die REST konformen Eigenschaften sowie die damit verbundenen Vorteile wurden der Literatur „REST und HTTP“ [43, S. 76 f.] von Tilkov et al. entnommen.

2.2.2. Relationale Datenbanksysteme

In diesem Kapitel wird kurz auf Grundlagen relationaler Datenbanksysteme und die in dem Zusammenhang verwendeten Begriffe eingegangen. Für detailliertere Informationen zu relationalen Datenbanken und deren Eigenschaften wird auf die zahlreich vorhanden Literatur verwiesen.

E. F. Codd, Mitarbeiter des IBM-Forschungsinstitut in San Jose, California, veröffentlichte 1970 seinen Artikel namens „A Relational Model for Large Shared Data Banks“ [5], in der er grundsätzliche Prinzipien des relationalen Datenbankmodells und der relationalen Algebra definiert [vgl. 31]. „In relationalen Datenbanken werden die Daten in Tabellen oder Relationen gespeichert.“ [30] Eine Relation setzt sich zusammen aus einer bestimmten Anzahl von Spalten (auch: Attribut oder Feld) sowie einer Menge von Zeilen (auch: Tupel, Satz oder Datensatz). Der Grad einer Relation wird durch die Anzahl der Attribute bestimmt, die Kardinalität entspricht der Anzahl der Tupel. Die zulässigen Werte eines Attributes werden durch dessen Domain (Gültigkeitsbereich) definiert. Die Relation selbst ist durch ihren Namen, sowie aller enthaltenen Attribute und deren Domains beschrieben. Diese Struktur wird auch als *Datenbankschema* bezeichnet. Unter den Tupeln besteht keine vorgegebene Reihenfolge und es darf kein Tupel mehrfach vorkommen. Die Primärschlüsselbedingung des Relationen-Modells verlangt, dass für eine Relation ein Attribut oder eine Kombination von Attributen als Primärschlüssel (engl.: primary key) dient. Die Aufgabe des Primärschlüssels besteht darin, ein Tupel eindeutig zu identifizieren. Fremdschlüssel (engl. foreign key) realisieren dahingegen Beziehungen zwischen Relationen. Die zugrundeliegende Fremdschlüsselbedingung verlangt, dass in der referenzierten Relation ein entsprechender Primärschlüsselwert definiert sein muss. Dies wird auch als „referentielle Integrität“ bezeichnet. [Vgl. 30]

Datenbanken werden von einem Datenbankmanagementsystem (DBMS) verwaltet. Für relationale Datenbanken fungiert ein Relationales Datenbankmanagementsystem (RDBMS) als Verwaltungsinstrument. Es gewährleistet die Zuverlässigkeit von Abfragen und Änderungen. „[Es] legt fest, dass die Software alle Änderungen sowohl an den Daten als auch an der Struktur der Tabellen und anderer Objekte (Metadaten genannt) innerhalb von Transaktionen durchführt, die logisch zusammenhängende Aktionen beziehungsweise Befehle bündeln.“ [13] Das Akronym *ACID*, welches für die Begriffe Atomicity, Consistency, Isolation und Durability steht, beschreibt die bei der Verarbeitung von Transaktionen berücksichtigten Eigenschaften [vgl. 13].

2.2.3. Der Standard SQL

> *Neben der Unterstützung durch wichtige Hersteller von Software im Unternehmensbereich ist SQL auch dank seiner Standardisierung ein zentraler Pfeiler der Softwarearchitektur geworden.* [52, S. 22]

Das American National Standards Institute (ANSI) betreibt die Standardisierung von SQL. Die Standards werden entsprechend des Verabschiedungsjahres benannt, so besipielsweise SQL:1986, SQL:1989 bis hin zu SQL:2008 [vgl. 52, S. 22]. Über die deklarative Sprache SQL wird die Ergebnismenge einer Anfrage oder Manipulation beschrieben. Dabei wird nur spezifiziert, „was“ abgefragt bzw. manipuliert werden soll, jedoch nicht „wie“. Die Anweisungen werden vom DBMS in eine prozedurale Syntax übersetzt, damit sie von einem Prozessor abgearbeitet werden können. [Vgl. 22]

SQL ist eine einfach zu lesende, aber dennoch sehr mächtige Programmiersprache, deren Syntax wie bei jeder Sprache eine entscheidende Rolle spielt. Der Sprachraum von SQL wird üblicherweise in folgende Bestandteile aufgeteilt:

- **Data Definition Language (DDL)** Die Datendefinitionssprache. Sie wird benötigt, um die grundlegenden Strukturen der Datenbank (Tabelle, Felder, ...) zu modifizieren und zu löschen. [Vgl. 52, S. 23]
- **Data Manipulation Language (DML)** Die Datenmanipulationssprache. Sie wird benötigt, um in bereits vorhandenen Datenstrukturen Daten abzufragen, zu verändern, zu löschen und einzufügen. Sie kann aufgrund ihres Aufgabenbereiches noch einmal in die nachfolgenden zwei Gruppen aufgeteilt werden. [Vgl. 52, S. 23]
  - die Abfrage von Daten, rein lesender Zugriff (`SELECT`)
  - die Veränderung der Daten, schreibender Zugriff (`INSERT, UPDATE, DELETE`)
- **Data Control Language (DCL)** Die Datenkontrollsprache. Sie dient der Pflege der Datenbankinfrastruktur. [Vgl. 52, S. 23]

Neben der DDL und DCL, die hauptsächlich von Datenbankadministratoren verwendet werden, spielt besonders der lesende Zugriff der Daten über das `SELECT`-Kommando eine zentrale Rolle [vgl. 52, S. 23]. Diese Select-Abfragen und deren syntaktischer Aufbau sind im Bezug auf diese Bachelor Thesis von hohem Interesse und werden daher im nächsten Kapitel näher beleuchtet.

2.2.4. Die SELECT Syntax

Die SELECT-Anweisung von SQL dient der Gewinnung von Informationen aus relationalen Datenbanken, weshalb sie in der Praxis die meist verwendete und wichtigste Anweisung darstellt. Unabhängig von der Formulierung einer SELECT-Anweisung, wird eine Menge von Datensätzen identischer Art, d.h. mit festgelegten Feldern, aus einer oder mehreren Tabellen zurückgeliefert. Da das Ergebnis somit auch eine Relation darstellt, verfügt es selbst über Felder und Datensätze. Die Herausforderung besteht dabei, die Anweisung so zu formulieren, dass daraus die gewünschten Datensätze resultieren. Wieken bezeichnet die SELECT-Anweisung zutreffend als „einfachste und komplizierteste Anweisung“ [52, S. 59], da sie einerseits aus einer kleinen Menge klar strukturierter Bausteine besteht, jedoch andererseits eine immense Anzahl an möglichen Kombinationen dieser Bausteine aufweist. [Vgl. 52, S. 59]

Die SELECT-Syntax kann mittels Erweiterte Backus-Naur Form (EBNF), einem daraus generierten Railroad-Diagramm oder in selbst erklärender textueller Form, wie sie oft in Literatur zu finden ist, beschrieben werden. Zunächst wird die letztere Beschreibung verwendet, bei der optionale Elemente in eckigen Klammern und optionale Wiederholungen mit einem Stern angegeben werden. Die SELECT-Syntax (entnommen aus [52, S. 59]) lautet somit wie folgt:

`SELECT` [`DISTINCT|ALL`] *ausdrucksliste*
`FROM` *tabellenliste*
[*joinliste*] ∗
[`WHERE` *bedingungsliste*]
[`GROUP BY` *ausdrucksliste*
[`HAVING` *bedingungsliste*]]
[`ORDER BY` *ausdrucksliste* [`ASC|DESC`]]
[`LIMIT` *zahl*
[`OFFSET` *zahl*]]

In der Literatur (beispielsweise in [52] oder [37]) findet man häufig die Zusammenfassung der SQL-Kommandos mit den zugehörigen Argumenten zu sogenannten Klauseln. Jede Zeile der hier

dargestellten Syntax entspricht einer nach dem enthaltenen Kommando benannten Klausel. Die erste Zeile entspricht somit der Select-Klausel, die zweite der From-Klausel und so weiter. In Abbildung 1 ist die in Klauseln unterteilte Syntax, dargestellt als Railroad-Diagramm. Das Diagramm wurde aus der in EBNF verfassten Metasyntax (Anhang A.5) mit der Software „Railroad Diagram Generator“ [29] generiert. Es wird von Links beginnend gelesen.

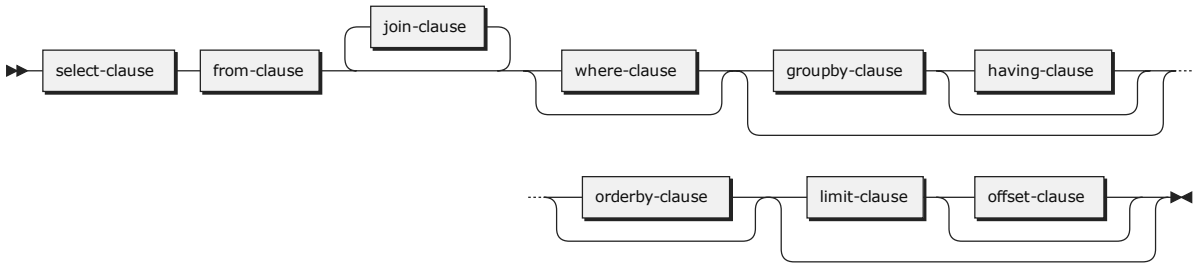


Abbildung 1: Railroad-Diagramm der SQL-Select Klauseln

Die Beachtung der angegebenen Reihenfolge der Syntaxelemente bzw. Klauseln ist essenziell [s. 52, S. 61]. Wird davon abgewichen, resultiert daraus eine ungültige Anweisung. Da ein grundsätzliches Verständnis des Aufbaus und der Eigenschaften jeder Klausel Voraussetzung für die in Kapitel 4 betrachteten Arbeitsschritte ist, wird in den nachfolgenden Abschnitten kurz darauf eingegangen.

Select-Klausel (select-clause)

Syntax: `SELECT` [`DISTINCT|ALL`] *feldname* [`AS` *feldalias*] {, *feldname* [`AS` *feldalias*]}

Die Select-Klausel bestimmt, welche Felder in die Ergebnismenge aufgenommen werden sollen, womit die Struktur der Ergebnistabelle bestimmt wird. Die optionalen Angabe `DISTINCT` verhindert Duplikate in den Datensätzen der Ergebnismenge. Manche Datenbankhersteller (beispielsweise Oracle) verwenden `UNIQUE` an Stelle von `DISTINCT`. `ALL` ist der Standardwert bei Select-Anfragen und kann somit weggelassen werden. Die Ausdrucksliste besteht aus Kommagetrennten Feldbezeichnern bzw. einen Asterisk (*), stellvertretend für alle Felder der in der From-Klausel angegebenen Tabellen. Ein Feldbezeichner kann sowohl eine Aggregatfunktion[3], als auch der Name des Feldes sein, wobei diesem auch der zugehörige Tabellenname mit einem Punkt (.) getrennt vorangestellt werden kann (beispielsweise *tabelle1.spalte2*). Optional kann für den Feldbezeichner auch ein Alias definiert werden. Das dieser Syntax zugehörige Railroad-Diagramm kann der Tabelle 7 im Kapitel 4.3 entnommen werden.

From-Klausel (from-clause)

Syntax: `FROM` *tabellenname* [`AS` *tabellenalias*] {, *tabellenname* [`AS` *tabellenalias*]}

Innerhalb der From-Klausel werden alle nicht in Verbindung stehenden Tabellen angegeben, auf die innerhalb der kompletten SELECT-Anweisung zurückgegriffen wird [vgl. 52, S. 60]. Weitere Tabellen, die mit diesen in Verbindung stehen, werden in den Join-Klauseln definiert. Für jeden Tabellennamen kann optional ein Alias vergeben werden.

[3] Eine Auflistung aller wichtigen Aggregatfunktionen mit deren Bedeutung kann der Literatur [52, S. 113] entnommen werden.

Join-Klausel (join-clause)

Syntax: [`FULL|LEFT|RIGHT|NATURAL`] `JOIN` *tabellenname2* [`AS` *tabellenalias*]
`ON` *tabellenname1.feld1* = *tabellenname2.feld1*
{ AND *tabellenname1.feldx* = *tabellenname2.feldy*}*

Mit Join-Klauseln können in Beziehung stehende Tabellen miteinander verbunden werden. Die Beziehung wird mit Hilfe von Fremd- und Primärschlüsseln realisiert, worauf im Umfang dieser Arbeit nicht weiter eingegangen wird. Die Angabe von Join-Klauseln ist optional. Es gibt verschiedene Typen eines Joins, zu den meist verwendeten gehören jedoch der `INNER JOIN` oder auch nur als `JOIN` bezeichnet, der `LEFT JOIN` und `RIGHT JOIN`, auch als `LEFT OUTER JOIN` und `RIGHT OUTER JOIN` bekannt, der `FULL JOIN` oder auch `FULL OUTER JOIN` und der `NATURAL JOIN`. Die Bedeutung der einzelnen Join-Typen können in der Literatur [52, S. 82 ff.] nachgelesen werden. Auf das dem Join-Typ entsprechende SQL-Kommando folgt der Name der zu verbindenden Tabelle. Optional kann für diesen ein Alias angegeben werden. Abschließend muss (außer beim `NATURAL JOIN`) nach dem `ON` Kommando die Verbindung der Tabellen über (mindestens) ein Paar von Feldern angegeben werden. Sollten mehrere Verbindungen bestehen, können diese logisch mit dem Kommando `AND` verknüpft werden. [Vgl. 52, S. 82 ff.]

Where-Klausel (where-clause)

Syntax: `WHERE` (*bedingung*) { `AND|OR` (*bedingung*)}

Wieken fasst die Aufgabe der Where-Klausel wie folgt zusammen: „Die WHERE-Klausel beschreibt die Bedingungen, die ein Datensatz erfüllen muss, damit er Teil der Ergebnismenge einer SELECT-Anweisung wird.“ [52, S. 76] Mehrere Bedingungen können mit dem logischen Verkettungsoperatoren `AND` oder `OR` aneinander gereiht werden. Eine einzelne Bedingung besitzt folgende Syntax:

Syntax (*bedingung*): (*feldname* Operator *ausdruck*)

Alle zugelassenen Operatoren sind in der Literatur [52, S. 80] nachlesbar und werden außerdem in Kapitel 4.2 und 4.3 näher betrachtet. Der Ausdruck kann sowohl ein numerischer, alphanumerischer Wert, als auch ein Ausdruck mit SQL-Platzhaltern (_ und %),auch Wildcards genannt, sein [vgl. 52, S. 81]. Eine Bedingung liefert WAHR oder FALSCH zurück. Durch die logische Verknüpfung aller Bedingungen untereinander, resultiert wiederum ein WAHR oder FALSCH als Gesamtergebnis. Sind die Bedingungen für einen Datensatz WAHR, wird er in die Ergebnismenge aufgenommen. [Vgl. 52, S. 76]

GroupBy-Klausel (groupby-clause)

Syntax: `GROUP BY` *feldname* { , *feldname*}

Alle identischen Datensätze der innerhalb der GroupBy-Klausel angegebenen Spalten, werden zu einem neuen Datensatz, dem Gruppendatensatz, zusammengefasst, bzw. gruppiert. Die Gruppe definiert sich über mindestens eine Gemeinsamkeit, welche hier der gleiche Wert in einem Datenfeld ist. Die in dieser Klausel angegebenen Felder müssen in den Tabellen, die in der From-Klausel und den Join-Klauseln angegeben worden sind, vorhanden sein. Erfolgt eine Gruppierung mit der GroupBy-Klausel, können innerhalb der Select-Klausel mit Aggregatfunktionen Berechnungen, beispielsweise das Zählen (`COUNT(*)`) oder Summieren (`SUM(*)`) aller gruppierten Datensätze, erfolgen. [Vgl. 52, S. 104 ff.]

Having-Klausel (having-clause)

Syntax: `HAVING` (*bedingung*) { `AND|OR` (*bedingung*)}

Mit der Having-Klausel kann die Ergebnismenge nochmals gefiltert werden, jedoch nachdem die Gruppierung mittels GroupBy-Klausel erfolgt ist. Die Having-Klausel setzt somit das Vorhandensein der GroupBy-Klausel voraus. Die Bedingungen werden analog zur Where-Klausel gebildet und auf jede Gruppe der GroupBy-Klausel angewendet. Erfüllt eine Gruppe die Bedingungen, wird sie in die Ergebnismenge aufgenommen. [Vgl. 52, S. 114 f.]

OrderBy-Klausel (orderby-clause)

Syntax: `ORDER BY` *feldname* [`ASC|DESC`] { , *feldname* [`ASC|DESC`] }

Die auf- bzw. absteigende Sortierung der Ergebnismenge wird durch Verwendung der OrderBy-Klausel ermöglicht. Die Namen der Felder, die als Sortier-Kriterium gelten, werden Komma getrennt aufgelistet. Für jede Spalte kann festgelegt werden, ob deren Werte aufsteigend (`ASC`) oder absteigend (`DESC`) sortiert werden soll, indem das entsprechende Kommando hinter dem Spaltennamen angegeben wird. Der Standardwert ist immer `ASC`. [Vgl. 52, S. 70 f.]

Limit-Klausel (limit-clause)

Syntax: `LIMIT` *anzahl*

Die Limit-Klausel begrenzt die Anzahl der Datensätze innerhalb der Ergebnismenge. Wird beispielsweise `LIMIT 10` an das Ende der Select-Anweisung angefügt, werden nur die ersten Zehn Datensätze der Ergebnismenge zurückgegeben.

Offset-Klausel (offset-clause)

Syntax: `OFFSET` *offsetwert*

Die Offset-Klausel setzt das Vorhandensein der Limit-Klausel voraus. Mit ihr können den angegeben Offset-Wert entsprechend viele Datensätze übersprungen werden. Mit der Angabe von `OFFSET 100` werden die ersten einhundert Datensätze der Ergebnismenge übersprungen.

Besteht für den Leser dieser Bachelor Thesis der Bedarf an tiefer greifenden Informationen zu den Klauseln, wird auf die Literatur „erfolgreich SQL lernen“ von J.-H. Wieken [52] verwiesen, worin alle Elemente der Syntax detailliert und leicht verständlich mit Beispielabfragen erläutert werden.

Neben einfachen Abfragen besteht auch die Möglichkeit, komplexe Abfragen bzw. Unterabfragen zu formulieren. In dieser Arbeit wird von deren Betrachtung abgesehen, da der damit verbundene Anstieg der Komplexität den Umfang des hier behandelten Themas übersteigen würde. Demnach werden ausschließlich einfache Abfragen behandelt.

2.2.5. JDBC

In fast allen Programmiersprachen gibt es Zugriffsmöglichkeiten auf relationale Datenbanken. Mit standardisierten Schnittstellen, wie JDBC, ODBC, Data Access Objects (DAO) oder Active Data Objects (ADO), kann die Kommunikation mit der Datenbank mittels SQL erfolgen [s. 52,

S. 17]. Da innerhalb dieser Arbeit der Zugriff auf eine Datenbank durch die Programmiersprache Java erfolgen soll (siehe Kapitel 3.1), wird in diesem Kapitel nur auf JDBC eingegangen.

In [45] definiert Ullenboom JDBC als „(...) einen Satz von Schnittstellen, um relationale Datenbanksysteme von Java zu nutzen.“ [45] Die JDBC-API [24] ermöglicht als einheitliche Programmierschnittstelle die Nutzung von Funktionen differierender Datenbanken in gleicher Weise. Zusammen mit den JDBC-Treibern erreicht die API eine wirksame Abstraktion von der angebundenen relationalen Datenbank. Die Implementierung des Datenbank-Protokolls ist ausschließlich dem Treiber bekannt, wodurch das Lernen verschiedener Zugriffsmethoden für Datenbanken unterschiedlicher Hersteller entfällt. [Vgl. 45]

Mit JDBC wir dem Programmierer die Möglichkeit geboten, „Verbindungen zu Datenbanken aufzubauen, Datensätze zu lesen oder neue Datensätze zu verfassen. Zusätzlich können Tabellen aktualisiert und Prozeduren auf der Serverseite ausgeführt werden“ [45]. Damit der Datenbankzugriff erfolgen kann, wird ein Treiber benötigt, der die Schnittstelle zwischen Programm und Datenbank bildet und die JDBC-API implementiert. Die Implementierung des Treibers ist Abhängig vom Hersteller, da er die von der Datenbank unabhängige API auf die konkrete Datenbank abbilden muss. [Vgl. 45]

3. Softwaredesign

In diesem Kapitel wird nach erfolgter Problemanalyse (Kapitel 1.1), Zieldefinition (Kapitel 1.2) und Technologiebeschreibung (Kapitel 2) das Softwaredesign entwickelt, welches die Grundlage für die Implementierung darstellt. Dabei werden zunächst die Architekturbausteine der Anwendung und deren Zusammenwirken mit externen Komponenten aufgezeigt. Im Anschluss werden die einzelnen Bestandteile der Anwendung separat designt.

3.1. Identifikation der Architekturbausteine

Neben dem internen Aufbau der Anwendung, wird auch auf die Komponenten eingegangen, mit denen die Anwendung interagiert. Die Abbildung 2 zeigt das Zusammenwirken dieser Bestandteile. Die Anwendung selbst soll als Desktop-Anwendung implementiert werden und besteht grundsätzlich aus drei Einheiten: Webserver-Instanz mit angebundenen Webservice, interner Anwendungslogik und Datenbankschnittstelle.

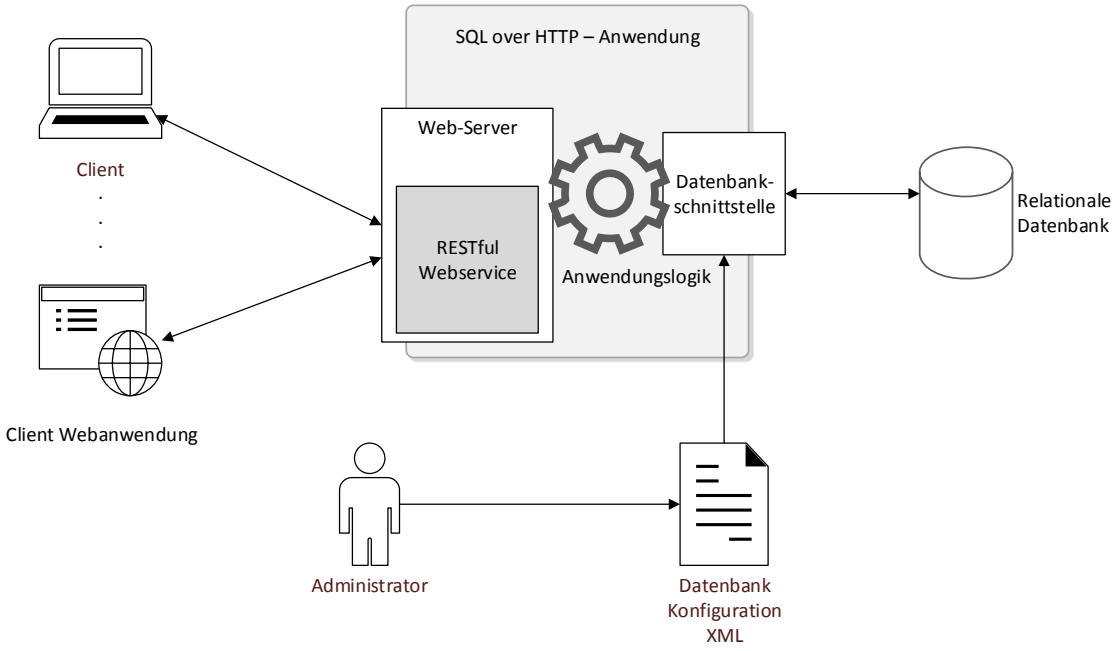


Abbildung 2: Architekturskizze der Anwendung „SQL over HTTP“

Die Aufgabe des Empfanges von Requests, sowie des Versendens von Responses übernimmt die Webserver-Instanz. Der angebundene Webservice, dessen Entwicklung in Kapitel 3.4 erläutert wird, nimmt diese Requests entgegen und veranlasst das Versenden der Responses. Er ist eng mit der Webserver-Instanz verknüpft und setzt das Vorhandensein dieser voraus. Sowohl ein Client, der mit seinem Browser auf die Anwendung zugreift, als auch eine Client-seitige Webanwendung, kommunizieren mittels HTTP mit dem Webserver und somit mit dem Webservice. Aufgaben der internen Anwendungslogik werden im nächsten Kapitel genannt. Zusammenfassend kann man sagen, dass sie als Vermittler zwischen Webservice und Schnittstelle zur Datenbank fungiert. Die Datenbankschnittstelle wird in Kapitel 3.3 erläutert. Sie stellt die Verbindung zur relationalen Datenbank her, kommuniziert bidirektional mit dieser und trennt die Verbindung beim Schlie-

ßen des Programms. Die benötigten Daten zum Herstellen der Datenbankverbindung, entnimmt die interne Logik der Konfigurationsdatei, die vom Administrator gepflegt wird. Der Name der Anwendung wurde auf „SQL over HTTP“ festgelegt.

