REpresentational State Transfer - Architekturstil

Felix Scheidel

Zusammenfassung
Das Abstract ist eine maximal 200 Worte lange Zusammenfassung des Inhalts der Arbeit, so dass sich der Leser vorab ein erstes Bild vom Inhalt machen kann.

Keywords
Web-Entwicklung, RESTful, SOA, JSON, Design, Java

Hochschule Kaiserslautern
Corresponding author: fesc0013@stud.hs-kl.de

Inhaltsverzeichnis

Einleitung

Dieser Artikel soll einen Einblick in die Hintergründe und die Implementierung einer *RESTful-API* gewähren. Die Abkürzung REST entsteht aus dem eigentlichen Namen *REpresentational State Transfer*. Wenn von einer Anwendung des REST-Architekturstil gesprochen wird handelt es sich um eine Applikation die unter gewissen Regeln definiert und programmiert wurde. Diese Regeln wurden erstmals in der Dissertation von Roy Thomas Fieldings im Jahre 2000 nieder geschrieben (vgl. [2]). Zu dieser Zeit waren in Sachen Service-orientierte Applikationen noch klassisch das SOAP (*Simple Object Access Protocol*) an erster Stelle. Faktoren wie dessen Schwerfälligkeit und die unflexible Handhabung haben Fielding dazu bewegt ein Modell zu entwickeln um leichte und agile Web Services zu kreieren. Web Services bilden mittlerweile einen Grundbaustein für den Alltag. Man muss sich nur mal die Apps auf den Smartphones von Jedermann betrachten. Ob bestellen über Amazon, den Tankvorgang eine Elektroautos zu abzuwickeln oder das komplette Haus mit einer App zu steuern. Alles ist möglich weil Web Services einfach und effizient entwickelt werden können. Das Smartphone dient somit als Client und die unterschiedlichen Applikationen können mit den jeweiligen Servern Informationsaustausch betreiben. Hierzu wird bis auf wenige Ausnahmen auf eine RESTful-API zurück gegriffen.
Im weiteren Verlauf des Artikels werden die Kriterien nach Fielding beschrieben. Weiterhin werden Unterschiede zum SOAP herausgestellt und ausgewertet. Am Beispiel eines Filme Portfolios wird im Kleinen eine mögliche Implementierung erstellt und die einzelnen Bestandteile werden erläutert. Abschließend folgt ein kleiner Ausblick auf ein Resultat welches durch Web Services via REST ermöglicht wurde. Die Microservice-Architektur.

1. REST: State of the art

Um zu verstehen, weshalb REST allgegenwärtig in der Entwicklung von Web-Applikationen ist, werden im Folgenden die Eigenschaften datiert, welche ein REST Interface spezifiziert. Um weiterhin die Unterschiede, sowie Vor- und Nachteile, zu diskutieren, wird eine Gegenüberstellung mit dem SOAP vorgenommen.

1.1 Paradigmen und Eigenschaften

Die Aufbau aktueller Web-Applikationen beruht auf der Überlegung einer Service orientierten Kommunikation. Aufbauend auf der *Service Oriented Aarchitecture(SOA)* wurde vom W3C die Definition für die noch heute gültige *Web Services Architecture* [1] festgelegt. Hier wird schon REST als Modell für das konstruieren für Web Services verwendet. Eine schematische Veranschaulichung finde sich Abbildung 1. Um jedoch eine REST-konforme Architektur zu realisieren wurden von Roy Thomas Fielding in seiner Dissertation [2] eine Anzahl von Voraussetzungen festgeschrieben, die eine Architektur auf Basis von REST erfüllen muss:

- **Client-Server:**
  Der Ansatz der Client-Server Architektur soll verfolgt werden. Durch das Lösen der UI von den Daten der Applikation wird eine Portabilität, Skalierbarkeit und das

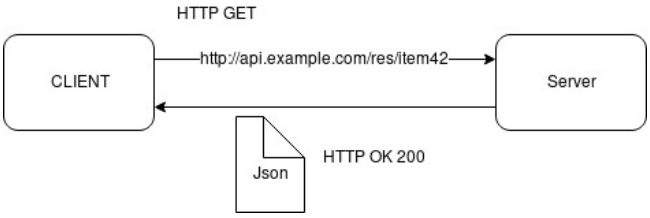

Abbildung 1. Schematischer Aufbau WSA via REST

Vereinfachen der Server-Komponente realisiert, was mittlerweile als Grundlage der kompletten Internet-Architektur gilt, denn der HTTP-Standard selbst beruht auf dieser Architektur.