Auswahl geeigneter Technologien Nach der Identifikation der einzelnen Architekturbausteine, müssen geeignete Technologien ausgewählt werden, die den ermittelten Anforderungen entsprechen und sich in die beschriebene Architektur eingliedern lassen. Diese Ermittlung ist notwendig, um die Anwendung im Nachgang vollständig implementieren zu können. Eine detaillierte Aufführung und Analyse aller für diesen Anwendungsfall geeigneten Technologien, mit einer anschließenden Entscheidungsfindung, würde den Rahmen dieser Arbeit deutlich übersteigen. Demnach werden nachfolgend nur die Technologien genannt, die die in Kapitel 1.3 erfassten Anforderungen erfüllen, sich während der Analyse als geeignet herauskristallisiert haben und demnach für die Implementierung der Anwendung verwendet werden. Die Auswahl ist Tabelle 2 zu entnehmen.

Tabelle 2: Ausgewählte Technologien und ihre Alternativen

Bestandteil	**Technologie**
Anwendung	Java SE - Java SE Development Kit (JDK) v. 1.7 [26]
Webserver	Glassfish Grizzly v. 2.3 [28]
RESTful Webservice	Jersey 2.17 (Referenzimplementierung von JAX-RS 2.0) [27]
Datenbankschnittstelle	JDBC [25]
Konfigurationsdatei	XML [48]
HTML Mapping	Mustache Templating [50]
JSON Mapping	Jackson [41]

Client-Server-Kommunikation Die Abbildung 3 beschreibt den vereinfachten Ablauf der Kommunikation innerhalb der 3-Tier Architektur. Durch das selbsterklärende Unified Modeling Language (UML) Sequenzdiagramm ist eine textuelle Beschreibung der Abläufe nicht notwendig.

3.2. Interne Anwendungslogik

Die interne Logik, in der Abbildung als Zahnrad gekennzeichnet, erfüllt folgende Aufgaben:

- Starten und Stoppen der Anwendung
- Protokollieren aller Ereignisse in eine Log-Datei
- Auslesen der Datenbank-Konfigurationsdatei
- Verarbeitung der Informationen eingegangenen Requests
- Übersetzen der URIs in eine SQL Select-Anweisung
- Anweisung über die Datenbankschnittstelle an die Datenbank übermitteln
- Entgegennehmen der Ergebnismenge an Datensätzen
- Erzeugen einer Ressource aus dem Abfrageergebnis
- Mappen der Ressource in das gewünschte Repräsentationsformat
- Repräsentation weiterreichen an den Webservice

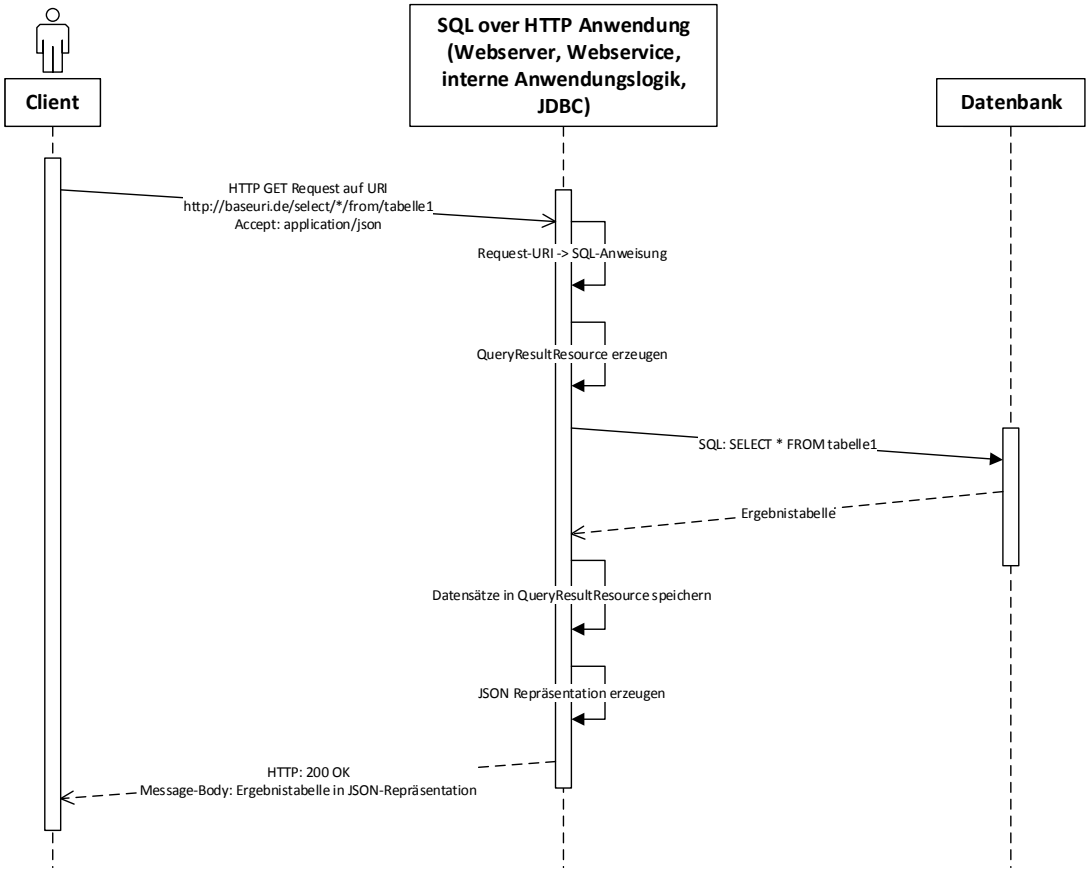


Abbildung 3: UML-Sequenzdiagramm der 3-Tier Softwarearchitektur

3.3. Datenbankanbindung

Zur ordnungsgemäßen Anbindung einer Datenbank mittels JDBC werden bestimmte Daten benötigt. Tabelle 3 beinhaltet eine Auflistung dieser Informationen mit jeweils einer kurzer Beschreibung und einem Beispielwert. Die aufgeführten Daten werden in einer Konfigurationsdatei festgelegt, damit die Anwendung von dem Datenbankhersteller entkoppelt wird. Es bietet sich an, diese im XML-Format anzulegen, da sie somit für den Menschen lesbar und von der Java-Anwendung mit internen Funktionen auslesbar ist.

Nach dem Start der Anwendung wird diese Konfigurationsdatei eingelesen. Die Werte werden auf Gültigkeit geprüft und Anwendungsintern gespeichert. Im Anschluss wird die Treiber-Klasse dynamisch geladen und die Verbindung zur Datenbank unter der angegebenen URI mit den Authentifizierungsdaten hergestellt.

Tabelle 3: Benötigte Angaben zur Anbindung der Datenbank

Information	**Beschreibung**	**Beispiel**
DB URI	Die URI zur Datenbank.	jdbc:h2:./db/bundesliga
Benutzername	Der Benutzername zur Authentifizierung beim RDBMS.	dbuser
Passwort	Das zum Benutzernamen zugehörige Passwort.	sicher!
Java-Archive mit JDBC-Treiber Klasse	Der Name und Pfad zur *.jar Datei, in der sich die JDBC-Treiber Klasse befindet.	./drivers/h2-1.4.188.jar
Klassenname des JDBC-Treibers	Der Name der JDBC-Treiber Klasse innerhalb der *.jar Datei.	org.h2.Driver

3.4. Architektur des RESTful Webservice

3.4.1. Daten in lesbare Ressourcen umwandeln

Bei der Konzeption eines Webservices beginnt man zunächst bei der Erfassung der Daten, die man über diesen bereitstellen möchte. Das sind in diesem Fall Relationen einer relationalen Datenbank samt ihrer Eigenschaften. Die Relation wird durch ihren eindeutigen Relationennamen identifiziert. Eine Tabelle ist die gebräuchliche Darstellungsform einer Relation, weshalb innerhalb dieser Arbeit der Begriff Tabelle verwendet wird. Die Spalten der Tabelle werden als Felder oder Attribute bezeichnet, die Zeilen als Datensätze oder Tupel. Welche konkreten Werte innerhalb eines Tupels der Tabelle gespeichert sind, ist irrelevant. Es zählt nur, in welchem Datenformat diese vorliegen. Das Relationenschema ist die Beschreibung einer gesamten Tabelle. Jedes einzelne Attribut ist grundsätzlich durch Namen und Datenformat gekennzeichnet.

Es stellt sich nun die Frage, welche dieser Daten der Nutzer des Webservices benötigt und wie diese als Ressourcen kombiniert werden. Sein Ziel ist es in erster Linie einzelne Werte aus einer oder mehreren Tabellen abzurufen, da diese verarbeitet und an den Anwender in geeigneter Form ausgegeben werden sollen. Die Menge dieser Werte wird durch eine Datenbankabfrage (auch Query) erzeugt. Das Ergebnis einer Abfrage ist wiederum eine Tabelle mit Feldern und zugehörigen Datensätzen. An dieser Stelle kann die erste Ressource als Primärressource definiert werden: das Ergebnis einer Datenbankabfrage, welche im Folgenden als *QueryResultResource* bezeichnet wird. Der Vollständigkeit halber beinhaltet sie zusätzlich die ausgeführte Anfrage in SQL Syntax, sowie die Angabe der Anzahl der enthaltenen Datensätze.

Erhält der Nutzer eine Repräsentation der *QueryResultResource*, erkennt er jedoch nicht ohne weiteres, von welchem Datentyp die einzelnen Werte sind. Da er diese Informationen jedoch benötigt, um eine korrekte Verarbeitung seinerseits zu ermöglichen, bietet die Ressource *QueryResultSchemaResource* das Schema der Ergebnistabelle. Jeder *QueryResultResource* ist somit eine *QueryResultSchemaResource* zugeordnet.

Um eine Anfrage zu formulieren, benötigt der Nutzer zunächst Kenntnisse über alle verfügbare Tabellen. Daraus kann die Listenressource *TableListResource* formuliert werden. Sie beinhaltet den Namen der Datenbank und eine Liste aller sich in dieser befindlichen Tabellen. Eine einzelne Tabelle innerhalb dieser Liste ist wiederum eine Ressource mit dem Namen *TableResource*, welche neben dem Namen der Tabelle eine Auflistung aller Primär- und Fremdschlüssel enthält. Alle Felder einer Tabelle sind mit Namen und Datentyp in der Listenressource *FieldListResource* aufgeführt. Die Eigenschaften eines einzelnen Feldes sind in der Primärressource

FieldResource enthalten. Dazu zählen: Name, Datentyp, textuelle Beschreibung, Default-Wert, Allow-Null Kennzeichnung, Auto-Increment Kennzeichnung, Längenangabe, Angabe der Präzision und Scale[4]. Die *FieldListResource* wird bezogen auf die Anzahl der Tabellen dynamisch gestaltet, wodurch der Nutzer beispielsweise alle Felder von Tabelle1, Tabelle2 und Tabelle3 abrufen kann. Da der Feldbezeichner im SQL Sprachgebrauch optional mit dem Tabellennamen als Präfix verwendet wird (beispielsweise *tabelle.feld*), stellt diese Dynamik kein Problem dar.

In Kapitel 3.4.6 wird entschieden, die Dokumentation des generischen URI-Schemas als eigene Ressource zu definieren. Die *DokuResource* beinhaltet den Aufbau des URI-Schemas mit entsprechenden textuellen Beschreibungen der Templates, Beispielantworten sowie zu erwartende Statuscodes und MIME-Types der Antworten.

Eine Übersicht aller Ressourcen kann Tabelle 4 entnommen werden. Die gewählten Ressourcennamen sind nur zur besseren Lesbarkeit aufgeführt. Die eigentliche „Benennung“[5] erfolgt im nächsten Abschnitt (3.4.2).

Tabelle 4: Übersicht Ressourcen

Name der Ressource	**Kurzbeschreibung**
TableListResource	Liste aller Tabellen der Datenbank
TableResource	einzelne Tabelle
FieldListResource	Liste aller Felder einer oder mehrerer Tabellen
FieldResource	einzelnes Feld
QueryResultResource	Ergebnis einer SQL Anfrage als Tabelle
QueryResultSchemaResource	Schema des Ergebnisses einer SQL Anfrage als Tabelle
DokuResource	Dokumentation des generischen URI-Schemas

3.4.2. Identifikation durch URIs

In Kapitel 2.1.4 ist die Verwendung einer URI als Ressourcen-ID erläutert. Die Ressourcen lassen sich, abgesehen von der *DokuResource*, sinngemäß in zwei Gruppen einteilen. Zum einen die Ressourcen, die Informationen zum Datenbankschema beinhalten (*TableListResource, TableResource, FieldListResource* und *FieldResource*) und zum anderen die, die in den Bereich der SQL-Abfrage fallen (*QueryResultResource* und *QueryResultSchemaResource*). In diesem Abschnitt werden den Ressorcen der ersten Gruppe ihre Identifikatoren zugewiesen. Da die Konzeption des URI-Schemas für die zweite Gruppe deutlich komplexer ist, wird sie im Kapitel 4 separat behandelt. Als Resultat entsteht ein REST konformes URI-Schema, das ein Hauptmerkmal des Webservices sein wird. Als Basis-URI wird *http://sql.example.net/* festgelegt. Zur Vereinfachung werden im Folgenden nur noch die zur Basis-URI relativen URIs genannt.

`/tables`

wird der Listenressource *TableListResource* zugewiesen. Geht man nach dem Zitat von Richardson aus Kapitel 4, sollten URIs sich an bestehende Konventionen halten. Genau wie bestehende Konventionen können intuitive Merkmale zur sinnvoll und geeignet wirkenden Erscheinung einer URI beitragen. Wenn man eine Auflistung aller Tabellen einer Datenbank anfragen möchte, erscheint die URI `/tables` zunächst so sinnvoll wie trivial. Die Pluralform von *table* lässt auf eine Menge von Tabellen schließen.

[4] Der *precision*-Wert gibt die Anzahl aller Ziffern an, der *scale*-Wert nur die Anzahl der Ziffern rechts vom Dezimaltrennzeichen [vgl. 38].

[5] Richardson bezeichnet in [35] den Vorgang der Zuweisung einer URI zu einer Ressource als Benennung.

`/tables/{tablename}`

wird der Primärressource *TableResource* zugewiesen. Der Platzhalter `{tablename}` dient als dynamische Pfadvariable für den Namen der Tabelle. Diese Fokusinformation [s. 35, S. 134] passt in die Hierarchie: an oberster Stelle die Liste aller Tabellen, darunter jede einzelne Tabelle.

`/tables/{tablename}/fields`

wird der Listenressource *FieldListResource* zugewiesen. Auch hier ist bei der ersten URI die Hierarchie erkennbar, da die Auflistung alle Felder der darüber aufgeführten Tabelle `{tablename}` beinhaltet.

`/tables/{tablename}/fields/{fieldname}`

wird der Primärressource *FieldResource* als unterstes Glied der Hierarchie zugewiesen. Dieses URI-Template beschriebt die Menge aller URIs, die aus einer Kombination aller verfügbaren Tabellen und deren Felder besteht. Enthält die Datenbank x Tabellen und jede Tabelle weist y Attribute auf, enthält die Menge $x * y$ verschieden URIs, wobei die Tatsache noch nicht berücksichtigt ist, dass für die Pfadvariable `{tablename}` auch mehrere Tabellen angegeben werden können.

`/documentation`

und

`/`

werden der Ressource *DokuResource* zugewiesen, wobei erstere die kanonische URI darstellt (siehe Kapitel 2.1.4). Wird ein Request auf die zweite, nicht kanonische URI gestellt, wird der Aufrufer auf die kanonische umgeleitet. Durch eine Anfrage an die Basis-URI ohne eine Pfadangabe gelangt der Nutzer somit zur Dokumentation des Webservice und erhält alle Informationen zu den möglichen Pfaden.

Das entstandene URI-Schema zur Identifikation der Ressourcen des Datenbankschemas wird zur besseren Veranschaulichung in Abbildung 4 als Railroad Diagramm dargestellt. Die beschrifteten Knoten kennzeichnen jeweils ein Pfadbestandteil der URI. Steht die Bezeichnung innerhalb eines Knotens mit abgerundeten Ecken, ist dies ein statisch festgelegter Bestandteil des Pfades, steht sie innerhalb eines rechteckigen Knotens, ist es ein dynamischer Pfad-Parameter. Verfolgt man das Diagramm angefangen von der Basis-URI (innerhalb der Abbildung als „baseUri“ bezeichnet) bis hin zum Ende, erhält man eine gültige URI. Weiterhin kann es auch nach bekannter textueller Schreibweise angegeben werden:

`/tables[/{tablename}[/fields[/{fieldname}]]]`

Dieser Teil des URI-Schemas wird außerdem innerhalb der API Dokumentation detailliert erläutert. Die zugehörige EBNF ist dem Anhang A.5 zu entnehmen.

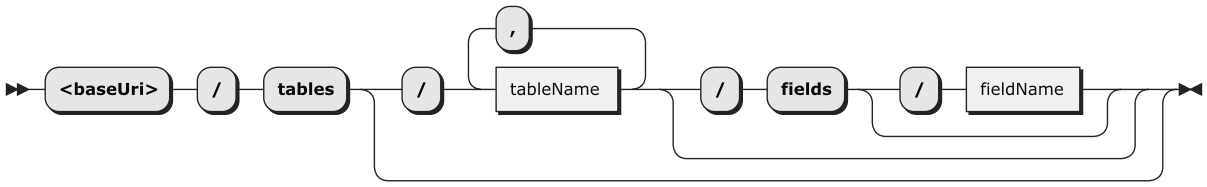


Abbildung 4: URI-Schema zum Abfragen der Datenbankinformationen als Railroad Diagramm

3.4.3. Auswahl und Entwurf der Repräsentationen

Im Anschluss an die Ressourcen- und URI-Festlegung muss sich entschieden werden, welche Daten in welchem Format an den Client gesendet werden, wenn dieser eine Ressource anfordert [s. 35, S. 140]. In Kapitel 2.1.3 wird neben dem Zusammenhang zwischen Repräsentationen und Ressourcen auch auf standardmäßige Medientypen Bezug genommen. Für den in dieser Arbeit zu entwickelnde Webservice wurde sich für die Verwendung der Formate *HTML (text/html)* und *JSON (application/json)* entschieden.

HTML ist eine allgemeingültige textuelle Auszeichnungssprache die den Zweck verfolgt, sowohl semantische Beschreibungen des Inhalts zu bieten als auch das Dokument hierarchisch zu strukturieren [vgl. 36, S. 7]. Eine für REST prädestinierte Eigenschaft von HTML sind die Anker-Elemente (<a href=...>), die von allen Browsern als Verknüpfungen (auch Links) zu anderen Ressourcen dargestellt werden. Durch das Anbieten des HTML Repräsentationsformats, kann der Nutzer die Anwendung bzw. den Webservice mit einem Browser mittels Hypermedia erkunden. Außerdem „[...] kann man wohlgeformtes HTML auch wunderbar als XML prozessieren“ [35, S. 143] wodurch auch Client-Anwendungen die HTML Repräsentation verarbeiten können.

JSON hingegen bietet sowohl den semantischen als auch den hypermedialen Aspekt nicht. „Darüber hinaus unterstützt JSON weder Namensräume noch eine standardisierte schemabasierte Validierung.“ [43, S. 89] Dennoch hat JSON mit dem Fokus auf Datenstrukturen den Vorteil, dass es in den gängigsten Programmiersprachen sehr einfach zu verarbeiten ist [s. 43, S. 89], wodurch es als zweites Repräsentationsformat aller Ressourcen gewählt wurde.

Der Webservice bietet mit der *DokuResource* insgesamt sieben verschiedene Ressourcen an. Da jede Ressource (außer der ausschließlich im HTML-Format vorliegenden *DokuResource*, siehe 3.4.6) in zwei Repräsentationen angeboten werden, würde eine Auflistung aller zwölf Templates der Repräsentationen den Umfang dieser Arbeit übersteigen. Daher werden in diesem Kapitel nur die beiden Vorlagen der *QueryResultResource* aufgeführt und erläutert, da diese die wichtigste Rolle innerhalb dieser Arbeit spielt. Die restlichen 10 sind auf der CD im Anhang vorhanden.

```
1 {
2    "statement":"The SELECT statement, with which the database was queried",
3    "dataset_count": the row count of the query result as integer,
4    "datasets":[
5       "field name":["value 1","value 2",...more values],
6       ... more datasets
7    ]
8 }
```

Listing 1: JSON Repräsentation der QueryResultResource

Im Listing 1 ist das JSON-Template der *QueryResultResource* angegeben. Es beinhaltet drei JSON-Objekte mit den statischen Keys „statement“, „dataset_count“ und „datasets“. „statement“ weist als Value einen String mit der ausgeführten SQL Anfrage auf. Das Objekt „dataset_count“ beinhaltet die Anzahl der Datensätze des Abfrageergebnisses als Dezimalzahl. Das Objekt „datasets“ ist ein JSON-Array, welches wiederum JSON-Arrays entsprechend der Anzahl der angefragten Felder beinhaltet. Als Key ist der Name bzw. der Alias, als Value die Werte des jeweiligen Feldes definiert. Die Darstellung der Abfrageergebnisse erfolgt somit Spalten-orientiert. Zusätzlich soll die Möglichkeit bestehen, die resultierende Tabelle auch nach dem Zeilen-orientierten Ansatz auszuliefern. Hier bietet es sich an einen Query-Parameter zu definieren, der optional an jede URI die eine *QueryResultResource* identifiziert angehängt werden kann. Als Name des Query-

Parameters wird „orientation“ festgelegt. Mögliche Werte sind *row* für eine Zeilen-orientierte oder *col* für eine Spalten-orientierte Ausgabe. Wird kein Query-Parameter angegeben erfolgt die Ausgabe standardmäßig Spalten-orientiert.

```
<!DOCTYPE html>
<html>
   <head>
      <title>Query Result</title>
   </head>
   <body>
      <h2>Query Result</h2>
      <h3>statement</h3>
      <span>{{query-statement}}</span>
      <h3>query result</h3>
      <table>
         <tr>
            <!-- repeat over all field names -->
            {{#field-names}}
            <th>{{fieldName}}</th>
            {{/field-names}}
            <!-- end:repeat -->
         </tr>
         <!-- repeat over all datasets rows -->
         {{#dataset-rows}}
         <tr>
            <!-- repeat over all field-values in the given row -->
            {{#.}}
            <td>{{.}}</td>
            {{/.}}
            <!-- end:repeat -->
         </tr>
         {{/dataset-rows}}
         <!-- end:repeat -->
      </table>
   </body>
</html>
```

Listing 2: Template der QueryResultResource HTML Repräsentation

Die Vorlage für die HTML-Repräsentation weist eine einfach gehaltene HTML-Struktur auf, um den Overhead der Markups möglichst gering zu halten. Da die HTML-Repräsentation in erster Linie für den Menschen vorgesehen ist und eine Formatierung der Inhalte für den Betrachter sich als vorteilhaft erweist, wird als Stylesheet-Bibliothek „Bootstrap“ [44] von Twitter eingebunden und dessen class-Attribute an die zu formatierenden Markups angefügt. Im Listing wird jedoch aus Übersichtlichkeitsgründen auf CSS-Notationen verzichtet. Der Inhalt dieser Repräsentation soll, identisch zur JSON-Repräsentation, die Daten der Ressource widerspiegeln, wodurch eine dynamische Generierung des HTML-Dokuments später bei der Implementierung berücksichtigt werden muss. Aus diesem Grund wurde die HTML-Vorlage mit Platzhaltern versehen, die sich an den doppelten geschweiften Klammern erkennen lassen. Diese Notation entspricht die der Mustache-Templating-Engine [50], welche eine weit verbreitete und leichtgewichtige Java Application Programming Interface (API) zur Generierung von Dokumenten aus Vorlagen darstellt und deshalb in diesem Belangen verwendet werden soll. Eine Dokumentation der Notationen kann dem Handbuch auf der Webseite des Herstellers (s. [51]) eingesehen werden. Zum

Verständnis dieser Vorlage genügt es jedoch folgende Fakten zur Syntax zu wissen:

Ein Platzhalter ohne inneren Präfix, wie zum Beispiel {{query-statement}} (Zeile 9, Listing 2) entspricht einen einfachen String mit dem in geschweiften Klammern angegebenen Namen. Ein Platzhalter mit einer Raute (#) als inneren Präfix, beispielsweise {{#links}} (Zeile 13), entspricht einer Schleife, die sich entsprechend der Länge des in geschweiften Klammern angegebenen Collection wiederholt. Das Ende des zu wiederholenden Bereichs wird durch einen Slash (/) als Präfix gekennzeichnet (siehe Zeile 15). Innerhalb dieses Bereichs können wiederum einzelne Eigenschaften eines Objektes der Collection mit den bekannten Präfix-losen Platzhaltern realisiert werden. Aus Gründen der Übersichtlichkeit wurden diese Schleifen mit Kommentaren versehen.

Das Resultat einer Datenbankabfrage wird als Tabelle zurückgegeben, demnach erscheint es sinnvoll auch dessen Darstellung in Tabellenform vorzunehmen, wofür sich die Verwendung des HTML table-Elements eignet. Die durch das <th>-Tag gekennzeichneten Spaltenbeschriftungen enthalten den Spaltennamen oder falls vorhanden dessen Alias-Namen. Durch den selbsterklärenden Charakter der Platzhalter und deren Kommentare ist eine nähere Beschreibung der restlichen Bestandteile überflüssig.

3.4.4. Hypermediaverweise

In Kapitel 2.1.6 wird die hohe Relevanz von Hypermedia innerhalb einer REST-konformen Webanwendung erläutert. Dieses Kapitel befasst sich nun mit der Anwendung dieses Architekturmerkmals innerhalb des zu konzipierenden Webservices. Als Zielstellung gilt die sinnvolle und strukturierte Verbindung aller Ressourcen durch Hyperlinks. Um dies zu erreichen müssen innerhalb der Ressourcenrepräsentationen Hypermedia-Elemente verwendet werden, über die der Nutzer zu andere, an die ausgelieferte Ressource anknüpfende Ressourcen gelangt. Bevor auf die Verwendung dieser Elemente eingegangen wird, muss zunächst festgelegt werden, welche Ressourcen miteinander verknüpft werden sollen.

In Abbildung 5 sind die Verlinkungen der Ressourcen als Graph dargestellt. Zu den klassischen Vorgehen zählt zunächst die hierarchisch bedingte Verknüpfung der Primär- und Listenressourcen unter- und miteinander. Die *TableListResource* beinhaltet somit je Tabelle einen Link auf die entsprechende *TableResource*. Diese wiederum enthält einen Link zur *FieldListResource*. Darin verweist jeder Listeneintrag auf die dem Eintrag entsprechende *FieldResource*. Bis auf die sich an der höchsten Hierarchieebene befindliche *TableListResource*, beinhalten alle Ressourcen wiederum eine Verknüpfung auf die hierarchisch übergeordnete. Diese von Tilkov in [43, S. 78] als intuitivste Form von Hypermedia bezeichnete, setzt wie beim konzeptionellen Objektmodell die Beziehungen der Objekte in Verknüpfungen um.

Jede Repräsentation beinhaltet weiterhin grundsätzlich eine Verknüpfung auf die Ressource, die sie präsentiert. Aus Gründen der Übersichtlichkeit würden die Kanten, die diese Gegebenheit widerspiegeln, nicht im Graphen dargestellt.

Die *QueryResultResource* wird mit allen anderen Ressourcen (abgesehen von der *FieldResource*) verlinkt, da dem Nutzer die Möglichkeit geboten werden soll, sich nach einer getätigten Abfrage über die Tabellen und deren Attribute zu informieren. Weiterhin beinhaltet sie einen Link auf die *QueryResultSchemaResource* (das zugehörige Schema des Abfrageresultats) und umgekehrt, sowie auf die *DokuResource*.

Durch dieses gut vermaschte Hypermedia-Modell kann sich der Nutzer ausgehend von der *TableListResource* durch alle Ressourcen des Webservice bewegen. Durch die kaum erfassbar große Menge an URIs, die die *QueryResultResource* identifizieren, muss an dieser Stelle auf einen damit verbundenen Kritikpunkt an der REST-Konformität im Kapitel 3.4.7 hingewiesen werden.

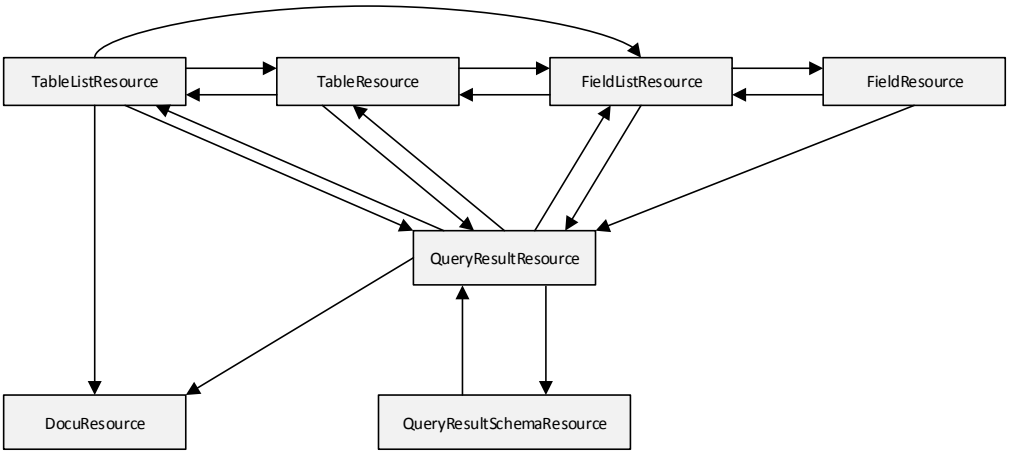


Abbildung 5: Übersicht Hypermedia Verlinkungen der Ressourcen

Die Darstellung von Hypermedia-Verweisen ist abhängig von dem Format der Ressourcenrepräsentation (siehe Kapitel 2.1.6). Im vorherigen Kapitel wurden HTML und JSON als Repräsentationsformate der einzelnen Ressourcen festgelegt. HTML bietet mit dem Anchor-Element eine standardisierte Darstellungsform, die sowohl die Link-Relation über das `rel`-Attribut, als auch den URI über das `href`-Attribut beinhaltet.

JSON bietet im Gegensatz zu HTML keine standardisierte Darstellung von Links. Da es trotzdem als sehr beliebtes Format im REST Umfeld gilt, wurden verschiedene auf JSON basierende Repräsentationsformate entwickelt, die Hypermedia unterstützen. In Kapitel 2.1.3 sind einige dieser Formate genannt. Da der hier zu entwickelnde Webservice ein sehr überschaubares Aufkommen von Hypermedia-Elementen aufweist, wird von der Verwendung eines dieser Formate abgesehen und eine eigene, leichtgewichtige und selbsterklärende Darstellung verwendet, welche in Listing 3 aufgeführt ist. Das Element mit dem Namen „links“ kann mehrere JSON-Objekte enthalten, die durch die Link-Relation (`rel`), den URI (`href`) und die HTTP-Methode (`method`) gekennzeichnet sind. Die Bezeichnungen sind an die Attribute des HTML Anchor-Elementes angelehnt. Unabhängig davon kann der Webservice bei Bedarf um eines dieser angepassten JSON Formate erweitert werden.

```
"links":[
    {
        "rel":"link-relation",
        "link":"http://base-uri.de/path",
        "method":"GET"
    },
    ...
]
```

Listing 3: Darstellung des Link-Elementes im JSON Format

Verweist ein Link-Element auf eine Ressource, muss eine Link-Relation festgelegt werden. Darüber erkennt der Client (bzw. die Client-Anwendung) den Typ des Links bzw. die Beziehung der

aktuellen mit der zu verlinkenden Ressource. In Tabelle 5 sind die Relationen für die oben genannten Verlinkungen der Ressourcen untereinander aufgeführt, sowie einer kurzen Beschreibung. Die dritte Spalte dient der Kennzeichnung, ob die Link-Relation von der IANA standardisiert ist oder eigens für diesen Webservice festgelegt wurde.

3.4.5. Fehlersituationen

Bei der Konzeption der REST-Architektur müssen auch ungültige Requests berücksichtigt werden. Tritt eine Fehlerbedingung ein wird je nach Art des Fehlers mit einem Response Code im Bereich 3xx, 4xx oder 5xx geantwortet. Damit der Nutzer den Fehler genauer identifizieren kann, wird im Message-Body der Response eine kurze textuelle Fehlerbeschreibung angegeben. [Vgl. 35, S. 158]

Drei mögliche Fehlerbedingungen für den hier betrachteten Webservice wären:

- Der Nutzer versucht Informationen über eine Instanz abzurufen, die nicht innerhalb der Datenbank existiert. Hier wird verstanden, wonach der Client fragt, die Daten sind jedoch nicht vorhanden. Für diesen Fall ist der korrekte Response Statuscode `404 (Not Found)`. [Vgl. 35, S. 159]
- Der Nutzer formuliert eine URI, die nicht dem Gültigkeitsbereich des URI-Schemas entspricht oder macht syntaktische Fehler, wie beispielsweise die Verwendung eines Leerzeichens innerhalb der URI. Hierbei wird mit dem Response Code `400 (Bad Request)` geantwortet.
- Bei der internen Verarbeitung des Requests oder bei der Interaktion mit der Datenbank tritt ein beliebiger Fehler auf. Der Statuscode `500 (Internal Server Error)` ist für diesen Fall zu verwenden.

3.4.6. Dokumentation

Für jedes System ist eine Dokumentation erforderlich. Sowohl der Entwickler einer Clientanwendung als auch ein Endanwender benötigen eine umfangreiche Dokumentation der RESTkonformen Webanwendung, um deren Funktionen korrekt verwenden zu können. Durch die Vereinigung von dynamischen Diensten und Dokumenten in einem einzigen konzeptionellen Modell ergeben sich Besonderheiten, die Tilkov in [43, S. 157] wie folgt zusammenfasst:

- Dokumente haben den Status eigenständiger Ressourcen. Hierzu zählen auch die, die ihre Ressourcen beschreiben. Die Dokumentation selbst ist somit auch eine Ressource, was bei der Konzeption der hier betrachteten Webanwendung schon berücksichtigt wurde (siehe Kapitel 3.4.1).
- Hypermedia verbindet die Navigation innerhalb der Dokumentation mit der Navigation der dynamischen Ressourcen. Die Ressourcen sind sozusagen bidirektional mit der Dokumentation verknüpft. Auch diese REST-konforme Besonderheit wurde umgesetzt (siehe Kapitel 3.4.4)

Man erkennt, dass die REST-konforme Architektur direkt an das aktive System und dessen individuelle Daten gekoppelt ist [vgl. 43, S. 157].

Zur Konzeption der Dokumentation gehört weiterhin die Festlegung, welche Informationen durch diese bereitgestellt werden sollen. Folgende Anforderungen können in diesem Zusammenhang an die Inhalte der Dokumentation gestellt werden:

Tabelle 5: Übersicht der Link-Relationen

Relation	**Kurzbeschreibung**	**IANA-Standard?**
`query`	Kennzeichnet einen Link, der auf die *QueryResultResource* verweist.	nein
`schema`	Kennzeichnet einen Link, der auf die *QueryResultSchemaResource* verweist.	nein
`help`	Kennzeichnet einen Link, der auf die *DocuResource* verweist.	ja
`tables`	Kennzeichnet einen Link, der auf die *TableListResource* verweist.	nein
`table`	Kennzeichnet einen Link, der auf eine *TableResource* verweist.	nein
`fields`	Kennzeichnet einen Link, der auf eine *FieldListResource* verweist.	nein
`field`	Kennzeichnet einen Link, der auf eine *FieldResource* verweist.	nein
`self`	Kennzeichnet den Link, der auf die Ressource der aktuellen Repräsentation verweist.	ja
`next`	Sollte innerhalb der *QueryResultResource* eine Ergebnismenge zurückgeliefert werden, deren Anzahl der Datensätze durch das LIMIT und OFFSET Kommando beschränkt wurde, kennzeichnet das Vorhandensein eines Links mit dieser Relation, dass noch weitere Datensätze in der Ergebnismenge vorhanden sind. Der enthaltene Link verweist auf eine *QueryResultRessource*, deren Status die nachfolgende Menge an Datensätzen beinhaltet.	ja
`prev`	Sollte innerhalb der *QueryResultResource* eine Ergebnismenge zurückgeliefert werden, deren Anzahl der Datensätze durch das LIMIT und OFFSET Kommando beschränkt wurde und das Argument des OFFSET-Kommandos größer als 0 ist, kennzeichnet das Vorhandensein eines Links mit dieser Relation, dass noch weitere Datensätze in der Ergebnismenge vorhanden sind, die sich vor der aktuellen Menge befinden. Der enthaltene Link verweist auf eine *QueryResultRessource*, deren Status die voranstehende Menge an Datensätzen beinhaltet.	ja

- das gesamte URI-Schema muss abgebildet werden
- alle Pfad- und Query-Parameter müssen beschrieben und mit einem Beispiel verdeutlicht werden
- mögliche Responses müssen mit Statuscode, MIME-Format und eventuell verfügbaren Beispiel-Inhalt angegeben werden
- weiterführende textuelle Beschreibungen zu einzelnen Komponenten müssen angegeben werden können

Eine Dokumentationsform, die alle aufgeführten Anforderungen erfüllt, ist RESTful API Modeling Language (RAML) [53]. RAML ist eine auf YAML[6] [2] basierende Sprache, um REST-konforme APIs zu beschreiben. Das vollständige RAML-Dokument entspricht der Ressource *DocuResource*. Da die Dokumentation in erster Linie für den Anwender gedacht ist, bietet sich als Repräsentationsformat HTML an. Mit dem Werkzeug *raml2html* [32] von Kevin Renskers, kann ein RAML-Dokument in ein interaktives HTML-Dokument konvertiert werden, wodurch die umfangreiche Dokumentation übersichtlich und ansprechend gestaltet wird. Weiterhin wird RAML selbst als Repräsentationsformat angeboten. Der zugehörige MIME-Type ist `application/raml+yaml`, welcher jedoch (noch) nicht in die Liste der standardisierten Medientypen der IANA aufgenommen wurde.

Der Aufbau und die Komponenten des RAML-Formates sind ausführlich auf der Internetpräsenz der RAML-Workgroup [53] erläutert und werden im Umfang dieser Arbeit nicht näher beleuchtet. Anhang A.1 zeigt einen Auszug aus der RAML-Dokumentation für das in diesem Kapitel beschriebene URI-Schema:

```
/tables[/{tablename}[/fields[/{fieldname}]]]
```

Innerhalb des Listings wurden alle mit `description` gekennzeichneten textuellen Beschreibungen gekürzt. Die gesamte RAML-Dokumentation ist auf der CD im Anhang zu finden.

3.4.7. Anmerkungen zur REST-Konformität

Als nicht vermeidbarer Kritikpunkt kann an dieser Stelle die Beschränkung der Verlinkungen auf eine Teilmenge aller möglichen Stati der *QueryResultResource* genannt werden. In Kapitel 2.1.6 ist Fieldings Vorgabe an einen REST-konformen Webservice aufgeführt, dass alle Anwendungszustände von einem Einstiegspunkt aus über Hypermedia-Elemente innerhalb der Repräsentationen erreichbar sind. Die *QueryResultResource* weist durch das in Kapitel 4 konzipierte generische URI-Schema eine enorme Anzahl an möglichen URIs auf. Innerhalb einer Response könnte eine Menge an Link-Elementen, die diese Kombinationen widerspiegeln, nicht abgebildet werden. Somit kann nicht *jeder* mögliche Status der *QueryResultResource* durch Hypermedia-Verweise erreicht werden. Dies widerspricht zwar in einem Punkt der REST-Konformität, stellt jedoch keinerlei Einschränkungen bei der Verwendung des Webservices dar.

Weiterhin hat Fielding im Bezug auf Hypermedia vorgegeben, dass der Client keine externe Beschreibung benötigen darf, um den Webservice nutzen zu können (siehe Kapitel 4). Wenn man die *DocuResource* als externe Beschreibung ansieht, kann man auch hier den Kritikpunkt äußern, dass sich beim erstmaligen Verwenden der API in das URI-Schema eingelesen werden muss. Durch die starke Orientierung des Schemas an der Abfragesprache SQL, ist der Aufwand für die Einarbeitung jedoch sehr überschaubar.

[6] YAML Ain't Markup Language (YAML) ist ein Standard zur Serialisierung von Daten, der für jede Programmiersprache verwendet werden kann.

3.4.8. JAX-RS

Um den Webservice innerhalb der Java SE Anwendung umzusetzen, wird der Java Standard Java API for RESTful Web Services (JAX-RS) verwendet. Dessen wesentlichen Ziele werden in [43] prägnant zusammengefasst: „[...] die Gewährleistung der REST-Konformität, die Einfachheit bei der Entwicklung, das Ausnutzen moderner Java-Sprachmittel (insbesondere Annotations), die Unterstützung unterschiedlicher Ablaufumgebungen (Java SE und EE) und vor allem die Möglichkeit, auf alle Konstrukte von HTTP zugreifen zu können.“ [43, S. 282 f.]

Die Hauptaufgabe von JAX-RS kann somit als Vermittler zwischen HTTP-Requests, die einzelne Ressourcen adressieren und den Methoden von Ressourcenklassen angesehen werden. Als Ressourcenklassen bezeichnet man in JAX-RS Java Klassen, die über die HTTP-Schnittstelle angesprochen werden können, wobei jede Klasse samt ihrer Annotationen eine im REST Sprachgebrauch verwendeten Ressource entspricht [vgl. 43, S. 283].

Das Framework Jersey 2.0 ist eine Referenzimplementierung von JAX-RS. Es stellt eine API zur Verfügung, die die JAX-RS mit zusätzlichen Features und Erweiterungsmöglichkeiten versieht, um ein einfach zu implementierendes und leichtgewichtiges Werkzeug zur Entwicklung von REST-konformen Webservices in Java anzubieten. Die Einfachheit wird dabei auf die Verwendung von Annotationen gestützt, auf welche in Kapitel 5 näher eingegangen wird. [Vgl. 33] Restlet, RESTEasy, Apache Wink und Apache CXF sind weitere, weniger verbreitetere Implementierungen von JAX-RS, die in dieser Arbeit nicht betrachtet werden.

Um den mit Jersey implementierten Webservice auszuführen, kann einerseits ein Servlet-Endpunkt definiert werden, um die Anwendung in einem Servlet-Container wie Apache Tomcat auszuführen und andererseits ein integrierter Java HTTP-Server genutzt werden. Für das innerhalb dieser Arbeit formulierte Projekt wird auf Grund des überschaubaren Anwendungsumfangs von der Verwendung eines Servlets abgesehen, da ein Java basierter HTTP-Server, wie besispielsweise der der Grizzly HTTP Server [28], einen deutlich minimalistischeren und flexibleren Weg darstellt. In der Jersey-Dokumentation wird die Verwendung eines Grizzly Servers als eine der leichtgewichtigsten und einfachsten Wege beschrieben, um eine funktionelle REST-konforme Webanwendung auszuführen. [Vgl. 17]

4. URI-Schema à la SQL

Im Kapitel 3.4.2 wurden alle entwickelten Ressourcen in zwei Gruppen eingeteilt. Die erste Gruppe beinhaltet alle Ressourcen, die Informationen zum Datenbankschema bereitstellen. In diesem Abschnitt wird das URI-Schema entwickelt, welches die Ressourcen der zweiten Gruppe (*QueryResultResource* und *QueryResultSchemaResource*) identifiziert.

4.1. Möglichkeiten der Übermittlung einer SQL-Abfrage

Die Repräsentation der *QueryResultResource* beinhaltet das Ergebnis einer SQL konformen SELECT Abfrage. Die Abfrage, auf der es beruht, muss dem Webservice mitgeteilt werden. Die innere Anwendungslogik übermittelt die Anfrage der angebundenen Datenbank, welche diese ausführt und Daten in Tabellenform als Ergebnis dieser Anfrage zurückliefert. Die Daten werden in das durch Content Negotiation ausgehandelte Repräsentationsformat gebracht und dem Client als Response auf sein Request mitgeteilt. Es stellt sich nun die Frage, wie die SQL-Anfrage der Serverinstanz am sinnvollsten übermittelt werden kann. Nachfolgend werden drei Möglichkeiten der Übermittlung beschrieben, welche im Anschluss diskutiert werden.

1. Als erste und trivialste Variante, kann die Übermittlung der SQL-Anfrage im Message Body eines GET-Requests genannt werden. Hierbei würde die Ressource mit nur einer einzige URI benannt werden, beispielsweise `/sqlquery`. Die Anfrage würde von dem Client in bekannter SQL-Syntax formuliert werden.

2. Anknüpfend an die erste Variante, könnte man die Anfrage nach festgelegten Richtlinien in einen Query-String [s. 4, S. 15] überführen. Ein Query-String besteht aus einen oder mehreren Query-Parametern, welche wiederum als Schlüssel-Wert-Paar definiert sind. Die Ressource würde zunächst auch mit einer alleinigen URI benannt werden. Überführt man beispielsweise die Anfrage `SELECT * FROM TABELLE1` in einen einzigen Query-Parameter, könnte die URI beispielsweise: `/sqlquery?query=SELECT_*_FROM_TABELLE1` lauten. Betrachtet man die Syntax der SELECT Anweisung aus Kapitel 2.2.4, folgt auf jedes SQL-Kommando ein Parameter. Überführt man diese Gegebenheit in Query-Parameter, könnte man als Key das SQL-Kommando und als Value den bzw. die Parameter festlegen. `/sqlquery?select=*&from=TABELLE1` wäre somit ein Beispiel für die Verwendung mehrerer Query-Parameter im Query-String.

3. Einen anderen Ansatz verfolgt die dritte Variante. Betrachten man den Aufbau einer URI (siehe Kapitel 2.1.4) innerhalb einer REST Umgebung, besteht deren Pfad aus statischen und dynamischen Pfadvariablen, aus denen Hierarchien aufgezeigt werden können [s. 35, S. 134]. Vergleicht man diese Tatsache analog zu Variante 2 mit der Syntax der SELECT Anweisung, erkennt man die Analogie, dass SQL-Kommandos statisch und die zugehörigen Parameter dynamisch sind. Führt man beides zusammen, entsteht ein URI-Schema dessen statische Pfadvariablen den SQL-Kommandos und dynamische Pfadvariablen den SQL-Parametern entsprechen. Die Beispielanfrage aus Variante 2 würde mit diesem Verfahren wie folgt lauten: `/select/*/from/TABELLE1`. Ein Query-String und Message Body werden bei diesem Ansatz nicht benötigt.