- **Stateless:**
  Die Zustandslosigkeit gilt für die Kommunikation zwischen Client und Server. Anfragen des Clients müssen alle Information enthalten um vom Server identifiziert und bearbeitet werden zu können. Einen Zustand oder Session-Informationen liegen somit komplett auf Seiten des Clients.

- **Cache:**
  Die Antwort auf eine Client-Anfrage kann vom Server als cacheble oder non-cacheble gelabelt werden. Der Cache befindet sich auf der Client-Seite und erlaubt dem Client die Antwort für weitere Anfragen gleicher Art zu verwenden. Dadurch kann teilweise eine Kommunikation zwischen Client und Server unterbunden werden was die Performanz und Effizienz client-seitig stark erhöhen kann.

- **Uniform Interface:**
  Das Uniform Interface repräsentiert das fundamentale Wesen eines REST-Services. Durch die Uniformierung jeder Ressource werden die Daten von der Architektur gelöst, was eine starke Entkopplung sowie Vereinfachung der Interaktion mit sich bringt. Weiterhin soll jeder Client die nötigen Information und Daten erhalten, die es ihm erlaubt, Ressourcen zu identifizieren und dadurch auch zu modifizieren. Die angefragte Ressource muss nicht die selbe Repräsentation auf Server- und Client-Seite besitzen. Objekte vom Server werden also zum Beispiel via XML oder JSON übertragen, enthalten die definierten Daten, müssen aber nicht dem Datentyp des Server-Objekts entsprechen. Nach Fieldings ist hier das *Hypermedia as the engine of application state (HATEOAS)*-Prinzip mit der wichtigste Faktor. Dieses besagt, dass der Client alle Ressourcen nur über die definierten URI's beziehen kann. Durch diesen Zugang kann der Server dynamisch Daten kommunizieren und weiter verfügbare Adressen der Antwort hinzufügen. Somit wird verhindert, dass der Client fest codierte Information enthalten muss. Dadurch entsteht die Flexibilität des REST-Services.

- **Layered System:**
  Dem Client muss nicht bekannt sein, ob die Kommunikation direkt mit dem Server stattfindet oder nicht. Die Anfrage wird gestellt und welcher Server aus einem Kollektiv die Antwort sendet, kann nicht zugeordnet werden. Dadurch können Prinzipien wie Load-Balancing und unterschiedliche Web-Sicherheitskriterien realisiert werden.

- **Code on demand(optional):**
  Dieser Schritt ist optional und beschreibt die Möglichkeit nach Anfrage ausführbaren Code vom Server zum Client zu transferieren, wie zum Beispiel Skripte in JavaScript. Somit kann der Server zeitweise die Funktionalität der Client-Applikation beeinflussen.

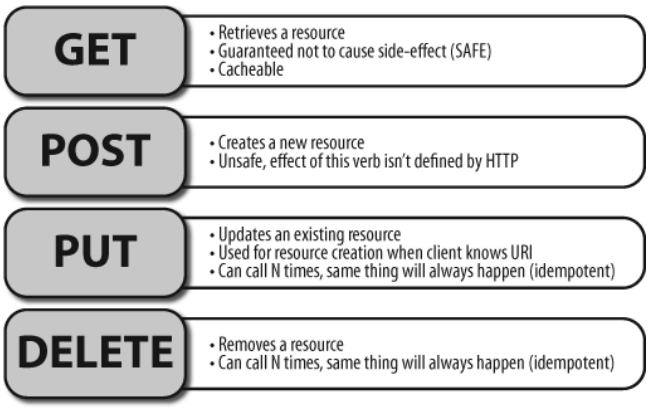

Abbildung 2. Beschreibung der Standard HTTP Methoden in der REST-Architektur. Quelle: http://www.itersdesktop.com/

Im Kontext von Web-Services, welche sich an die REST-Vorgaben binden, wird zumeist die Kommunikation über HTTP umgesetzt. Die API wird über eine Basis-URL angesprochen, über welche dann die benötigten Ressourcen angefragt werden müssen. HTTP bietet eine gewisse Anzahl an Standardmethoden(hauptsächlich: *GET, PUT, POST, DELETE*) die auch für den Datentransfer in einer RESTful-API genutzt werden. Die grundlegenden Eigenschafen der Methoden finden sich im Bild 2. Diese Methoden repräsentieren auch die bekannten *CRUD*-Operationen(Create, Read, Update, Delete) zur Datenverwaltung und Datenverarbeitung. Eine mögliche Variante mit den entsprechenden HTTP-Befehlen ist in der Tabelle 1 zu sehen. Es gilt weiterhin zu erwähnen, dass eine URL mit der entsprechenden HTTP-Methode als eindeutig gilt. Die URL kann jedoch mit einer weiteren HTTP-Methode genutzt werden. Dadurch wird festgelegt welche von den CRUD-Operation auf die entsprechende Ressource angewandt werden darf.