Zu den Vorteilen der ersten Variante zählen die Verwendung der allgemeingültigen SQL-Syntax, die Nutzung eines einzigen Pfades innerhalb der URI und die daraus folgende Leichtgewichtigkeit der Implementierung, Umsetzung und Dokumentation der API. Ein Nachteil dieser Variante ist die Verwendung und damit verbundene Erzeugung des Message Body. Bei der

Konzeption der REST-Architektur (siehe Kapitel 3.4.3) wurde festgelegt, dass HTML als Repräsentation der Ressourcen verwendet werden soll, damit der Nutzer über den Browser durch die Anwendung navigieren kann. Dies könnte mit Variante 1 nicht realisiert werden, da der Nutzer ausschließlich die URI in die Adressleiste des Browsers eingeben, jedoch ohne zusätzliche Browser-Plugins kein Message Body des GET-Requests festgelegt werden kann. Als weiterer Nachteil kann die fehlende Beschreibung der Ressource durch die URI angesehen werden. Richardson beschreibt in [35, S. 93 f.], dass URIs eine Struktur aufweisen, vorhersagbar sein und intuitiv mit ihren Ressourcen zusammenpassen sollten. Die alleinige URI `/sqlquery` weist weder eine Struktur auf, noch passt sie Intuitiv mit der zu tätigenden Abfrage zusammen. Der Nutzer erkennt ausschließlich, dass darüber Daten abgefragt werden, jedoch nicht nach welcher SQL-Anweisung. Obgleich man nicht zu viel Wert auf das Design von URIs legen [vgl. 43, S. 47] sollte, „sind elegante URI-Schemata oft Zeichen einer sinnvollen Ressourcenstruktur“ [43, S. 47]. Diese Variante kommt somit für die Übermittlung nicht in Frage.

Die zweite Variante löst das Problem des Message Body, da dieser nicht verwendet werden muss. Die SELECT-Anweisung kann über den Browser ohne Hilfsmittel definiert werden, da der Query-String Bestandteil der URI ist und somit in die Adressleiste eingegeben werden kann. Auch die intuitive Zusammensetzung und Erweiterbarkeit des Query-String ist von Vorteil anzusehen, da die oben beschriebene Analogie zwischen SQL-Kommandos und Query-Parametern besteht. Ein weiterer Vorteil ist die Möglichkeit der Verwendung eines HTML-Formulars, um die URI zu generieren [vgl. 43, S. 50]. Neben diesen Vorteilen bringt diese Variante auch einige Nachteile mit sich. Eine mit Query-String versehene URI wird von einigen Proxys und Cache-Implementierungen ignoriert [s. 35, S. 139]. Richardson und Ruby bezeichnen deshalb die Verwendung von Query-Parametern als „Art von URIs [...], die von schlecht designten Webanwendungen genutzt wird, die HTTP-GET missbrauchen“ [35, S. 139]. Im RFC3986 [4] wird der Query-String definiert als „Komponente, die **nicht hierarchische** Daten enthält, die zusammen mit den Daten der Pfad-Komponenten dazu dient, die Ressource innerhalb des Bereiches des URI-Schemas zu identifizieren“ [4, S. 23]. Es sollen laut RFC demnach nur nicht hierarchische Daten mittels Query-String angegeben werden. Die Definition hierarchischer Daten, wird jedoch im RFC nicht angegeben. Nach [42] ist eine Hierarchie ein System von Elementen, die einander über- bzw. untergeordnet sind. Nach Auffassung des Autors dieser Arbeit hat die SQL-Syntax der SELECT-Anweisung einen deutlichen hierarchischen Charakter. Jedes SQL-Kommando ist mit seinem Parameter dem Voranstehenden syntaktisch untergeordnet[7]. Ein untergeordnetes Kommando bezieht sich auf alle Kommandos, die diesem vorangestellt sind. Betrachtet man beispielsweise die in SQL formulierte Abfrage `SELECT * FROM TABELLE1 WHERE id=5`, folgt die `WHERE`-Klausel auf die voranstehende Anweisung `SELECT * FROM TABELLE1`. Der Parameter des `WHERE` Kommandos ist diesem untergeordnet und folgt somit ausschließlich unmittelbar darauf. Aus diesen Gründen wird sich auch gegen diese Variante entschieden.

Bei Variante drei kann mittels HTML-Formular ohne die Verwendung einer Skriptsprache keine URI in diesem Schema erzeugt werden. Jedoch birgt diese Variante einige Vorteile. Zum einen wird kein Query-String und Message Body verwendet, was die damit verbundenen Nachteile (siehe Oben) vermeidet. Die intuitive erweiterbare Struktur, kann durch die Analogie zur SQL-Syntax auch als vorhersagbar bezeichnet werden. Die Hierarchie der SQL-Kommandos wird in die der Pfadparameter der URI überführt. Weiterhin wird der Inhalt der Ressourcenrepräsentation, dementsprechend das Ergebnis der Datenbankabfrage, mit diesem Vorgehen eindeutig über die

[7] Die logische Verarbeitungsreihenfolge der SELECT-Anweisung ist von der Reihenfolge der Kommandos in der SELECT-Anweisung verschieden [s. 37]. Der Bezug zum hierarchischen Charakter ist demnach nur auf die syntaktischen Reihenfolge anwendbar.

URI beschrieben. Eine Erweiterbarkeit ist analog zur Variante Zwei auch gegeben. An dieser Stelle ist auch folgende Aussage von Richardson und Ruby anzuführen:

> *URIs sollten wann immer möglich unsere bestehenden Konventionen verwenden* [35, S. 135].

Zu den *bestehenden Konventionen* zählt in diesem Fall die SQL SELECT-Syntax. Eine einfache SELECT-Anweisung kann nach dieser Variante theoretisch ohne weitere Dokumentation in die entsprechende URI übersetzt werden. Komplexere Anweisungen benötigen jedoch eine Dokumentation, nicht zuletzt wegen der Festlegung gültiger Zeichen innerhalb eines Pfadparameters. Alles in allem kann Variante Drei als geeignetstes Verfahren zur Übermittlung der SQL-Abfrage angesehen werden und wird somit in den nachfolgenden Kapiteln als Konzept berücksichtigt.

4.2. Erlaubte Zeichen im Pfad-Segment der URI

Bevor auf die Entwicklung des URI-Schemas eingegangen werden kann, muss geklärt werden, welche Zeichen im Pfad-Segment (auch Pfad-Komponente) verwendet werden dürfen und wie nicht verwendbare Zeichen innerhalb des Segmentes codiert werden, damit sie nicht in Konflikt mit den Regelungen in den entsprechenden RFC geraten. RFC3986 [4] ist ausschlaggebend für diese Betrachtung. Darin ist die generische URI Syntax und einen Prozess zur Auflösung von URI Referenzen definiert. Dier innerhalb einer URI erlaubten Zeichen sind auf den US-ASCII Zeichensatz eingeschränkt. Betrachtet man die Syntax der SELECT-Anweisung (Kap. 2.2.4), lassen sich folgende darin verwendete Sonderzeichen erfassen: `. , + - _ * / < > = %( )` und das Leerzeichen ␣. Im RFC3986 sind folgende Zeichen innerhalb eines Pfad-Segments erlaubt [vgl. 4, S. 12 f.,21 f.,48 f.] und können somit verwendet werden: `. , ; + - * = ( ) _ ~ ! $ & '`. Entfernt man alle erlaubten Zeichen aus der Menge der in den SQL-Anweisungen verwendeten Zeichen, erhält man die Menge an Zeichen, die durch ein erlaubtes Trennzeichen oder eine Zeichenfolge codiert werden muss. Die daraus entstehenden Festlegungen sind für das zu entwickelnde generische URI-Schema allgemeingültig und müssen bei dem Entwicklungsprozess im nächsten Kapitel berücksichtigt werden. Die Tabelle 6 beinhaltet diese unerlaubten Sonderzeichen und das codierte Trennzeichen bzw. die Zeichenfolge, die an Stelle des Zeichens verwendet werden muss.

Unabhängig davon besteht die Möglichkeit den „percent-encoding“-Mechanismus zu nutzen, um alle Zeichen die nicht verwendet werden dürfen, als Oktett zu codieren. Ein mit diesem Mechanismus codiertes Oktett besteht aus dem %-Symbol[8], gefolgt von einem oder zwei hexadezimalen Zeichen, die dessen nummerischen Wert repräsentieren. Beispielsweise ist „%20“ das Leerzeichen, codiert mit dem „percent-encoding“-Mechanismus. [Vgl. 4, S. 11]

Das Verwenden dieser Codierungen führt zu einer deutlichen Verschlechterung der Lesbarkeit des URIs. Daher wird der „percent-encoding“-Mechanismus im Umfang dieser Arbeit nicht verwendet.

4.3. Entwicklung des Schemas

Die Entwicklung des generischen URI-Schemas zur Durchführung von Datenbankabfragen, aus dem ein URI zur Identifikation der *QueryResultResource* gebildet werden kann, wird auf Grundlage der Entscheidung für die dritte Variante der Übermittlung aus dem vorherigen Kapitel

[8] Da das Prozent-Symbol für den „percent-encoding“-Mechanismus benötigt wird, ist es nicht als frei verwendbares Zeichen innerhalb einer URI definiert.

Tabelle 6: Nicht erlaubte Sonderzeichen und deren Codierung

Zeichen	Codierung	Bemerkung
␣	∼	Innerhalb von SQL wird das Leerzeichen zur Trennung von einzelnen Argumenten genutzt. Operatoren wie `AND`, `OR`, `LIKE` und andere aus alphanumerischen Zeichen bestehende, müssen innerhalb der URI von zwei Tilde-Symbolen eingeschlossen werden.
/	div	Wird innerhalb von SQL zur Division verwendet, daher Abkürzung 'div'. Es ist zwingend erforderlich, dass diese Codierung von zwei Tilde-Symbolen zur Abgrenzung von umliegenden alphanumerischen Zeichen umgeben wird (∼`div`∼).
<	lt	Kleiner-als-Zeichen (engl. less than), daher Abkürzung 'lt'. Es ist zwingend erforderlich, dass diese Codierung von zwei Tilde-Symbolen zur Abgrenzung von umliegenden alphanumerischen Zeichen umgeben wird (∼`lt`∼).
<=	leq	Kleiner-gleich-Zeichen (engl. less or equal), daher Abkürzung 'leq'. Es ist zwingend erforderlich, dass diese Codierung von zwei Tilde-Symbolen zur Abgrenzung von umliegenden alphanumerischen Zeichen umgeben wird (∼`leq`∼).
>	gt	Größer-als-Zeichen (engl. greater than), daher Abkürzung 'gt'. Es ist zwingend erforderlich, dass diese Codierung von zwei Tilde-Symbolen zur Abgrenzung von umliegenden alphanumerischen Zeichen umgeben wird (∼`gt`∼).
>=	geq	Größer-gleich-Zeichen (engl. greater or equal), daher Abkürzung 'geq'. Es ist zwingend erforderlich, dass diese Codierung von zwei Tilde-Symbolen zur Abgrenzung von umliegenden alphanumerischen Zeichen umgeben wird (∼`geq`∼).
<>	neq	Ungleich-Zeichen (engl. not equal), daher Abkürzung 'neq'. Es ist zwingend erforderlich, dass diese Codierung von zwei Tilde-Symbolen zur Abgrenzung von umliegenden alphanumerischen Zeichen umgeben wird (∼`neq`∼).
%	prc	Wird innerhalb von SQL als Platzhalterzeichen verwendet. Die Abkürzung 'prc' ist angelehnt „percent“, der englischen Übersetzung von „Prozent“. Es ist zwingend erforderlich, dass diese Codierung von zwei Tilde-Symbolen zur Abgrenzung von umliegenden alphanumerischen Zeichen umgeben wird (∼`prc`∼).

durchgeführt. Bei diesem Verfahren muss ausschließlich die Pfad-Komponente der URI betrachtet werden. Für die nachfolgenden Beschreibungen und Darstellungen werden aus Übersichtlichkeitsgründen ausschließlich zum Basis-URI relative Pfadangaben genutzt. Damit entfallen die redundanten Angaben der Pfad-Komponente voranstehenden Bestandteile der URI.

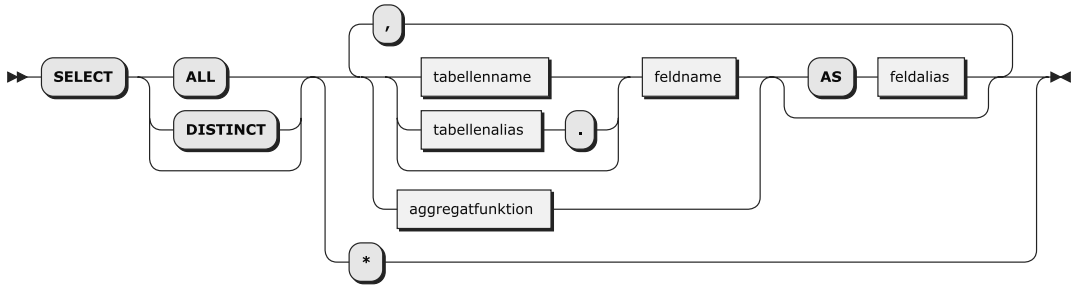
Wie in Kapitel 2.2.4 angegeben, empfiehlt es sich die Syntaxelemente der SQL SELECT-Anweisung und deren Argumente in Klauseln zusammenzufassen. Diese Menge an Klauseln kann nun herangezogen werden, um jede einzelne in das URI-Schema zu überführen. Zu beachten ist, dass nur einfache Anweisungen berücksichtigt werden sollen (siehe Kapitel 2.2.4).

Die nachfolgenden Tabellen (Tabelle 7 bis 15) erläutern die Überführung der Klauseln in das URI-Schema. Jede Tabelle ist in drei Komponenten unterteilt: ein Railroad-Digramm der SQL-Syntax, gefolgt von einer textuellen Beschreibung der Überführung in das URI-Schema und ein Railroad-Diagramm, welches die Syntax des entstandenen URI-Schemas der betrachteten Klausel beschreibt.

Tabelle 7: Überführung der Select-Klausel in das generische URI-Schema

1. Select

select-clause:



Die Select-Klausel besteht aus dem SQL-Kommando `SELECT`, dem optionalen Kommandos `DISTINCT` und `ALL` und der zugehörigen SELECT-Anweisung. Als relativer Pfad wird demnach `/select/[all/|distinct/]{selectExpression}` gewählt. Die enthaltenen Nichtterminalsymbole *tabellenname*, *feldname*, etc. sind einfache Zusammensetzungen alphanumerischer Zeichen, die ohne weitere Betrachtungen übernommen werden können. Die Terminalsymbole Stern (*), Komma (,) und Punkt (.) zählen zu den im RFC3986 festgelegten Trennzeichen, die innerhalb eines Pfad-Segments verwendet werden dürfen (siehe Kapitel 4.2) und können somit auch im Pfadparameter verwendet werden. Das Terminalsymbol `AS`, welches den Feldbezeichner vom Alias trennt, muss von einem Leerzeichen (␣) umgeben sein, da es selbst aus zwei alphanumerischen Zeichen besteht und sonst nicht von der Spaltenbezeichnung unterschieden werden könnte. Wie im vorherigen Kapitel festgelegt, ersetzt das Tilde-Symbol diese Leerzeichen. Das Beispiel `SELECT tab1.col1 AS a,tab2.col3 AS b,SUM(*)` wird somit in folgende relative Pfadangabe überführt: `select/tab1.col1~AS~a,tab2.col3~AS~b,SUM(*)`.

select-clause:

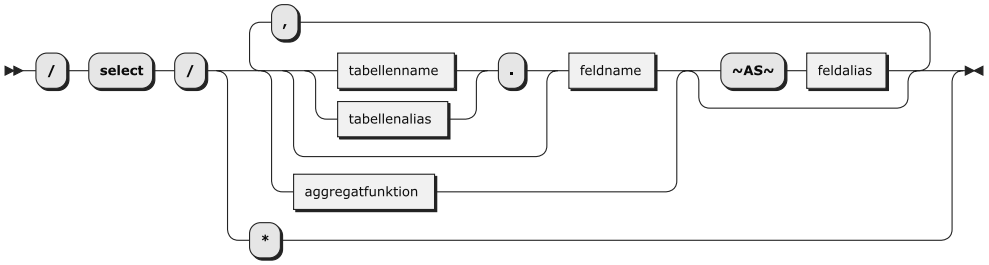
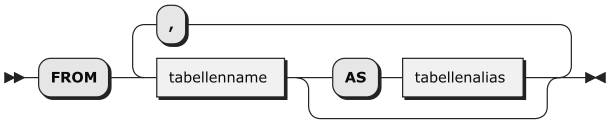


Tabelle 8: Überführung der From-Klausel in das generische URI-Schema

2. From

from-clause:



Der relative Pfad der From-Klausel lautet: `/from/{tableExpression}`, da diese aus dem SQL-Kommando `FROM` und der Tabellen-Anweisung besteht. Analog zur Select-Klausel müssen nur die Leerzeichen, die das Terminalsymbol `AS` umgeben, durch das Tilde-Symbol ersetzt werden. Die Nichtterminalsymbole *tabellenname* und *tabellenalias* sind abermals einfache Zusammensetzungen aus alphanumerischen Zeichen und können übernommen werden. Das obige Beispiel `FROM tabelle1 AS tab1, tabelle2 AS tab2` wird nach Beachtung dieser Regeln überführt in `/from/tabelle1~AS~tab1,tabelle2~AS~tab2`.

from-clause:

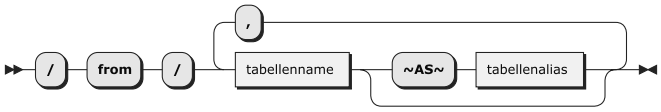
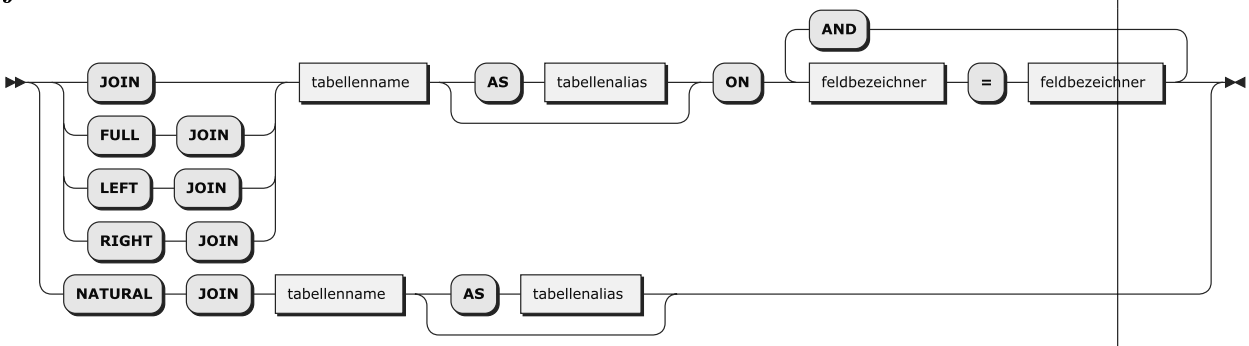


Tabelle 9: Überführung der Join-Klausel in das generische URI-Schema

3. Join

join-clause:



`/{joinType}/{joinTable}[/on/{tableRelation}]` wird für die Join-Klausel als relativer Pfad gewählt, wobei der Pfad-Abschnitt `/on/{tableRelation}` optional ist, da bei einem *NATURAL JOIN* keine Tabellenbeziehung angegeben wird. Wichtig ist, dass `{joinType}` kein klassischer Pfadparameter ist, sondern ein Platzhalter für eine festgelegte Menge an Pfad-Komponenten, die die verschiedenen Typen der Joins widerspiegeln. Diese Menge besteht aus folgenden fünf Pfad-Elementen: `join`, `fullJoin`, `leftJoin`, `rightJoin` und `naturalJoin`. Da für die zu vereinigende Tabelle ein Alias mit dem Kommando `AS` definiert werden kann, muss dieses, aus dem selben Grund wie bei den letzten beiden Klauseln, mit dem Tilde-Symbol umgeben werden. Wie aus dem Railroad-Diagramm in Abbildung 1 zu erkennen ist, kann die Join-Klausel wiederholt angegeben werden. Die Überführung der beispielhaften Join-Klausel `LEFT JOIN tabelle3 AS tab3 ON tab3.col1 = a` in das URI-Schema lautet: `/leftJoin/tabelle3~AS~tab3/on/tab3.col1=a`

join-clause:

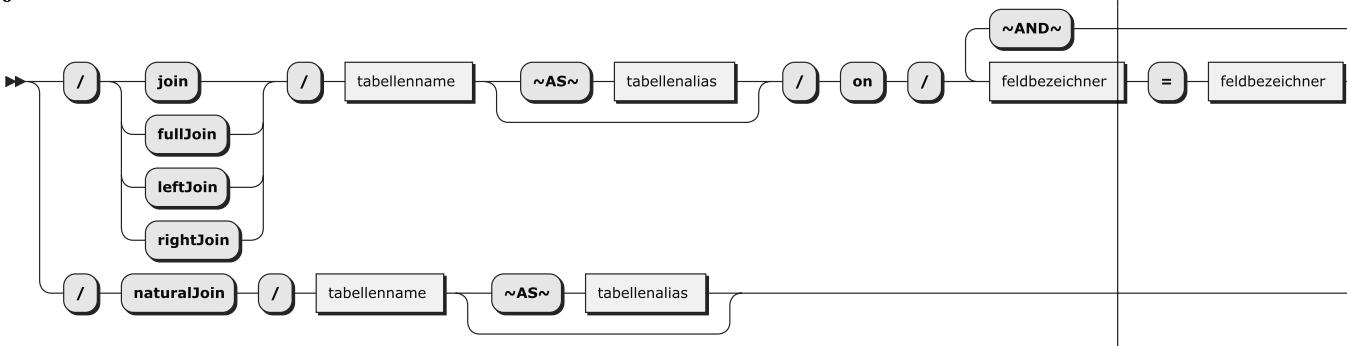
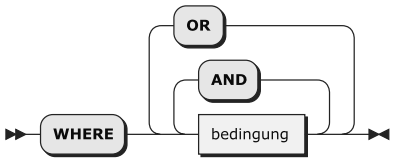


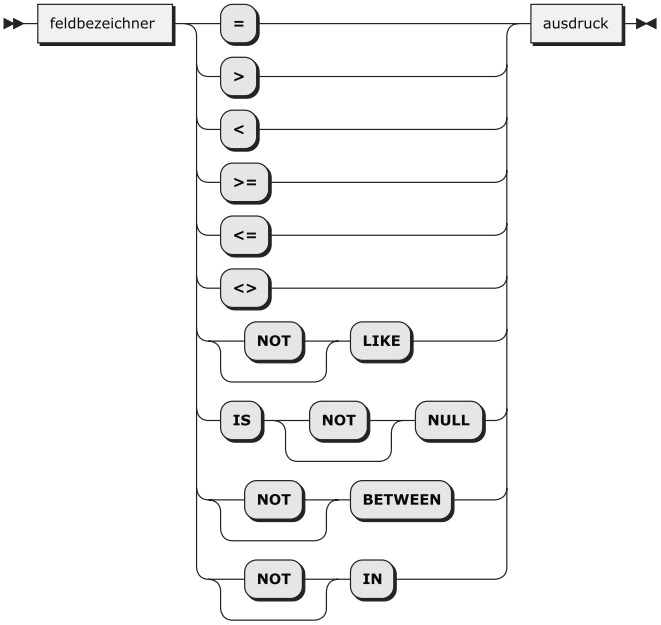
Tabelle 10: Überführung der Where-Klausel in das generische URI-Schema

4. Where

where-clause:

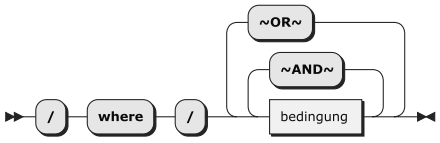


bedingung:



Die Where-Klausel besteht aus dem SQL-Kommando `WHERE` und logisch verknüpften Bedingungen als zugehöriger Parameter. Der relative Pfad lautet somit `/where/{whereCondition}`. Eine Bedingung innerhalb der `{whereCondition}` besteht aus dem Feldbezeichner, einem Operator und einem Ausdruck. Ein Vergleichsoperator wird nach Tabelle 6 innerhalb des Pfad-Parameters codiert. Werden mehrere Bedingungen angegeben, können diese logisch mittels `AND` oder `OR` verknüpft werden. Wie gehabt müssen diese Kommandos mit dem Tilde-Symbol umgeben sein, um im Pfad Verwendung zu finden. Das Beispiel `WHERE tab2.col3 > 100 AND tab1.col1 LIKE 'Haus%'` aus Tabelle `??` wird überführt in `/where/tab2.col3~gt~100~AND~tab1.col1~LIKE~'Haus~prc~'`.

where-clause:



bedingung:

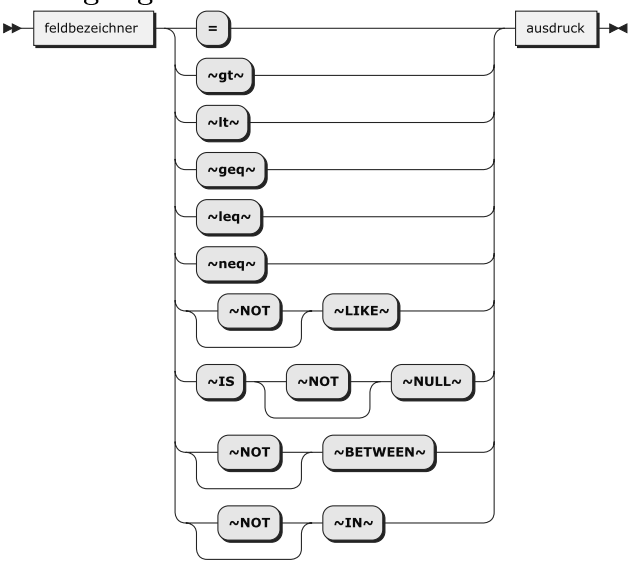
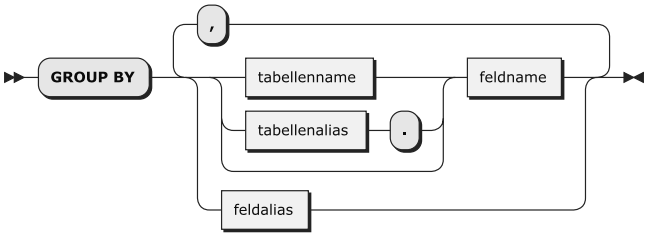


Tabelle 11: Überführung der Group By-Klausel in das generische URI-Schema

5. Group By

groupby-clause:



Das SQL-Kommando `GROUP BY` und die zugehörige Group By-Anweisung sind Bestandteil der Group By-Klausel, wodurch als relativer Pfad `/groupBy/{groupByExpression}` festgelegt wird. Sowohl die Terminalsymbole Komma (,) und Punkt (.), als auch die drei Nichtterminalsymbole können ohne spezielle Codierung übernommen werden. Das Beispiel `GROUP BY a,b` ergibt somit den Pfad `/groupBy/a,b` .

groupby-clause:

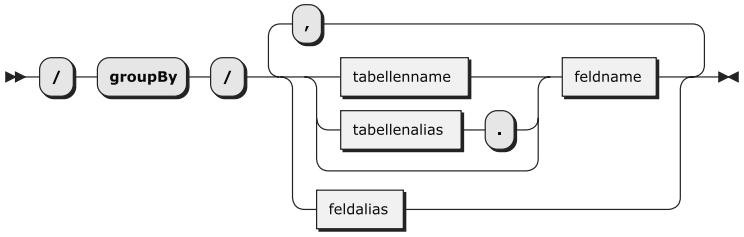


Tabelle 12: Überführung der Having-Klausel in das generische URI-Schema

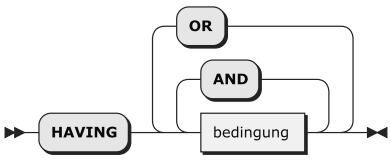
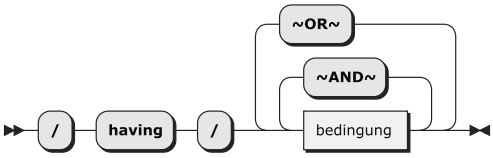
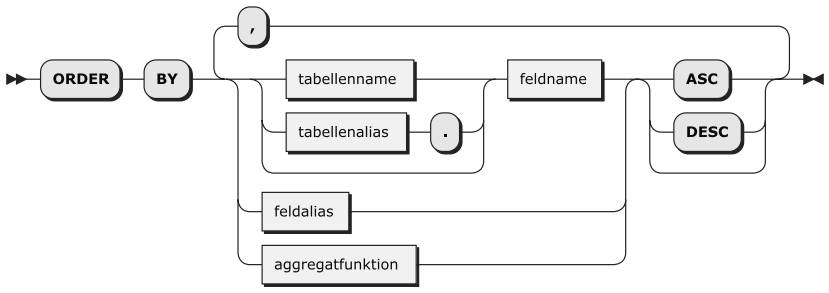
6. Having
having-clause:    **bedingung:** siehe Tabelle 10 (oben)
Die Überführung der Having-Klausel erfolgt analog zur Where-Klausel (Tabelle 10) und muss somit nicht näher erläutert werden. `/having/{havingCondition}` ist als reletiver Pfad festgelegt. Aus dem Beispiel `HAVING SUM(*) > 455` ergibt sich der relative Pfad `/having/SUM(*)~gt~455` .
having-clause:    **bedingung:** siehe Tabelle 10 (unten)

Tabelle 13: Überführung der Order By-Klausel in das generische URI-Schema

7. Order By

orderby-clause:



Die Order By-Klausel wird analog zur Group-By-Klausel (Tabelle 11) in das URI-Schema überführt, mit dem Unterschied, dass an dieser Stelle als Spaltenbezeichnung auch eine in der Select-Anweisung gewählte Aggregatfunktion verwendet werden kann. Der relative Pfad ist somit: `/orderBy/{orderByExpression}`. Um die Sortierreihenfolge festzulegen, wird das Kommando `ASC` (ascending, z.Dt. aufsteigend) oder `DESC` (descending, z.Dt. absteigend) nach der Spaltenbezeichnung angefügt. Bei der Überführung in das URI-Schema ist abermals die Angabe des Tilde-Symbols vor und nach dem Kommando zur Trennung der alphanumerischen Zeichen notwendig. Die beispielhafte Order By-Klausel `ORDER BY SUM(*) DESC` ergibt nach erfolgter Überführung den Pfad: `/orderBy/SUM(*)~DESC~` .

orderby-clause:

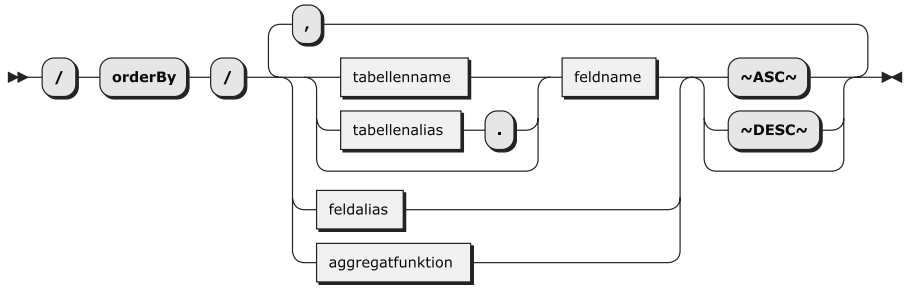


Tabelle 14: Überführung der Limit-Klausel in das generische URI-Schema

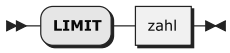
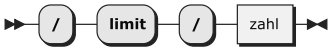
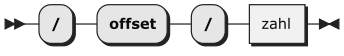
8. Limit
limit-clause: 
Die Limit-Klausel besteht aus dem SQL-Kommando `LIMIT` und einer positiven Ganzzahl als dessen Argument. Als relativer Pfad wurde daher `/limit/{limitExpression}` festgelegt. Nach Überführung des Beispiels `LIMIT 100` entsteht der relative Pfad `/limit/100` .
limit-clause: 

Tabelle 15: Überführung der Offset-Klausel in das generische URI-Schema

9. Offset
offset-clause:
Analog zur Limit-Klausel (Tabelle 14) erfolgt auch die Überführung der Offset-Klausel. Der entstandene relative Pfad ist somit `/offset/{offsetExpression}`. Das Beispiel `OFFSET 555` aus Tabelle **??** lautet als Pfad formuliert `/offset/555` .
offset-clause: 

Mit den Erläuterungen aller Klauseln ist das URI-Schema beschrieben, aus dem alle URIs, die die *QueryResultResource* identifizieren, gebildet werden können. Durch den direkten Vergleich der Railroad-Diagramme für jede Klausel ist die Analogie zwischen SQL-Syntax und URI-Schema deutlich erkennbar. Das nachfolgende Beispiel (Tabelle 16) illustriert die Überführung einer Select-Abfrage[9] über das URI-Schema in die gleichbedeutende URI.

Tabelle 16: Beispiel der Überführung einer SQL Select-Abfrage in eine URI

SQL	`SELECT max(l.spiele_rekordspieler) AS Rekordspielerspiele,`
	`count(s.spieler_id) AS Spielernummeranzahl,avg(s.trikot_nr)`
	`AS Trikotnummerschnitt,s.land AS Spielerland,sum(s.Spiele) AS`
	`Spielsumme,sum(s.Tore) AS Torsumme`
	`FROM SPIELER AS s`
	`JOIN verein AS v ON s.vereins_id=v.V_ID`
	`JOIN liga AS l ON v.liga=l.LIGA_NR`
	`WHERE (s.land LIKE 'C%' OR s.land LIKE 'D%' OR s.land LIKE 'E%') AND`
	`s.spiele>0 AND s.tore>0`
	`GROUP BY Spielerland`
	`HAVING Torsumme>5 AND Spielsumme>10`
	`ORDER BY Torsumme DESC ,Spielsumme ASC`
	`LIMIT 2`
	`OFFSET 3`

URI	`http://sqloverhttp.de`
	`/select/max(l.spiele_rekordspieler)~as~Rekordspielerspiele,`
	`count(s.spieler_id)~as~Spielernummeranzahl,`
	`avg(s.trikot_nr)~as~Trikotnummerschnitt,s.land~as~Spielerland,`
	`sum(s.Spiele)~as~Spielsumme,sum(s.Tore)~as~Torsumme`
	`/from/SPIELER~as~s`
	`/join/verein~as~v/on/s.vereins_id=v.V_ID`
	`/join/liga~as~l/on/v.liga=l.LIGA_NR`
	`/where/(s.land~like~'C~prc~'~or~s.land~like~'D~prc~'~or~`
	`s.land~like~'E~prc~')~and~s.spiele~gt~0~and~s.tore~gt~0`
	`/groupBy/Spielerland`
	`/having/Torsumme~gt~5~and~Spielsumme~gt~10`
	`/orderBy/Torsumme~desc~,Spielsumme~asc~`
	`/limit/2`
	`/offset/3`

Um abschließend die Ressource *QueryResultSchemaResource* (siehe Kapitel 3.4.1) zu benennen, wird der zu allen gültigen URIs der *QueryResultResource* relative Pfad `/schema` definiert. Der Vollständigkeit halber ist das entsprechende Railroad-Diagramm in Abbildung 6 dargestellt.

Abbildung 6: Railroad-Diagramm des Identifikators der *QueryResultSchemaResource*

[9] Die Abfrage bezieht sich auf ein von der Universität Bayreuth veröffentlichtes Relationenschema, einsehbar unter [20].

Weiterhin wurde in Kapitel 3.4.3 ein QueryParameter festgelegt, der an alle die *QueryResultResource* identifizierende URIs angefügt werden kann, um das Ergebnis der Abfrage, innerhalb der Repräsentation im JSON Format, Zeilen- bzw. Spalten-orientiert zurückzugeben. Ach hierfür ist zur Vollständigkeit das Railroad-Diagramm der Abbildung 7 zu entnehmen.

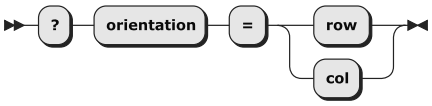


Abbildung 7: Railroad-Diagramm des Query-Parameters „orientation“

Die Abbildung 8 zeigt als Ergebnis das gesamte URI-Schema als Railroad-Diagramm. Im Anhang (A.6) befindet sich die zugehörige Syntaxbeschreibung in der EBNF.

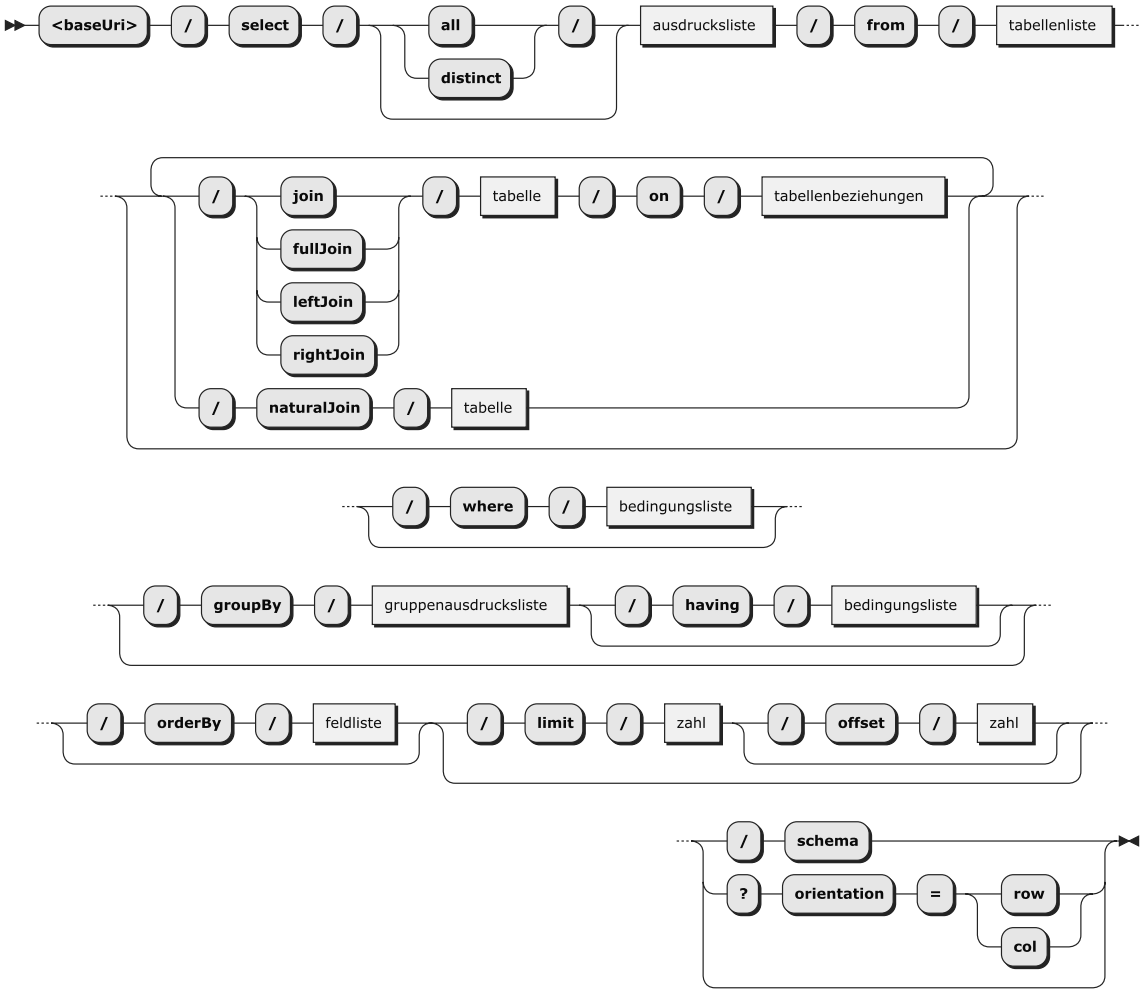


Abbildung 8: URI-Schema zum formulieren der SQL SELECT Anweisung als Railroad Diagramm

4.4. Das URI-Schema und dessen generischer Charakter

In der Themenbezeichnung dieser Bachelor Thesis und in anderen Kapiteln wurde das hier entwickelte URI-Schema als *generisch* bezeichnet. Was ist jedoch ein URI-Schema und weshalb wird es als „generisch“ bezeichnet? Dies wird in diesem Kapitel geklärt.

Die Bezeichnung „Schema“ taucht missverständlicher weise an zwei Stellen im Bezug auf einen URI auf. Wie schon in Kapitel 2.1.4 erläutert, ist der erste Bestandteil vor dem Doppelpunkt (:) der URI-Syntax das sogenannte „Schema“, welches auch als „Protokoll“ bezeichnet wird. Es bezieht sich auf einen von der IANA definierten Kontext (und somit Typ), der für den gesamten URI gültig ist [vgl. 4, S. 16]. Bekannte Schemen sind beispielsweise http, ftp, ssh und mailto [s. 21]. Zum anderen gilt der Begriff URI-Schema im Zusammenhang mit HTTP auch als Beschreibung der Struktur einer URI. Genau wie ein XML-Schema den Aufbau einer XML-Datei beschreibt, kennzeichnet das URI-Schema den Aufbau eines gültigen URIs. Im Umfang dieser Arbeit wird der Begriff des URI-Schemas in diesem Kontext verwendet.

Generisch bedeutet „allgemeingültig“ oder „im allgemeingültigem Sinne“ bzw. „die Gattung betreffend“ [s. 46]. Das URI-Schema wird als generisch bezeichnet, da es, aufgrund der Orientierung an der SQL Syntax, für alle SQL konformen Datenbanken verwendet werden kann, besser gesagt: allgemeingültig für alle SQL-konformen Datenbanken ist. Das Schema und der Inhalt der Datenbank ist für das URI-Schema irrelevant.

4.5. RAML Dokumentation des Schemas

In Kapitel 3.4.6 wurde das URI-Schema zum abrufen des Datenbankschemas aus Kapitel 3.4.2 schon in ein für diesen Webservice generiertes RAML-Dokument eingepflegt. Dieses Dokument muss nun mit der Beschreibung des hier entwickelten URI-Schemas erweitert werden. Ein Auszug der Dokumentation ist im Anhang A.2 dargestellt. Das vollständige RAML-Dokument ist auf der CD im Anhang abgelegt.

Da alle URIs dieses Schemas zu gleichen Teilen die *QueryResultResource* und *QueryResultSchemaResource* identifizieren, bietet es sich an, ein eigenen Ressourcentyp für jede Ressource zu definieren. Dies ermöglicht die Vermeidung redundanter Angaben innerhalb der Dokumentation. Als Bezeichnungen der Ressourcentypen wurden `queryResult` und `querySchemaResult` gewählt. Die Definition des Typen `queryResult` ist in dem Ausschnitt der RAML-Dokumentation im Anhang A.2 aufgeführt (Zeile 8-41). Jeder Ressourcentyp enthält eine textuelle Beschreibung (Zeile 10), die Angabe der möglichen HTTP-Methoden (in diesem Fall ausschließlich `GET`), die Definition der verwendbaren Query-Parameter mit deren Beschreibung, Typ und Beispielangabe (hier ausschließlich `orientation`) und möglichen Responses mit Definition der Statuscodes und Repräsentationsformate.

Die Dokumentation des URI-Templates

```
/select/{selectExpression}/from/{tableExpression}
```

kann den Zeilen 43 bis 60 entnommen werden. Da dieser URI die *QueryResultResource* identifiziert, wird als Ressourcentyp `queryResult` angegeben. Die Ressource erbt somit alle Eigenschaften, die innerhalb des Typs definiert wurden. Zusätzlich werden die beiden Pfadparameter `{selectExpression}` und `{tableExpression}` dokumentiert, indem jeweils eine textuelle Beschreibung, der zu verwendende Datentyp, ein Beispielparameter und die „required“-Kennzeichnung angegeben wird.

Das Abbilden sich wiederholender Pfadkomponenten ist leider noch nicht in den RAML-Standard enthalten. Wie schon erwähnt, kann die Join-Klausel mehr als einmal angegeben werden kann. Diese Möglichkeit der Wiederholung weist auch das hier entwickelte URI-Schema auf. Der

Pfad der Join-Klausel kann darin wiederholt angegeben werden, um eine Select-Anweisung mit mehreren Joins abzubilden. Nach Rücksprache des Autors dieser Arbeit mit den Entwicklern des Dokumentationsformates RAML, existiert aktuell keine Möglichkeit, diese Pfadangaben der Join-Klausel im RAMl-Format zu dokumentieren. Deshalb wurde ausschließlich eine textuelle Beschreibung für diese Option in die Dokumentation aufgenommen.

5. Realisierung

Die Anwendung wurde innerhalb der IntelliJ IDE mit dem Build-Management-Tool *Maven* der Apache Software Foundation implementiert. In diesem Kapitel werden nur einige ausgewählte Implementierungsdetails näher betrachtet. Der gesamte Quellcode und das ausführbare Java Archiv sind auf der CD im Anhang zu finden.

5.1. JAX-RS Integration

In diesem Kapitel wird der Integrationsvorgang des Jersey-Frameworks beschrieben. Dessen korrekte Einbindung in die Anwendung ist essentiell, da der RESTful Webservice mit Hilfe diese Frameworks implementiert wird.

```
1  import org.glassfish.grizzly.http.server.HttpServer;
2  import org.glassfish.jersey.grizzly2.httpserver.GrizzlyHttpServerFactory;
3  import org.glassfish.jersey.server.ResourceConfig;
4  ...
5  public class Main {
6      public static HttpServer startServer() {
7          Set<Class<?>> s = new HashSet<>();
8          //Add all resource classes to this set
9          s.add(RootPathHandler.class);
10         final ResourceConfig rc = new ResourceConfig(s);
11         rc.setApplicationName("SQL over HTTP WebService");
12         return GrizzlyHttpServerFactory.createHttpServer(URI.create(BASE_URI),
               rc);
13     }
14
15     public static void main(String[] args) {
16         ...
17         final HttpServer server = startServer();
18         ...
19         server.shutdown();
20     }
21 }
```

Listing 4: Auszug aus Klasse Main - Initialisierung und Start des HTTP-Servers mit Jersey Webservice

Innerhalb der Maven Konfigurationsdatei muss die Softwareabhängigkeit zu den benötigten Paket *org.glassfish.jersey* angegeben werden, damit dieses in das lokale Maven-Repository kopiert wird und für die Anwendung zur Verfügung steht. Darin enthalten ist sowohl die Klassen des Grizzly HTTP Servers, als auch die des Jersey-Frameworks. In der Main Klasse wurde zum konfigurieren und starten des Webservers, sowie zum initialisieren des Webservices, die in Listing 4 dargestellte Methode `public static HttpServer startServer()` implementiert. Darin wird ein Objekt der Klasse `ResourceConfig` erstellt (Zeile 10), dem als Argument ein `HashSet` übergeben wird, dass alle Class-Objekte der Klassen beinhaltet, die einer im REST Sprachgebrauch verwendeten Ressource entsprechen. Im Listing ist dies aktuell nur die Klasse `RootPathHandler` (Zeile 9). Sobald eine neue Klasse mit Jersey-spezifischen Annotationen dem Webservice zur Verfügung stehen soll, muss diese dem `HashSet` hinzugefügt werden. In Zeile 10 wird dem Webservice ein Name zugewiesen. Zuletzt wird über die statische Methode `createHttpServer(...)` der Klasse

`GrizzlyHttpServerFactory` eine Instanz des Grizzly HTTP-Servers erstellt und als Rückgabewert definiert. Der Methode wird die Basis-URI und das Jersey-Konfigurationsobjekt übergeben (Zeile 12). In der `main` Methode wird ein `HttpServer` Objekt erzeugt und über die beschriebene Methode initialisiert (Zeile 17). Der HTTP-Server kann außerdem über die Methode `shutdown()` beendet werden (Zeile 19).