1.2 REST vs. SOAP

Bevor sich REST im Bereich der verteilten Anwendung etablierte, war die größte Vertreter für die Kommunikation inner-

Tabelle 1. HTTP Methoden für spezifische URL's

URL	**http://api.example.com/res/item42**
GET	Eine Repräsentation des Items42 wird an den Client übertragen
POST	Wird zum Erstellen neuer Ressourcen verwendet, würde in diesem Fall eher nicht genutzt werden
PUT	Item42 wird überschrieben oder erstellt wenn noch nicht vorhanden
DELETE	Das adressierte Element Item42 wird gelöscht

halb einer SOA das SOAP. Bei SOAP handelt es sich um eine RPC Middleware die zum Beispiel HTTP, SMTP oder FTP als Transportprotokoll für XML als Nachrichtenformat verwendet. Schon im grundsätzlichen Ansatz unterscheiden sich beide Herangehensweisen. Wird bei REST von einer exakten Uniformierung jeder Ressource gesprochen, die somit auch immer separat angesprochen werden können, so wird bei SOAP mit einem Dispacher gearbeitet. In sogenannten Envelopes werden alle Anfragen an den Server via XML mit dem HTTP-Post-Befehl an den zentralen Dispatcher geleitet. Die Anfrage gehen an genau eine URL die den Dispatcher adressiert. Im Listing 1 ist ein SOAP-Request mit seiner korrespondierenden Response zu sehen. Im Body wird eine *GetStockPrice* Anfrage gestellt für das Unternehmen IBM. Im Body der Antwort wird somit die *GetStockPriceResponse* mit dem Wert von 34,5 übermittelt. Die URL *http://www.example.org/stock* spricht den Dispatcher des Webservers an. Dieser verteilt die Anfrage an die richtige Methode, hier *GetStockPrice*. Diese Methode muss auf dem Server verfügbar sein. Durch die Anfrage wird ein Methodenaufruf mit den gelieferten Parametern durchgeführt. Das Ergebnis des Methodenaufrufs wird als Response in einem neuen Envelope zurück geliefert. Da jede Antwort einer exakten Anfrage gilt, ist ein erneutes stellen der gleichen Anfrage nicht möglich. Somit entfällt eine Möglichkeit caching auf Seiten des Clients zu betreiben.

```
 POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=
    utf-8
Content-Length: nnn

<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-
    envelope/"
soap:encodingStyle="http://www.w3.org
    /2003/05/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/
    stock">
<m:GetStockPrice>
<m:StockName>IBM</m:StockName>
</m:GetStockPrice>
</soap:Body>

</soap:Envelope>

-------------------------------------------

 HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=
    utf-8
Content-Length: nnn

<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-
    envelope/"
soap:encodingStyle="http://www.w3.org
    /2003/05/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/
    stock">
<m:GetStockPriceResponse>
<m:Price>34.5</m:Price>
</m:GetStockPriceResponse>
</soap:Body>

</soap:Envelope>
```

Listing 1. Beispiel: SOAP-Request und SOAP-Response als Envelope

Ebenso wie REST wurde SOAP vom W3C spezifiziert. Unterschied ist allerdings, bei REST handelt sich um einen Architekturstil bei SOAP um ein Protokoll. in der Spezifikation steht zum Beispiel, dass ausschließlich XML als Datenformat verwendet wird und was ein XML wohl definiert um als SOAP konform zu gelten [3]. Der Unterschied zu REST liegt hier darin, dass außer XML bei REST weitere Datenformate wie zum Beispiel JSON und YAML verwendet werden können. Die genaue Adressierung der Ressourcen unter REST erlaubt eine leichtgewichtige Transportmöglichkeit. Metadaten, wie das Encoding oder der Content-Type, werden im Header des HTTP-Requests codiert. Zusätzliche Information werden nicht benötigt. SOAP hingegen bietet diese Möglichkeit nicht. Durch die Konvention, welche Bestandteile ein XML-Envelope vorzuweisen hat, ist diese Leichtigkeit nicht zu erreichen. Das XML-Dokument muss auf Seiten des Sender aufgebaut und anschließen validiert werden um die Konformität zu bewahren. Dadurch werden zusätzliche Metadaten zur Beschreibung der Datei dem XML-Dokument hinzugefügt. Gerade bei einfachen Anfrage, zum Beispiel eine Zustandsabfrage wahr oder falsch, ist das Verhältnis von Nutz- zu Metadaten eher mäßig. Je aufwändiger die Anfrage desto besser entwickelt sich das Nutzlastenverhältnis. Anfra-