5.2. Generisches URI-Schema durch Annotationen und Sub-Ressourcen

Im vorherigen Kapitel wurde die Klasse `RootPathHandler` der Ressourcenkonfiguration hinzugefügt. Diese Klasse ist eine Root-Resource Klasse. Sie ist gekennzeichnet durch eine `@Path`-Annotation vor der Klassenbezeichnung. Der Wert dieser Annotation gibt den relativen Pfad bezogen auf den Basis URI an, in diesem Fall „/“. Innerhalb der Klasse existieren Ressourcenmethoden. Steht eine `@Path`-Annotation vor der Methodendeklaration, kennzeichnet diese eine Sub-Ressource Klasse. Steht ein Request-Methoden Bezeichner (`@GET, @PUT, @POST` oder `@DELETE`) vor der Methode, ist diese für die Beantwortung eines HTTP-Request zuständig. Eine Sub-Resource Klasse ist eine Ressourcenklasse, die wiederum Ressourcenmethoden beinhaltet [s. 15, S. 27 ff.].

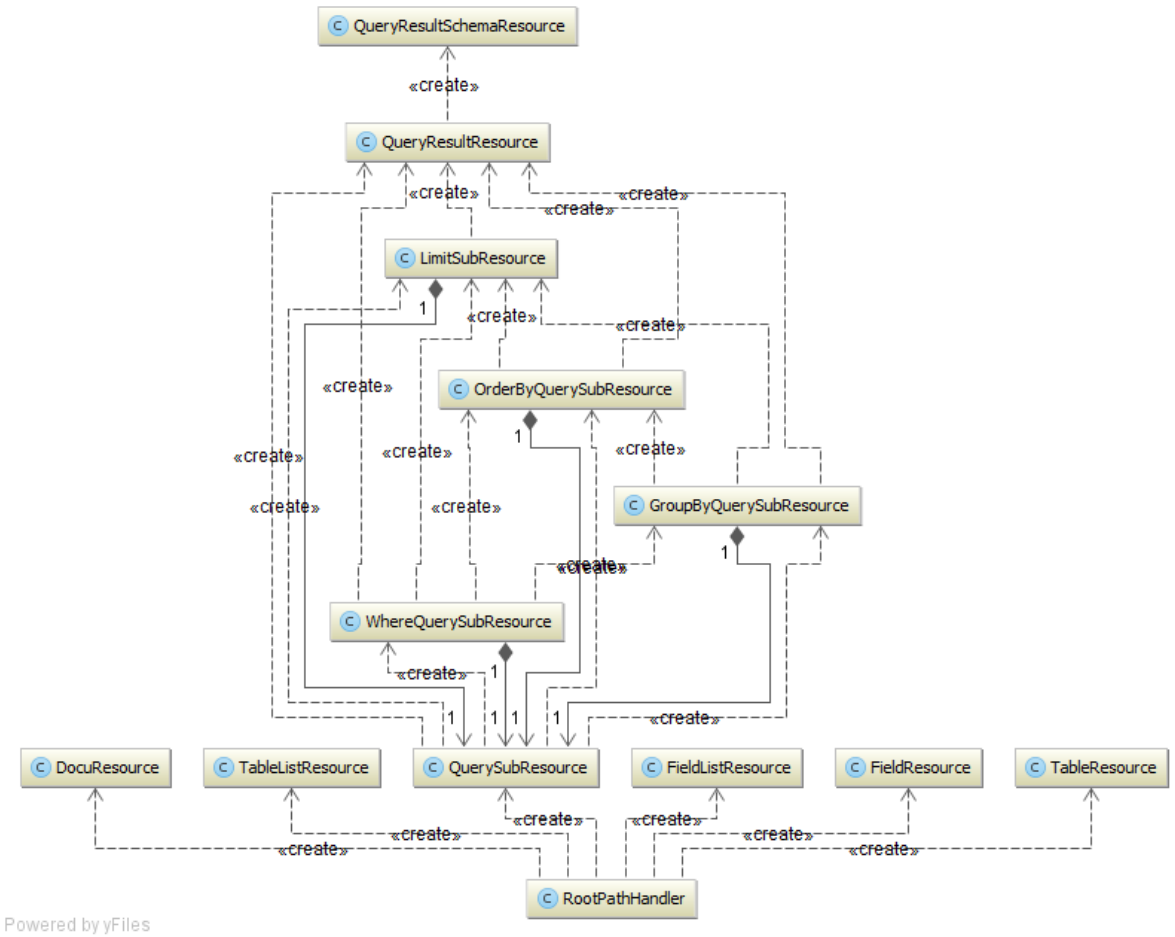


Abbildung 9: Ressourcenklassen des Package `resources` im Klassendiagramm

Das in dieser Arbeit konzipierte generische URI-Schema kann mit Hilfe dieser von JAX-RS

bereitgestellten Annotationen vollständig programmatisch abgebildet werden. Abbildung 9 zeigt das Klassendiagramm[10], welches sich aus der Kombination von Root-Resource Klasse (unteres Wurzelelement), Sub-Resource Klassen und deren Beziehungen untereinander ergeben hat. Eine Besonderheit stellt dabei die Umsetzung der möglichen Wiederholungen der Join-Klausel dar, welche in Kapitel 5.4 näher betrachtet wird. Alle Ressourcenklassen befinden sich in einem separaten Package namens `resources`.

5.3. Abbildung der Ressourcen

Ressourcen sind die zentralen Bausteine eines REST-konformen Webservice, da deren Repräsentationen über den Webservice für den Nutzer bereitgestellt werden (siehe Kapitel 2.1.3). Die in Kapitel 3.4.1 entworfenen Ressourcen müssen nach dem Prinzipen der Objektorientierung auf Klassen abgebildet werden. Im vorherigen Kapitel wurde schon auf diese Ressourcenklassen eingegangen. In diesem Kapitel wird die Klasse `QueryResultResource` beschrieben, welche die gleichnamige REST-Ressource symbolisiert und gleichzeitig als Sub-Resource Klasse fungiert. Grundsätzlich kann die Menge aller Attribute und Methoden der Klasse in zwei Teilmengen unterschiedlicher Funktionsbereiche aufgeteilt werden. Eine Teilmenge beinhaltet alle Attribute der zu repräsentierenden Daten und deren zugehörige Getter-Methoden. Die andere weist alle Eigenschaften und Methoden der internen Verarbeitung eines eintreffenden Requests auf. Diese funktionale Aufteilung trifft für alle Klassen zu, die als Abbildung einer Ressource angesehen werden. In Listing 5 ist ein Ausschnitt der Elemente, die der Verarbeitung des Requests dienen.

```
1  @Produces(MediaType.APPLICATION_JSON)
2  public class QueryResultResource {
3      ...
4      @GET
5      public Response getQueryResultResAsJson(@DefaultValue("col") @QueryParam("
           orientation") String orientation ){
6          if(orientation.equals("row")) this.roworiented = true;
7          requestedMediaType = MediaType.APPLICATION_JSON;
8          buildQueryResultObject();
9          return Response.ok(RepresentationHelper.createQueryResultJson(this)).
               build();
10     }
11
12     @GET
13     @Produces(MediaType.TEXT_HTML)
14     public Response getQueryResultResAsHtml(){
15         requestedMediaType = MediaType.TEXT_HTML;
16         buildQueryResultObject();
17         return Response.ok(RepresentationHelper.createQueryResultHtml(this)).
               build();
18     }
19
20     @Path("schema")
21     public QueryResultSchemaResource getSchemaResource(){
22         buildQueryResultObject();
23         return new QueryResultSchemaResource(this);
24     }
25 }
```

Listing 5: Ausschnitt QueryResultResource.java

[10] Das Klassendiagramm wurde erstellt mit dem Programm IntelliJ IDEA 14.1.4.

Die Annotation `@Produces(MediaType.APPLICATION_JSON)` über den Klassennamen, legt das Repräsentationsformat aller Responses standardmäßig auf `application/json` fest. Innerhalb der Klasse sind drei öffentliche Methoden abgebildet. Die erste Methode wird aufgerufen, wenn ein HTTP GET-Request empfangen wird, der entweder keinen `Accept` Header oder den Header `Accept:application/json` enthält. Als Parameter der Methode wird der Query-Parameter der Request URI als String übergeben. Enthält der URI den Query-Parameter `orientation` nicht, wird die Variable standardmäßig auf „col“ gesetzt, was eine spaltenorientierte Ausgabe der Ergebnismenge bewirkt. Die private Methode `buildQueryResultObject()` löst die interne Verarbeitung des Request aus. Darin wird der Pfad des Requests in einzelne Bestandteile zerlegt, aus denen eine SQL Select-Abfrage erzeugt wird. Die Abfrage wird an die Datenbank übermittelt. Die resultierenden Ergebnisdatensätze werden in den privaten Attributen der Klasse gespeichert. Das Objekt, welches nun alle Daten der Ressource beinhaltet, wird der statischen Methode `createQueryResultJson()` übergeben, welche die Repräsentation im JSON Format als String zurückliefert. Dieser String wird als Inhalt der Response-Nachricht definiert. Die statische Methode `ok()` gibt an, dass der Statuscode der Response `200 (OK)` ist.

Nach dieser Erläuterung ist die Funktion der zweiten Methode weitestgehend selbsterklärend. Diese wird ausgeführt, wenn mittels Content-Negotiation `text/html` vom Client als Repräsentationsformat angefordert wurde.

Für jede *QueryResultResource* existiert eine *QueryResultSchemaResource*, die das Schema der zugehörigen Ergebnismenge beinhaltet. Der URI dieser Ressource entspricht den URI der korrelierenden *QueryResultResource* mit dem Suffix `/schema` (siehe Kapitel 4.3). Die Definition dieser Eigenschaft des generischen URI-Schemas ist in Listing 5 (Zeile 20-24) zu sehen. Die Annotation in Zeile 20 legt die relative Pfadkomponente als Suffix fest. Die Methode `getSchemaResource()` zeigt die Verwendung einer Sub-Resource Klasse. Die Klasse `QueryResultSchemaResource` ist, analog zur Klasse `QueryResultResource` das objektorientierte Pendant der zugehörigen REST-Ressource. Da ein auf `/schema` endender URI die *QueryResultSchemaResource* identifiziert, gibt die Methode ein Objekt der zugehörigen Klasse zurück. Die Verarbeitung des Requests erfolgt innerhalb dieser Klasse. Dem Konstruktor der Klasse wird das gesamte Objekt der `QueryResultResource` übergeben, denn es enthält u.a. das Abfrageergebnis, aus dem das Schema konstruiert werden kann. Man erkennt abermals, dass Sub-Resources einen Mechanismus zur klaren Strukturierung der Ressourcenstruktur und damit auch des Quellcodes darstellen.

5.4. JOINs über rekursiven Aufruf der Sub-Resource

Das URI-Schema des Webservice ist so konzipiert, dass man (theoretisch) beliebig oft den relativen Pfad der Join-Klausel verwenden kann. Zur Verdeutlichung kann der folgende relative Pfad herangezogen werden:

`/select/*/from/t1/naturalJoin/t2/naturalJoin/t3/orderBy/a`

Die Unterstreichung kennzeichnet das URI-Template der Join-Klausel, welches in dem Beispiel zwei mal verwendet wurde. Um die Implementierung dieser Besonderheit möglichst verständlich darzulegen, werden die Verarbeitungsschritte eines eintreffenden GET-Requests auf einen URI, der den Beispielpfad beinhaltet, Schrittweise erläutert. Listing 6 zeigt die zugehörigen Quellcodefragmente der drei Ressourcenklassen, die dabei verwendet werden. Sämtliche Attribute und die Deklarationen der Pfadparameter wurden aus Gründen der Übersichtlichkeit weggelassen.

```
1  @Path("/")
2  public class RootPathHandler {
3
4      @Path("select/{selectExpression}/from/{tableExpression}")
5      public QuerySubResource getQueryResult(...){
6          return new QuerySubResource(...);
7      }
8      ...
9  }
10
11 public class QuerySubResource {
12
13     @Path("naturalJoin/{joinTable}")
14     public QuerySubResource getQuerySubResourceNaturalJoin(...){
15         QuerySubResource querySubResource = this;
16         querySubResource._joinTables.add(joinTable);
17         querySubResource._tableRels.add("");
18         querySubResource._joinTypes.add("natural");
19         return querySubResource;
20     }
21
22     @Path("orderBy/{orderByExpression}")
23     public OrderByQuerySubResource getOrderByResult(...){
24         return new OrderByQuerySubResource(...);
25     }
26     ...
27 }
28
29 public class OrderByQuerySubResource {
30
31     @Path("/")
32     public QueryResultResource getQueryResultResource(){
33         QueryResultResource queryResultResource = new QueryResultResource(
               _querySubResource);
34         ...
35         return queryResultResource;
36     }
```

Listing 6: Ausschnitt QuerySubResource.java

Die Interpretation des Pfades wird von links nach rechts durchgeführt. Der relative Pfad `/select/*/from/t1` bewirkt das Aufrufen der Methode `getQueryResult()` (Zeile 5) der Root-Resource Klasse, welche ein Objekt der Sub-Resource Klasse `QuerySubReource` zurückgibt. Innerhalb der `QuerySubReource` Klasse wird das nächste Pfadsegment `/naturalJoin/t2` interpretiert. Die Methode `getQuerySubResourceNaturalJoin()` (Zeile 14) wird aufgerufen, welche ein Objekt der Sub-Resource Klasse zurückgibt, in der sich die Methode selbst befindet (Zeile 19). Hier ist der rekursive Aufruf der Sub-Resource erkennbar. Das nächste Pfadsegment `/naturalJoin/t3` wird abermals von dieser Klasse interpretiert und die selbe Methode wird aufgerufen, welche wiederum ein Objekt dieser Klasse zurückgibt. Da letzte Pfad-Segment `orderBy/a` der `@Path`-Annotation (Zeile 22) der Methode `getOrderByResult()` (Zeile 23) entspricht, wird diese auch aufgerufen. Sie erzeugt ein Objekt der Sub-Resource Klasse `OrderByQuerySubResource`. Da man nun am Ende des Pfades angekommen ist, wird die darin enthaltene Ressourcenmethode aufgerufen, deren `@Path`-Annotation einen leeren Wert („/“) aufweist. In diesem Fall ist das die Methode `getQueryResultResource`. Diese erzeugt wiederum ein Objekt der im vorherigen Kapitel betrach-

teten Ressourcenklasse `QueryResultResource`. Darin wird eine Methode mit `@GET`-Annotation aufgerufen, welche die Response erzeugt, die an den Aufrufer zurückgesendet wird.

Wie schon in Kapitel 4.5 erwähnt, existiert noch keine Möglichkeit, diesen rekursiven Aufruf einer Sub-Resource in dem Dokumentationsformat RAML zu formulieren.

5.5. Ressourcen zu Repräsentationen

Der Client interagiert bei einem REST-konformen Webservice ausschließlich mit Repräsentationen der Ressourcen (s. Kapitel 2.1.3). Über den „Accept"-Header (s. Kapitel 2.1.2) teilt der Client dem Webserver (und somit auch dem Webservice) das gewünschte Repräsentationsformat als Teil der Anfrage mit [vgl. 43, S. 87]. In der Konzeption der hier zu entwickelnden API wurde festgelegt, dass für jede Ressource (ausgenommen die *DocuResource*) das Format HTML und JSON zur Verfügung stehen soll. Der für den Client bestimmte Inhalt der Ressource muss in beide Formate überführt werden. Für diese Aufgabe wurde die Klasse `RepresentationHelper` erzeugt, welche statische Methoden beinhaltet, denen ein Ressourcenobjekt als Parameter übergeben wird, diese es in das gewünschte Format (HTML oder JSON) umwandeln und als String an den Aufrufer zurückliefern. Dieser Prozess kann auch als „Mapping" bezeichnet werden. In Listing 5 ist der Aufruf einer solchen Methode aufgeführt (Zeile 20). Der entstandene String wird innerhalb einer Ressourcenmethode als Message Body der Response definiert. Die Annotation `@Produces` wird verwendet, um den Medien-Typ der Response festzulegen (siehe Listing 5, Zeile 16).

Für das Mappen der Ressource in ein Repräsentationsformat wurden zwei Technologien eingesetzt. Zur Erzeugung der HTML-Repräsentation wird die Mustache-Templating-Engine genutzt, deren Konzept und Eigenschaften schon in Kapitel 3.4.3 beschrieben sind. Listing 7 zeigt die Verwendung des Templating-Systems am Beispiel der `createQueryresultHtml()` Methode.

```
1  public static String createQueryResultHtml(QueryResultResource
       queryResultObject){
2      HashMap<String, Object> scopes = new HashMap<>();
3      String content;
4      MustacheFactory mf;
5      Mustache mustache;
6      StringWriter writer = new StringWriter();
7      String htmlResult = "";
8      InputStream inputStream;
9
10     /* Fill the Scope with data from the resource object */
11     scopes.put("query-statement", queryResultObject.getSqlStatement());
12     scopes.put("links", queryResultObject.getLinkList());
13     scopes.put("queried-tables",queryResultObject.getQueriedTables());
14     scopes.put("field-names", queryResultObject.getResultFieldList());
15     scopes.put("dataset-count", queryResultObject.getDatasetCount());
16     queryResultObject.getResultListOfList().remove(0); //delete first row (col
           names)
17     scopes.put("dataset-rows", queryResultObject.getResultListOfList());
18
19     /* read the template from the classpath */
20     inputStream = RepresentationHelper.class.getClassLoader().
           getResourceAsStream("templates/QueryResult.html");
21
22     if(inputStream != null){
23         content = new Scanner(inputStream, StandardCharsets.UTF_8.name()).
               useDelimiter("\\A").next();
```

```
24          mf = new DefaultMustacheFactory();
25          mustache = mf.compile(new StringReader(content), "QueryResult");
26
27          /* replace the placeholders with the data */
28          mustache.execute(writer, scopes);
29          htmlResult = writer.toString();
30          try {
31              inputStream.close();
32          } catch (IOException e) {
33              Log.exception(e);
34          }
35      }
36
37      writer.flush();
38      return htmlResult;
39  }
```

Listing 7: Methode createQueryResultHtml aus RepresentationHelper.java

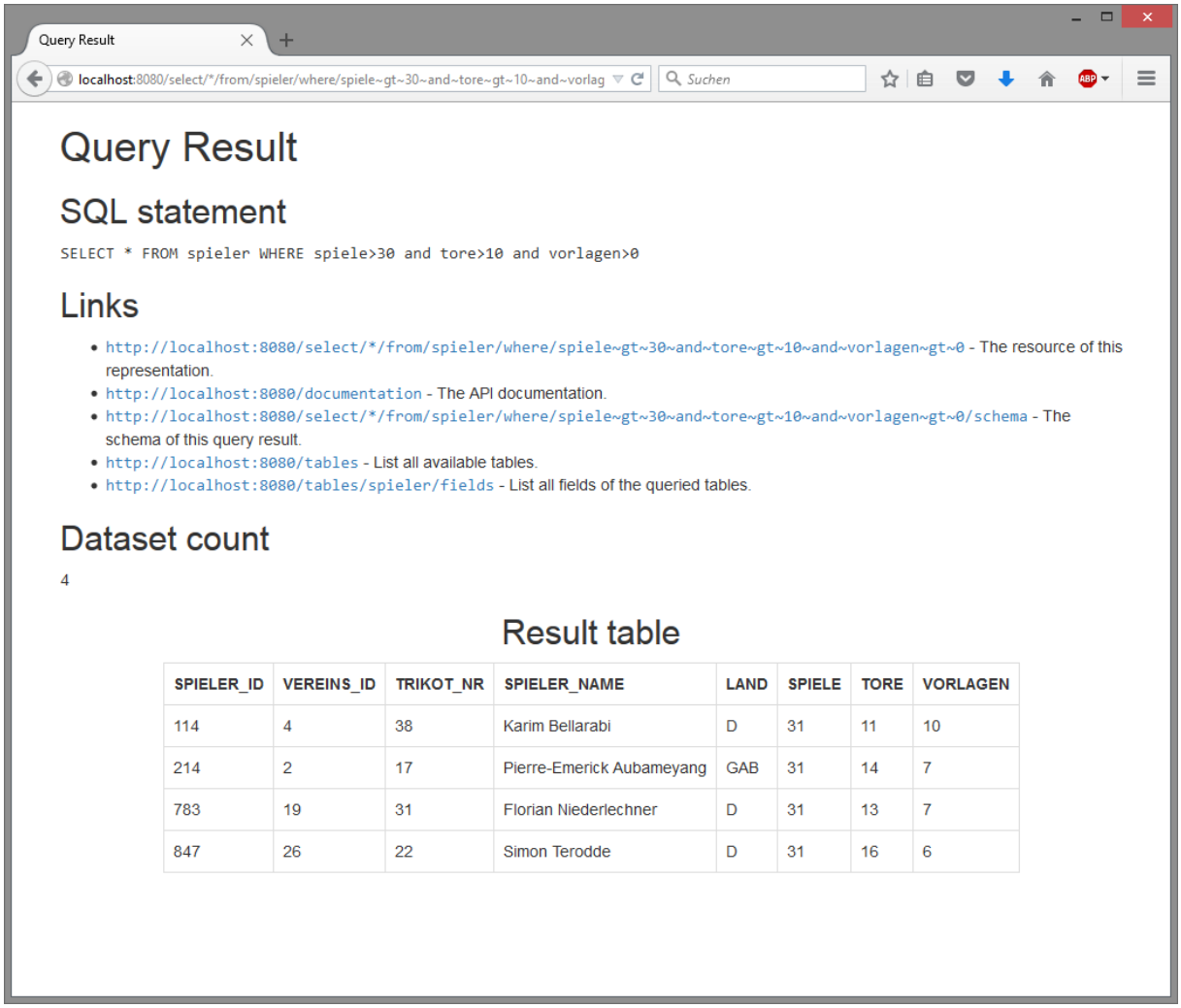
Zunächst wird eine Hashmap als Datenvorrat erzeugt (Zeile 2) und mit den Daten der übergebenen Ressource gefüllt (Zeile 11 bis 17). Die Keys der Hashmap müssen dabei zwingend mit den Namen der Platzhalter innerhalb des Templates übereinstimmen. Danach wird das HTML-Template eingelesen (Zeile 20) und in einen String konvertiert (Zeile 23). Anschließend wird ein Objekt des Templating-Systems erzeugt (Zeile 24), welches den String kompiliert (Zeile 25) und die Platzhalter mit den Daten austauscht (Zeile 28). Der dabei entstandene StringWriter, der das fertige HTML-Dokument beinhaltet, wird als String konvertiert (Zeile 29) und an den Aufrufer zurückgegeben (Zeile 38). Die Abbildung 10 zeigt die von dem Browser Firefox gerenderte HTML-Repräsentation einer *QueryResultResource*.

Zur Erzeugung der JSON-Repräsentation wird die für Java entwickelte JSON-Bibliothek „Jackson“ [41] verwendet. Listing 8 zeigt die sehr einfache Verwendung des Jackson spezifischen ObjectMapper am Beispiel der Methode `createQuerySchemaJson()`, die das Mapping der *QueryResultSchemaResource* in das JSON-Format prozessiert.

```
1  public static String createQuerySchemaJson(QueryResultSchemaResource
       queryResultSchemaObject) {
2      ObjectMapper jsonMapper = new ObjectMapper();
3      String returnString = "";
4      try{
5          returnString = jsonMapper.writeValueAsString(queryResultSchemaObject);
6      } catch (IOException e) {
7          Log.error("Error while mapping queryResultSchemaObject to JSON.");
8          Log.exception(e);
9      }
10     return returnString;
11 }
```

Listing 8: Methode createQuerySchemaJson aus RepresentationHelper.java

In Zeile 2 wird ein Objekt der Klasse ObjectMapper instantiiert, welche Bestandteil der Jackson-Bibliothek ist. Das Ressourcenobjekt wird der Methode `writeValueAsString` des ObjectMapper-Objektes übergeben, welche die Daten im JSON-Fomat als String zurückliefert (Zeile 5).



SPIELER_ID	VEREINS_ID	TRIKOT_NR	SPIELER_NAME	LAND	SPIELE	TORE	VORLAGEN
114	4	38	Karim Bellarabi	D	31	11	10
214	2	17	Pierre-Emerick Aubameyang	GAB	31	14	7
783	19	31	Florian Niederlechner	D	31	13	7
847	26	22	Simon Terodde	D	31	16	6

Abbildung 10: HTML Repräsentation der QueryResultResource

5.6. Fehlerbehandlung

Stellt der Client einen fehlerhaften Request an die API oder fordert eine Ressource bzw. eine Repräsentation an, die nicht existiert, muss mit einer Response geantwortet werden, die einen Fehlercode und eine Beschreibung des Fehlers enthält. Bei der Konzeption des Webservice wurden die Statuscodes und deren Fehlerbedingung festgelegt (siehe Kapitel 3.4.5). Die Umsetzung dieser Fehlerübermittlung wird im Jersey Framework mittels Exceptions durchgeführt. Tritt ein interner Fehler bei der Verarbeitung eines Requests auf, wird automatisch eine Response mit dem Statuscode `500 (Internal Server Error)` versendet. Dieses Verhalten ist fest in dem Framework implementiert. Zum Erzeugen von Responses mit den Statuscodes `400 (Bad Request)` und `404 (Not Found)` wurden zwei Klassen erzeugt, `CustomBadRequestException` und `CustomNotFoundException`, die beide von der Elternklasse `WebApplicationException` (aus JAX-RS) erben. Wird eine dieser Exceptions während der Verarbeitung eines Requests geworfen, wird die Verarbeitung abgebrochen und der Client erhält sofort eine Response mit den jeweils festgelegten Statuscode und einer Fehlerbeschreibung, welche dem Konstruktor der Exception übergeben werden kann. Da diese Fehler im REST-Sprachgebrauch gewissermaßen auch Res-

sourcen sind, muss sich auf ein Repräsentationsformat festgelegt werden. Da sich HTML für die Darstellung im Browser eignet, jedoch auch von einer Client-Webanwendung interpretiert werden kann, wurde es als Medien-Typ ausgewählt. Eine Methode der `RepresentationHelper` Klasse übernimmt das Mappen der Fehlerbeschreibung in das HTML-Format. Listing 9 zeigt beispielhaft die Klasse `CustomNotFoundException`.

```
1  public class CustomNotFoundException extends WebApplicationException {
2
3      /* Creates a HTTP 404 (Not found) exception. */
4      public CustomNotFoundException() {
5          super(Response.status(Response.Status.NOT_FOUND).build());
6      }
7
8      /* Creates a HTTP 404 (Not found) exception with message body in HTML. */
9      public CustomNotFoundException(String message) {
10         super(Response.status(Response.Status.NOT_FOUND).
11                 entity(RepresentationHelper.createErrorHtml(message)).type(
                       MediaType.TEXT_HTML).build());
12     }
13 }
```

Listing 9: CustomNotFoundException.java

Man kann erkennen, dass zwei Konstruktoren existieren. Der erste besitzt keine Möglichkeit der Angabe einer Fehlerbeschreibung. Wird über diesen die Exception erzeugt, versendet der Webservice eine Response mit leerem Message Body. Dem zweiten Konstruktor kann ein String übergeben werden, welcher in das HTML-Format gemappt und als Message-Body der Response definiert wird (Zeile 11).

6. Zusammenfassung

Zu Beginn der Arbeit ist die Problemstellung der individuellen Implementierung eines Webservices für einen konkreten Anwendungsfall dargelegt und die damit verbundene Motivation, einen allgemeingültigen Webservice mit generischen URI-Schema zu Entwickeln, verdeutlicht. Die damit verbundene Systemidee und das Ziel dieser Bachelor Thesis war es, ein generisches, sich an der SQL-Syntax orientierendes URI-Schema zu konzipieren und dies innerhalb eines REST-konformen Webservices zu implementieren. Bei dessen Entwicklung wurde großen Wert auf die Einhaltung Fieldings Vorgaben für eine REST Architektur gelegt. Trotzdem wurde deutlich, dass gerade im Bezug auf den Hypermedia-Aspekt nicht alle dieser Kriterien erfüllt werden konnten. Im Zuge der Entwicklung des generischen URI-Schemas musste sich zunächst über die Art der Übermittlung einer Select-Anweisung festgelegt werden. Die Anweisung innerhalb der Pfad-Komponente des URI umzusetzen, stellte sich als geeignetste Variante heraus und stützte somit die innerhalb der Einleitung beschriebene Systemidee. Durch das einheitliche Verfahren der Umwandlung einzelner Klauseln in das URI-Schema, konnte die gesamte erfasste Syntax einer Select-Abfrage auf gleiche Weise überführt werden. Sonderzeichen, die nicht in einem URI verwendet werden dürfen, wurden codiert. Ein URI-Schema zum Abrufen des Datenbankschemas wurde ebenfalls erzeugt. Die Implementierung der Anwendung samt Webserver und Webservice, konnte durch die sinnvolle Verwendung der Annotationen des Jersey-Frameworks vollständig erfolgen. Um die entstandene API zu beschreiben, wurde eine Dokumentation erstellt. Diese beinhaltet alle Informationen, die zum Erzeugen eines gültigen URIs benötigt werden. Auch Festlegungen zu HTTP-Methoden, Repräsentationsformaten, Statuscodes, sowie Pfad- und Query-Parametern sind innerhalb der Dokumentation erfasst. Durch die Konfigurationsmöglichkeit der Datenbankanbindung und der Verwendung einer standardisieren Schnittstelle, wurde eine Unabhängigkeit im Bezug auf die zu verwendende relationalen Datenbank geschaffen.

Besonders zeitaufwändig und dennoch sehr interessant war die Umsetzung der sich optional wiederholenden Join-Klauseln mit dem Jersey-Framework. Die Idee des beschriebenen rekursiven Aufrufes einer Sub-Ressource ist in keiner Literatur zu finden. Sie entstand sowohl aus der tiefgründigen Auseinandersetzung mit Jersey Annotationen und Sub-Ressourcen, als auch aus Diskussionen, die in verschiedenen Foren geführt wurden.

Mögliche Erweiterungen der entstandenen Anwendung wären beispielsweise die Integration von Unterabfragen in das URI-Schema, was den Anwendungsbereich der Lösung deutlich vergrößert, jedoch die Lesbarkeit des resultierenden URIs voraussichtlich verschlechtert. Realisiert werden könnte dies eventuell mit der selben Idee, die bei den Join-Klauseln angewandt wurde. Weiterhin könnte man die Repräsentationen im JSON Format durch ein speziell für Hypermedia angepasstes JSON Format ersetzen. Dies würde durch die damit einhergehende Standardisierung der Hypermedia-Elemente die Attraktivität des Webservices für Client-Webanwendungen erhöhen. Um das Überführen einer SQL Anweisung in das entwickelte URI-Schema dem Nutzer zu erleichtern, wäre auch ein Mechanismus denkbar, der eine beliebige SQL-Anweisung beispielsweise im Message-Body eines HTTP PUT-Requests entgegen nimmt, diese programmatisch in einen gültigen URI überführt und an den Aufrufer zurück liefert.

Die innerhalb dieser Bachelor Thesis entstandene Anwendung erfüllt alle festgelegten Anforderungen und wurde mit verschiedensten Datenbanken fehlerfrei getestet. Das definierte Ziel wurde somit erreicht. Zusammenfassend kann man die Kombination aus dem RESTful Webservice und generischen URI-Schema als allgemeingültige Softwarelösung betrachten, die ohne Implementationsaufwand zur Bereitstellung von Daten einer relationalen Datenbank für eine Client-Webanwendung eingesetzt werden kann.

Literatur

[1] Mike Amundsen. *Collection+JSON - Hypermedia Type*. 24. Feb. 2013. URL: http://amundsen.com/media-types/collection (besucht am 12. 06. 2015).

[2] Oren Ben-Kiki, Clark Evans und Ingy döt Net. *YAML: YAML Ain't Markup Language*. URL: http://yaml.org/ (besucht am 17. 08. 2015).

[3] Tim Berners-Lee. *Universal Resource Identifiers - Axioms of Web Architecture*. Techn. Ber. Dez. 1996. URL: http://www.w3.org/DesignIssues/Axioms.html.

[4] Tim Berners-Lee, Roy Thomas Fielding und L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986. RFC Editor, Jan. 2005. URL: http://www.rfc-editor.org/rfc/rfc3986.txt.

[5] E. F. Codd. *A Relational Model of Data for Large Shared Data Banks*. Whitepaper. IBM Research Laboratory, Juni 1970. URL: http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf (besucht am 03. 08. 2015).

[6] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine, 2000.

[7] Roy Thomas Fielding. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. RFC Editor, Juni 1999. URL: http://www.rfc-editor.org/rfc/rfc2616.txt.

[8] Roy Thomas Fielding. *REST APIs must be hypertext-driven*. Techn. Ber. Okt. 2008. URL: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven (besucht am 17. 08. 2015).

[9] Roy Thomas Fielding und Julian F. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. RFC 7230. RFC Editor, Juli 2014. URL: http://www.rfc-editor.org/rfc/rfc7230.txt.

[10] Roy Thomas Fielding und Julian F. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. RFC Editor, Juni 2014. URL: http://www.rfc-editor.org/rfc/rfc7231.txt.

[11] Ned Freed. *Media Types*. 18. Juni 2015. URL: http://www.iana.org/assignments/media-types/media-types.xhtml (besucht am 12. 06. 2015).

[12] Ned Freed und Nathaniel Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. RFC 2046. RFC Editor, Juni 1996. URL: http://www.rfc-editor.org/rfc/rfc2046.txt.

[13] Dmytro Gerasymchuk. „Datenbanken heute“. In: *IX Magazin 06 2015 - Moderne Datenbanken* (Juni 2015).

[14] J. Gregorio u. a. *URI Template*. RFC 6570. RFC Editor, März 2012. URL: http://www.rfc-editor.org/rfc/rfc3986.txt.

[15] Sunil Gulabani. *Developing RESTful Web Services with Jersey 2.0*. 1. Auflage. PACKT, 2014.

[16] Christian Heise und Ansgar Heise, Hrsg. *ix Magazin für professionelle Informationstechnik* (Juni 2015).

[17] *Jersey 2.7 User Guide*. 2015. URL: https://jersey.java.net/documentation/2.7/user-guide.html#preface (besucht am 12. 06. 2015).

[18] Mike Kelly. *HAL - Hypertext Application Language*. 13. Juni 2011. URL: http://stateless.co/hal_specification.html (besucht am 12. 06. 2015).

[19] Kaveh Keshavarzi und Thomas Bayer. *JSON, XML und YAML im Vergleich*. 6. Juli 2011. URL: http://predic8.de/xml-json-yaml.htm (besucht am 12. 06. 2015).

[20] *Klassendiagramm zur Datenbank Fußball-Bundesliga*. URL: http://dbup2date.uni-bayreuth.de/downloads/bundesliga/Klassendiagramm_Bundesliga.pdf (besucht am 28. 07. 2015).

[21] Graham Klyne. *Uniform Resource Identifier (URI) Schemes*. Techn. Ber. IANA, Juli 2015. URL: http://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml.

[22] "FH Köln". *SQL - Structured Query Language*. URL: http://wikis.gm.fh-koeln.de/wiki_db/Datenbanken/SQL (besucht am 03. 08. 2015).

[23] Mark Nottingham, Julian Reschke und Jan Algermissen. *Link Relation Types*. 26. Aug. 2005. URL: http://www.iana.org/assignments/link-relations/link-relations.xhtml (besucht am 17. 08. 2015).

[24] Oracle. *Java™ JDBC API*. URL: http://docs.oracle.com/javase/7/docs/technotes/guides/jdbc/ (besucht am 03. 08. 2015).

[25] Oracle. *Java™ JDBC API*. URL: http://docs.oracle.com/javase/7/docs/technotes/guides/jdbc/ (besucht am 26. 08. 2015).

[26] Oracle. *Java SE Overview*. URL: http://www.oracle.com/technetwork/java/javase/overview/index.html (besucht am 26. 08. 2015).

[27] Oracle. *Jersey - RESTful Web Services in Java*. URL: https://jersey.java.net/ (besucht am 26. 08. 2015).

[28] *Project Grizzly*. URL: https://grizzly.java.net/ (besucht am 26. 08. 2015).

[29] Gunther Rademacher. *Railroad Diagram Generator*. URL: http://www.bottlecaps.de/rr/ui (besucht am 30. 07. 2015).

[30] Erhard Rahm. *Mehrrechner-Datenbanksysteme: Grundlagen der verteilten und parallelen Datenbankverarbeitung*. Oldenbourg-Verlag, 1994. URL: http://dbs.uni-leipzig.de/file/MRDBS-rahm-1994.pdf (besucht am 03. 08. 2015).

[31] *Relationale Datenbank*. 29. Dez. 2009. URL: http://www.itwissen.info/definition/lexikon/Relationale-Datenbank-database-relational.html (besucht am 03. 08. 2015).

[32] Kevin Renskers. *RAML to HTML documentation generator*. URL: https://github.com/kevinrenskers/raml2html (besucht am 17. 08. 2015).

[33] *RESTful Web Services in Java*. 8. Juni 2015. URL: https://jersey.java.net/ (besucht am 12. 06. 2015).

[34] Leonard Richardson und Sam Ruby. *RESTful Web Services*. Sebastopol: Ö'Reilly Media, Inc.", 2008.

[35] Leonard Richardson und Sam Ruby. *Web-Services mit REST*. 1. Aufl. Köln: O'Reilly Germany, 2007.

[36] Jennifer Niederst Robbins. *HTML & XHTML - kurz & gut*. 3. Aufl. Sebastopol: O'Reilly, 2006.

[37] *SELECT (Transact-SQL)*. URL: https://msdn.microsoft.com/de-de/library/ms189499(v=sql.120).aspx (besucht am 24. 07. 2015).

[38] Richard Strohm. *Oracle Data Types*. 1. Jan. 2011. URL: http://docs.oracle.com/cd/B28359_01/server.111/b28318/datatype.htm#CNCPT012 (besucht am 20.06.2015).

[39] Kevin Swiber. *Siren: a hypermedia specification for representing entities*. URL: https://github.com/kevinswiber/siren (besucht am 12.06.2015).

[40] Ed T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 7159. RFC Editor, März 2014. URL: http://www.rfc-editor.org/rfc/rfc7159.txt.

[41] "Jackson development team". *Jackson JSON Processor*. URL: http://wiki.fasterxml.com/JacksonHome (besucht am 18.08.2015).

[42] „The Free On-line Dictionary of Computing - hierarchy“. In: (19. Juli 2015). URL: http://dictionary.reference.com/browse/hierarchy (besucht am 19.07.2015).

[43] Stefan Tilkov u. a. *REST und HTTP - Entwicklung und Integration nach dem Architekturstil des Web*. 3. akt. und erw. Auflage. dpunkt, 2015.

[44] Twitter. *Twitter Bootstrap*. URL: http://holdirbootstrap.de/ (besucht am 12.07.2015).

[45] Christian Ullenboom. *Java ist auch eine Insel*. 9. Auflage. Galileo Computing, 2010.

[46] *Universal-Lexikon - generisch*. URL: http://universal_lexikon.deacademic.com/29290/generisch (besucht am 27.07.2015).

[47] Sascha Üreten. *Web- statt Desktop-Framework: Umstieg mit System*. 29. Dez. 2009. URL: http://heise.de/-893435 (besucht am 30.07.2015).

[48] "XML Core Working Group - W3C". *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. 28. Nov. 2008. URL: http://www.w3.org/TR/xml/ (besucht am 12.06.2015).

[49] "w3schools.com". *XML - XLink*. URL: http://www.w3schools.com/xml/xml_xlink.asp (besucht am 17.08.2015).

[50] Chris Wanstrath. *Mustache - Logic-less templates*. URL: https://mustache.github.io/ (besucht am 12.07.2015).

[51] Chris Wanstrath. *Mustache - Manual*. URL: https://mustache.github.io/mustache.5.html (besucht am 12.07.2015).

[52] John-Harry Wieken. *Erfolgreich SQL lernen* -. 2. Auflage. Amsterdam: Addison-Wesley, 2012. ISBN: 978-3-8-33086-4.

[53] "RAML workgroup". *RESTful API Modeling Language*. URL: http://raml.org/ (besucht am 17.08.2015).

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir an der Hochschule für Telekommunikation Leipzig (FH) eingereichte Abschlussarbeit zum Thema

Entwicklung eines generischen RESTful URI Schemas zur Ausführung von Datenbankabfragen

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Mücheln, den 07. September 2015

Christopher Rost

A. Anhang

A.1. Ausschnitt I der RAML API-Dokumentation

```
/documentation:
  displayName: API Documentation
  description: The documentation of the SQL over HTTP API.
  get:
    description: Returns this API documentation as html or as raml.
    responses:
      200:
        body:
          text/html: !!null
          application/raml+yaml: !!null
/tables:
  displayName: Explore database - II
    description: The list of all tables existing in the connected database.
      get:
        description: Returns a list of all tables existing in the connected
            database.
        responses:
          200:
            body:
              application/json:
                example: !include examples/table_list_example.json
              text/html:
                example: !include examples/table_list_example.html
  /{tableName}:
    displayName: Table
    description: Details of one or more tables.
    uriParameters:
     tableName:
       displayName: Table Name
       description: The name of the table(s). You can ...
       type: string
    get:
      description: Returns details of the table(s) given by the path parameter
          {tableName}.
      responses:
        200:
          body:
            application/json:
              example: !include examples/table_example.json
            text/html:
              example: !include examples/table_example.html
        400:
            description: There are one ore more failures in the request URI,
                ...
            body:
              text/plain:
                example: |
                  Forbidden char detected in URI parameter: space
        404:
          description: The resource you requested doesn't exist. Maybe ...
          body:
            text/plain:
              example: |
```

```
51                Tabelle "SIELER" nicht gefunden
52                Table "SIELER" not found; SQL statement:
53                SELECT * FROM SIELER [42102-187]
54    /fields:
55      displayName: Field List
56      description: List all fields of one ore more table(s).
57        get:
58          description: Returns a list of all fields existing in the given
                table(s).
59          responses:
60            200:
61              description: The requested URI is OK and the list ...
62              body:
63                application/json:
64                  example: !include examples/field_list_example.json
65                text/html:
66                  example: !include examples/field_list_example.html
67            400:
68              description: There are one ore more failures in ...
69              body:
70                text/plain:
71                  example: |
72                    Forbidden char detected in URI parameter: space
73            404:
74              description: The resource you requested doesn't exist. Maybe ...
75              body:
76                text/plain:
77                  example: |
78                    Tabelle "SIELER" nicht gefunden
79                    Table "SIELER" not found; SQL statement:
80                    SELECT * FROM SIELER [42102-187]
81      /{fieldName}:
82        displayName: Field
83        description: Details of a field.
84        uriParameters:
85         fieldName:
86           displayName: Field Name
87           description: The name of the field.
88           type: string
89        get:
90          description: Returns details of the field {fieldName} ...
91          responses:
92            200:
93              body:
94                application/json:
95                  example: !include examples/field_example.json
96                text/html:
97                  example: !include examples/field_example.html
98            400:
99              description: There are one ore more failures in ...
100             body:
101               text/plain:
102                 example: |
103                   Forbidden char detected in URI parameter: space
104           404:
105             description: The resource you requested doesn't exist. Maybe ...
106             body:
107               text/plain:
```

```
                    example: |
                      Tabelle "SIELER" nicht gefunden
                      Table "SIELER" not found; SQL statement:
                      SELECT * FROM SIELER [42102-187]
```

Listing 10: Ausschnitt I der RAML API-Dokumentation zum Abrufen des DB-Schemas

A.2. Ausschnitt II der RAML API-Dokumentation

```
#%RAML 0.8
# Created by Christopher Rost (rost@gesundheitsforen.net)
title: SQL over HTTP
version: 1.0
baseUri: http://localhost:8080
[...]
resourceTypes:
  - queryResult:
      usage: This resourceType should be used for any query result resources
      description: The result of the SQL query.
      get:
        description: |
          If the syntax of the requested URI is OK and ...
        queryParameters:
          orientation:
            description: Specify the orientation of the result ...
            type: string
            required: false
            example: row
        responses:
          200:
            description: The requested SQL statement is OK and ...
            body:
              application/json:
                example: !include examples/query_example.json
              text/html:
                example: !include examples/query_example.html
          400:
            description: There are one ore more failures in the request URI...
            body:
              text/plain:
                example: |
                  Forbidden char detected in URI parameter: space
          404:
            description: The resource you requested doesn't exist. Maybe ...
            body:
              text/plain:
                example: |
                  Tabelle "SIELER" nicht gefunden
                  Table "SIELER" not found; SQL statement:
                  SELECT * FROM SIELER [42102-187]
[...]
/select/{selectExpression}/from/{tableExpression}:
  displayName: Querying the database
  type: queryResult
  uriParameters:
    selectExpression:
```

```
48        displayName: Select Expression
49        description: |
50          The select expression located ...
51        type: string
52        required: true
53        example: id~AS~identification,timestamp,count(*)
54      tableExpression:
55        displayName: Table Expression
56        description: |
57          The table expression ...
58        type: string
59        required: true
60        example: reportemg,reporthmg~as~hmgreport
```

Listing 11: Ausschnitt II der RAML API-Dokumentation zum übermitteln einer SQL Abfrage

A.3. HTML-Repräsentation der RAML API Dokumentation

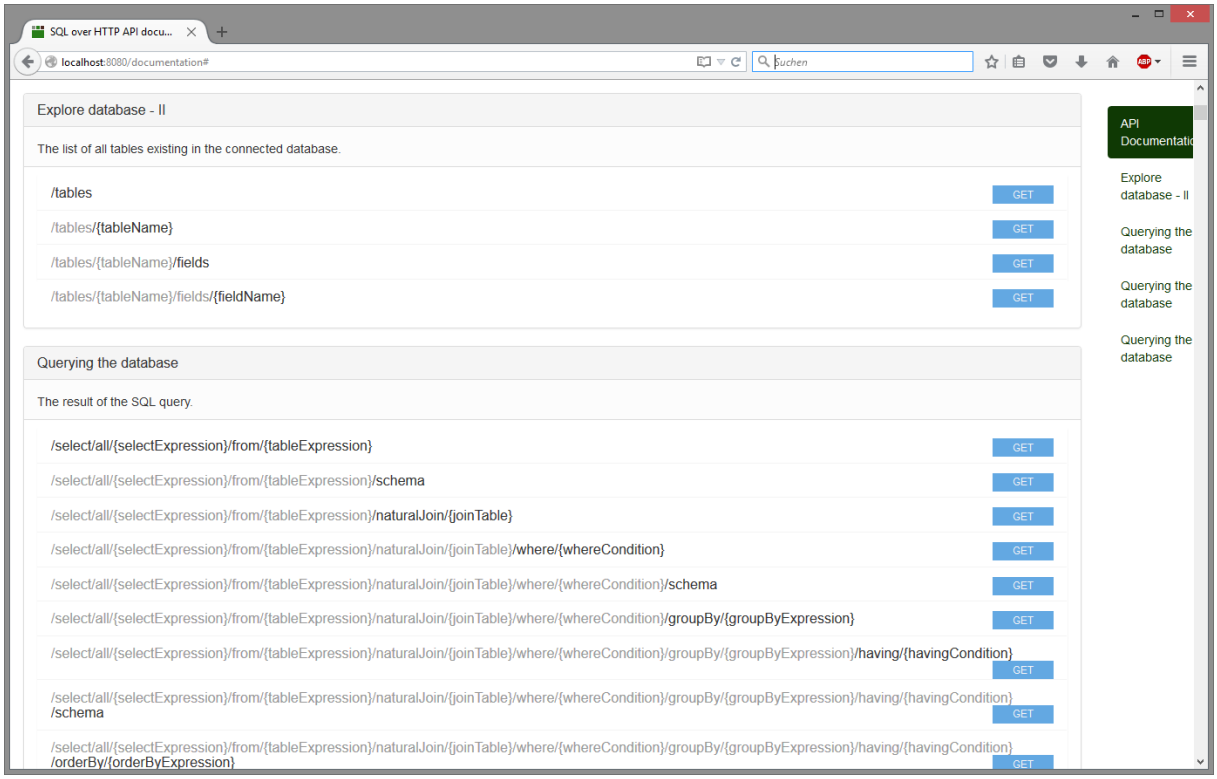
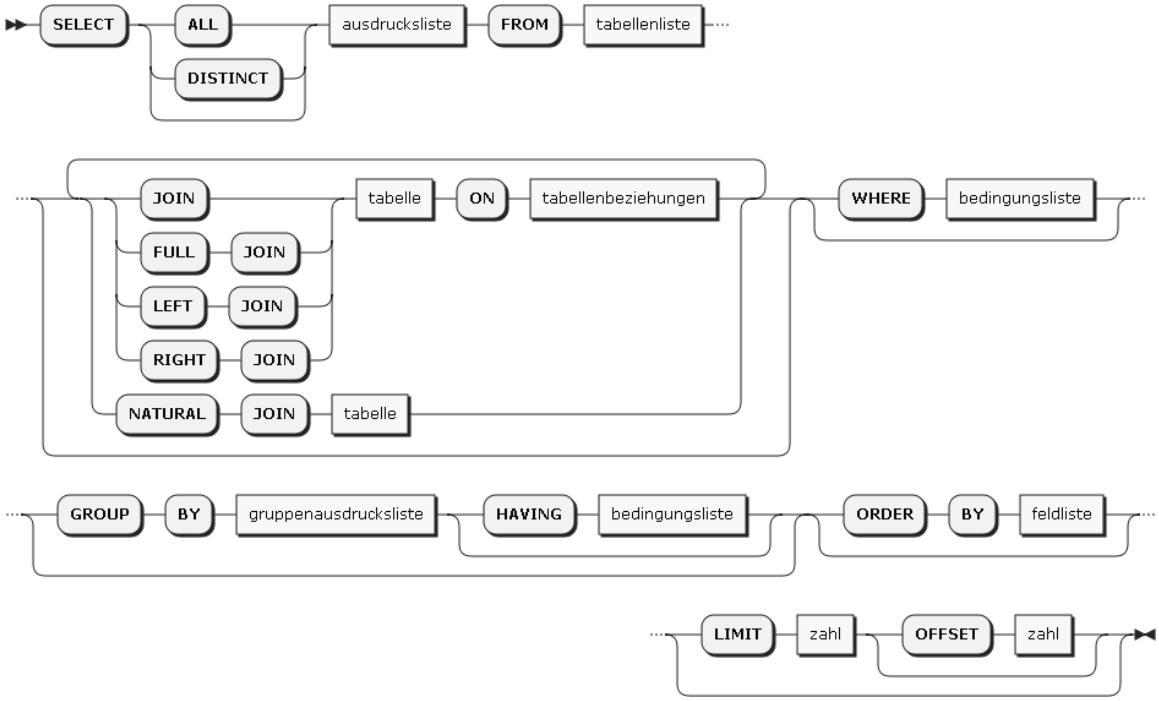


Abbildung 11: Screenshot der HTML-Repräsentation im Browser

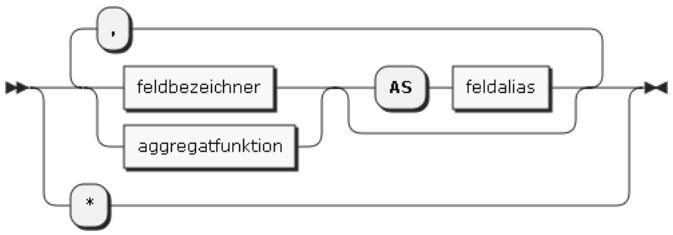
A.4. SQL SELECT Syntax in EBNF

sql-select:



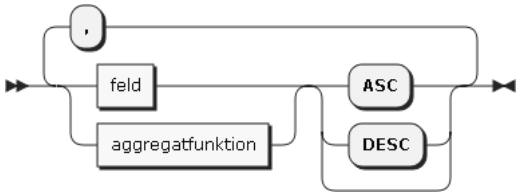

```
sql-select ::= 'SELECT' ('ALL'|'DISTINCT')? ausdrucksliste 'FROM'
    tabellenliste (('JOIN' | 'FULL' 'JOIN' | 'LEFT' 'JOIN' | 'RIGHT' 'JOIN')
    tabelle 'ON' tabellenbeziehungen) | 'NATURAL' 'JOIN' tabelle)* ( 'WHERE'
    bedingungsliste)? ( 'GROUP' 'BY' gruppenausdrucksliste ('HAVING'
    bedingungsliste)?)? ( 'ORDER' 'BY' feldliste)? ( 'LIMIT' zahl ( 'OFFSET'
    zahl)?)?
```

ausdrucksliste:



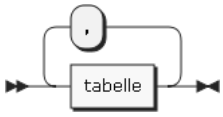

```
ausdrucksliste ::= ((feldbezeichner | aggregatfunktion) ('AS' feldalias)? (','
    (feldbezeichner | aggregatfunktion) ('AS' feldalias)?)*) | '*'
```

feldliste:



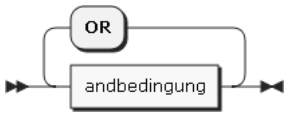

```
feldliste ::= (feld | aggregatfunktion) ('ASC'|'DESC')? (',' (feld |
    aggregatfunktion) ('ASC'|'DESC')?)*
```

tabellenliste:




```
tabellenliste ::= tabelle (',' tabelle)*
```

bedingungsliste:

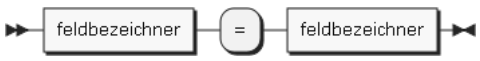



```
bedingungsliste ::= andbedingung ('OR' andbedingung )*
```

tabellenbeziehungen:

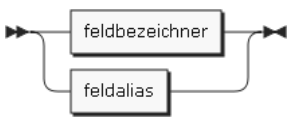
```
tabellenbeziehungen ::= tabellenverbindung ('AND' tabellenverbindung)*
```

tabellenverbindung:



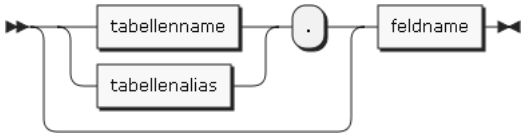

```
tabellenverbindung ::= feldbezeichner '=' feldbezeichner
```

feld:



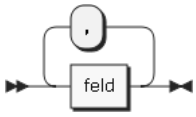
```
feld ::= (feldbezeichner | feldalias)
```

feldbezeichner:



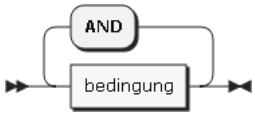

```
feldbezeichner ::= ((tabellenname | tabellenalias) '.')? feldname
```

gruppenausdrucksliste:




```
gruppenausdrucksliste ::= feld (',' feld)*
```

andbedingung:

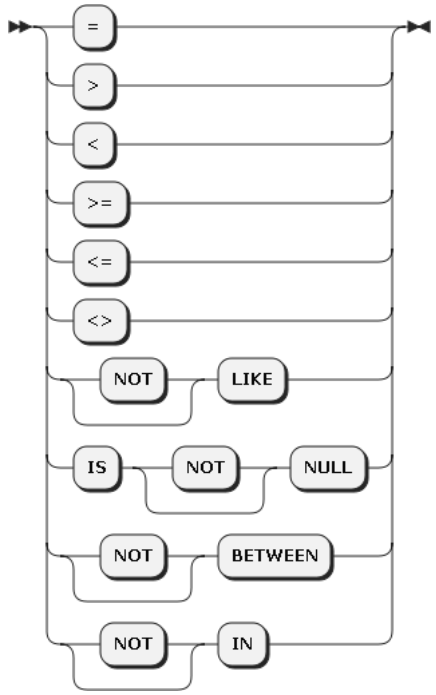



```
andbedingung ::= bedingung ('AND' bedingung)*
```

bedingung:

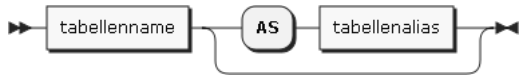
```
bedingung ::= feldbezeichner operator ausdruck
```

operator:



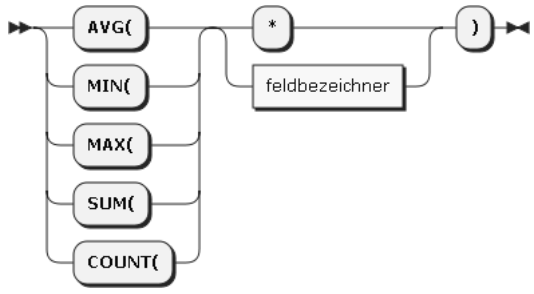

```
operator ::= ('='|'>'|'<'|'>='|'<='|'<>'| 'NOT'? 'LIKE' | 'IS' 'NOT'? 'NULL' |
    'NOT'? 'BETWEEN' | 'NOT'? 'IN')
```

tabelle:



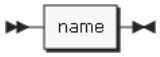

```
tabelle ::= tabellenname ('AS' tabellenalias)?
```

aggregatfunktion:



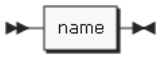

```
aggregatfunktion ::= ('AVG(' | 'MIN(' | 'MAX(' | 'SUM(' | 'COUNT(') ('*' |
   feldbezeichner) ')'
```

tabellenalias:



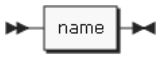

```
tabellenalias ::= name
```

feldalias



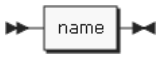

```
feldalias ::= name
```

tabellenname:



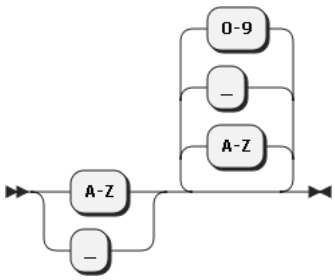

```
tabellenname ::= name
```

feldname:



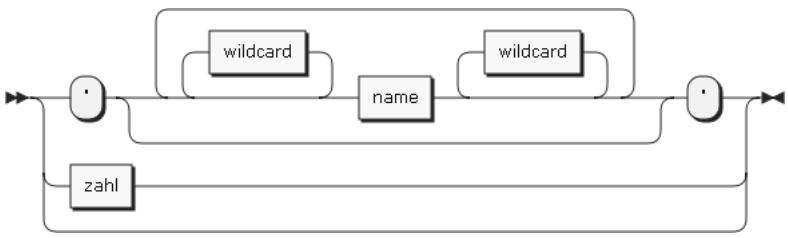

```
feldname ::= name
```

name:




```
name ::= ('A-Z' | '_') ('A-Z' | '_' | '0-9')*
```

ausdruck:

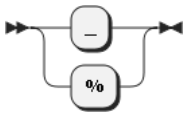


```
ausdruck ::= (("'"( wildcard* name wildcard* )* "'") | zahl )?
```

zahl:

```
zahl ::= '0-9' ('0-9')*
```

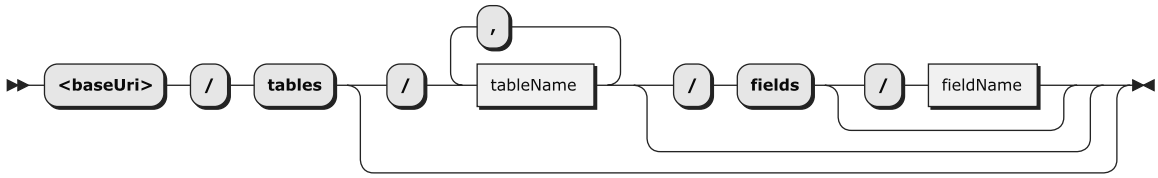
wildcard:




```
wildcard ::= '_' | '%'
```

A.5. RESTful URI-Schema zur Identifikation der Datenbankschema beschreibenden Ressourcen in EBNF

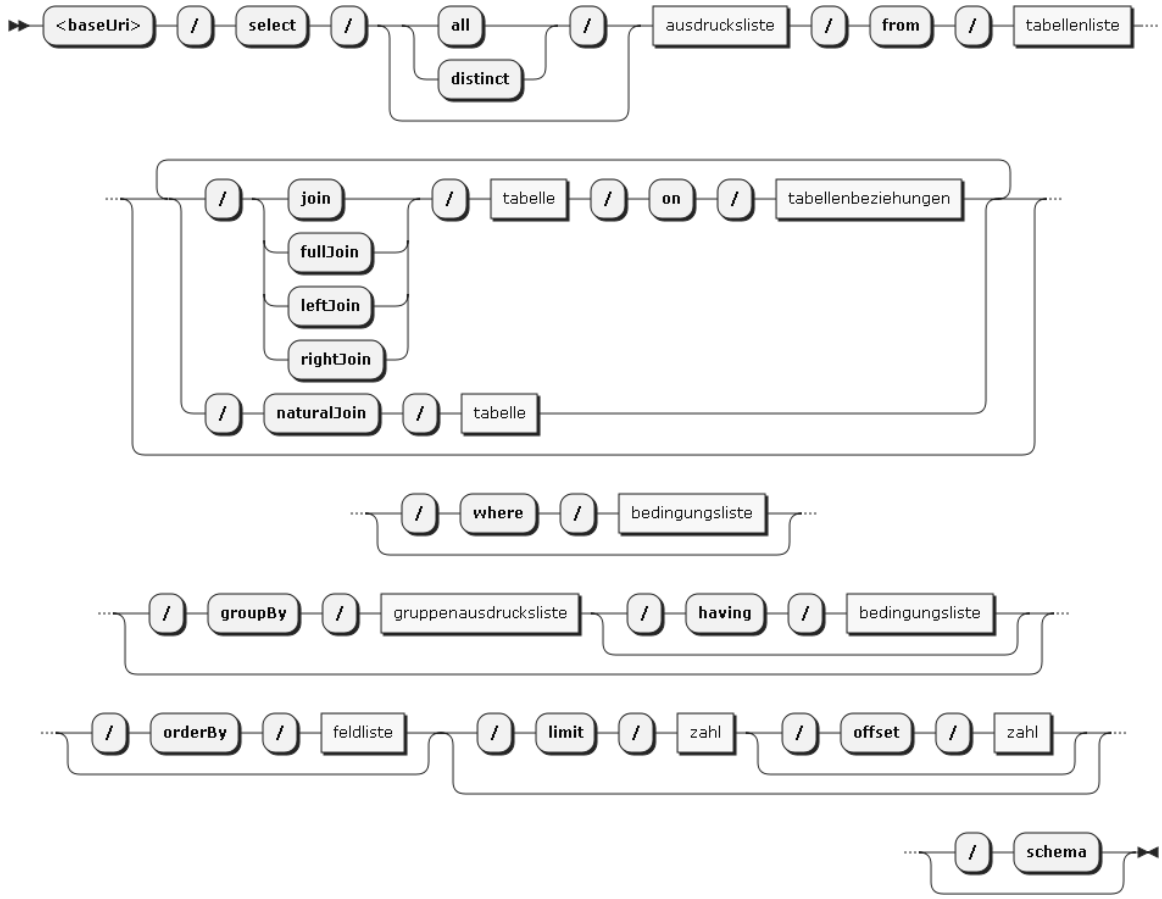
uri-schema-discover:




```
uri-schema ::= '<baseUri>' '/' 'tables' ('/' tableName (',' tableName)* ('/' '
    fields' ('/' fieldName)?)?)?
```

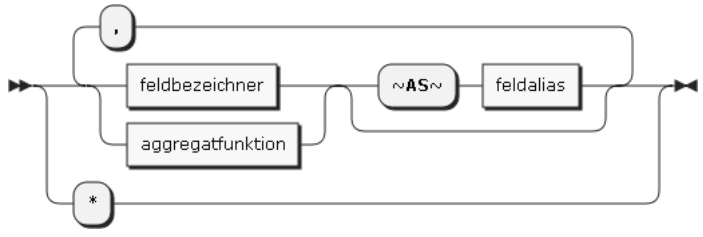
A.6. RESTful URI-Schema zur Übermittlung einer SQL Selecgt-Abfrage in EBNF

uri-schema:



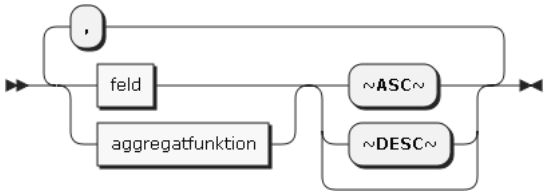

```
uri-schema ::= <baseUri> '/' 'select' '/' ( ( 'all' | 'distinct' ) '/' )?
    ausdrucksliste '/' 'from' '/' tabellenliste ( '/' ( 'join' | 'fullJoin' |
    'leftJoin' | 'rightJoin' ) '/' tabelle '/' 'on' '/' tabellenbeziehungen |
    '/' 'naturalJoin' '/' tabelle )* ( '/' 'where' '/' bedingungsliste )? ( '/
    ' 'groupBy' '/' gruppenausdrucksliste ( '/' 'having' '/' bedingungsliste )
    ? )? ( '/' 'orderBy' '/' feldliste )? ( '/' 'limit' '/' zahl ( '/' 'offset
    ' '/' zahl )? )? ( '/' 'schema' )?
```

ausdrucksliste:



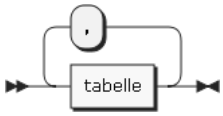
```
ausdrucksliste ::= ( feldbezeichner | aggregatfunktion ) ( '~AS~' feldalias )?
    ( ',' ( feldbezeichner | aggregatfunktion ) ( '~AS~' feldalias )? )* | '*
    '
```

feldliste:




```
feldliste
::= ( feld | aggregatfunktion ) ( '~ASC~' | '~DESC~' )? ( ',' ( feld |
    aggregatfunktion ) ( '~ASC~' | '~DESC~' )? )*
```

tabellenliste:




```
tabellenliste ::= tabelle ( ',' tabelle )*
```

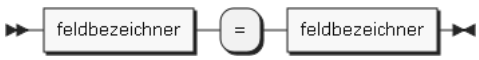
bedingungsliste:

```
bedingungsliste ::= andbedingung ( '~OR~' andbedingung )*
```

tabellenbeziehungen:

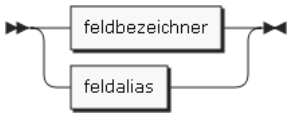
```
tabellenbeziehungen ::= tabellenverbindung ( '~AND~' tabellenverbindung )*
```

tabellenverbindung:



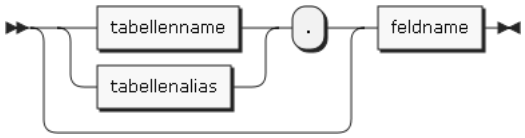

```
tabellenverbindung ::= feldbezeichner '=' feldbezeichner
```

feld:



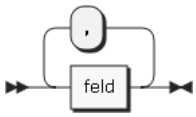

```
feld ::= feldbezeichner | feldalias
```

feldbezeichner:




```
feldbezeichner ::= ( ( tabellenname | tabellenalias ) '.' )? feldname
```

gruppenausdrucksliste:




```
gruppenausdrucksliste ::= feld ( ',' feld )*
```

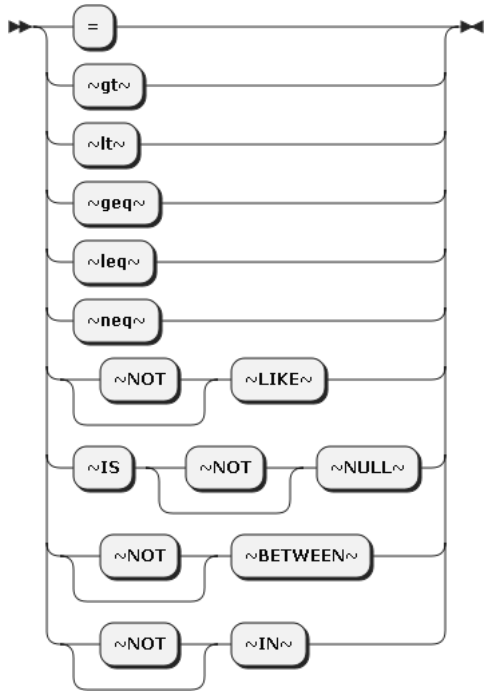
andbedingung:

```
andbedingung ::= bedingung ( '~AND~' bedingung )*
```

bedingung:

```
bedingung ::= feldbezeichner operator ausdruck
```

operator:

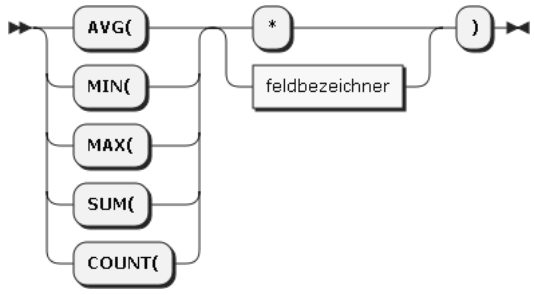



```
operator ::= '='
| '~gt~'
| '~lt~'
| '~geq~'
| '~leq~'
| '~neq~'
| '~NOT'? '~LIKE~'
| '~IS' '~NOT'? '~NULL~'
| '~NOT'? '~BETWEEN~'
| '~NOT'? '~IN~'
```

tabelle:

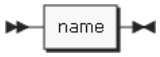
```
tabelle  ::= tabellenname ( '~AS~' tabellenalias )?
```

aggregatfunktion:



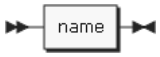
```
aggregatfunktion ::= ( 'AVG(' | 'MIN(' | 'MAX(' | 'SUM(' | 'COUNT(' ) ( '*' |
    feldbezeichner ) ')'
```

tabellenalias:



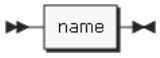

```
tabellenalias ::= name
```

feldalias:



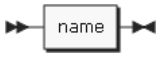

```
feldalias ::= name
```

tabellenname:



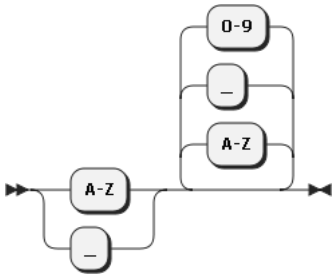

```
tabellenname ::= name
```

feldname:



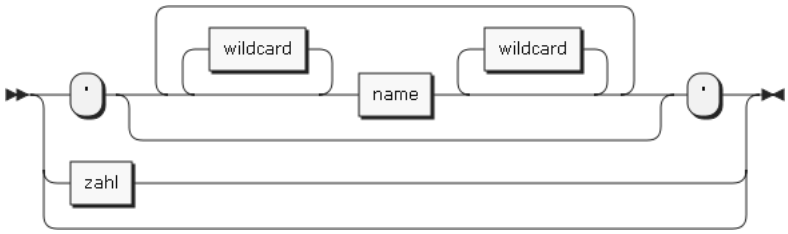

```
feldname ::= name
```

name:




```
name      ::= ( 'A-Z' | '_' ) ( 'A-Z' | '_' | '0-9' )*
```

ausdruck:

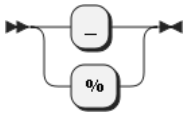



```
ausdruck ::= ( "'" ( wildcard* name wildcard* )* "'" | zahl )?
```

zahl:

```
zahl     ::= '0-9' '0-9'*
```

wildcard:




```
wildcard ::= '_' | '%'
```

A.7. Java-Anwendung auf CD

Auf der CD befindet sich das Java-Projekt als Quellcode und ausführbares Java-Archiv (jar). In dem Projekt ist auch die API Dokumentation als *.raml und *.html Datei vorhanden, sowie die HTML-Templates, die für die Repräsentationen der Ressourcen verwendet wurden.