gen eine XML-Dokuments werden immer atomar ausgeführt. Somit können komplexe Sachverhalte in einer Anfrage definiert werden und sequentiell abgearbeitet werden. Bei REST werden meist leichtgewichtige Schnittstellen exponiert die genau eine Anfrage nach einer bestimmten Ressource beantworten. Somit sind bei komplexeren Schritten mehre Anfragen nötig. Dadurch ist der Kommunikationsaufwand etwas höher als bei SOAP. Relativ ähnlich verhalten sich die beiden in Bezug auf die Interaktion. Da beide Plattform unabhängige Datenformate verwenden entfaltet sich ein großen Potenzial in der Nutzung. Programme jeglicher Art, entwickelt in einer belieben Programmiersprache, welche die Datenformate ebenso unterstützen, können mit den Schnittstellen interagieren. Bei REST muss eine API-Beschreibung mit den exponierten URL's, bei SOAP die URL mit den möglichen Methodenaufrufen öffentlich gemacht werden. Bei SOAP wird die Beschreibung meist als WSDL(*Web Services Description Language*) [4] vorgelegt. In dem WSDL-File werden alle Funktionen, Daten und Datentypen beschrieben. Es werden im wesentlichen die Operation definiert, die von außen zugänglich sind. Mit diesem WSDL-File wird auch die Validierung des XML-Dokuments durchgeführt. Es handelt sich auch hierbei um einen weiteren Standard definiert vom W3C.

1.3 Evaluierung

Durch die Kriterien welche Fieldings vorgibt, sowie durch den Vergleich mit SOAP haben sich einige Punkte heraus kristallisiert, welche hier nochmals kurz zusammen gefasst werden.
REST wird als leichte Schnittstelle definiert, in welcher jede URL eine exakte Ressource anspricht. Die Kommunikation findet über HTTP mit den üblichen HTTP-Methoden statt. Dem Client ist nicht bekannt mit welchem Server genau die Kommunikation stattfindet, was eine hohe Flexibilität und Skalierbarkeit generiert. Es müssen für alle benötigten Informationen immer die entsprechenden Requests angefragt werden, was wiederum zu einem starken Kommunikationsbedürfnis zwischen Client und Server führt. Durch caching auf der Client-Seite kann dem etwas vorgebeugt werden. Meta-Informationen werden innerhalb des HTTP-Headers gespeichert und im Body befinden sich nur die Nutzdaten. Dadurch entsteht ein hohes Nutzlastverhältnis. Die Information des Bodys kann in unterschiedlichen Datenformaten verschickt werden, was eine vielfältige Anwendungsmöglichkeit bietet. Durch die vielen Vorteile ist leicht nachzuvollziehen wieso REST aktuell der vorherrschende Architektur Stil in Sachen Web-Anwendung ist. Im Bereich von großen Enterprise-Lösungen bis hin zu kleinen leichten Web-Anwendungen werden Applikationen nach dem REST-Architekturstil realisiert.

2. Entwicklungsansatz und Aufbau

Um zu Verstehen wie sich eine solche RESTful API zusammen setzt, wird in folgendem die grundlegende Spezifikation *JAX-RS 2.0 - Java API for RESTful Web Services* vorgestellt. Anschließend wird ein in Java realisiertes Beispiel geschildert um die einzelnen Komponenten zu analysieren.

2.1 JAX-RS - die Basis für REST-Services in Java

Bei JAX-RS handelt es sich um die Java eigene Programmierschnittstelle um das Entwickeln von Webservices unter Verwendung des Architekturstils REST zu vereinheitlichen. Die API wurde, bestehend aus einem Konsortium(u.a. Red Hat, Oracle, Fujitsu, Motorola) unter der Führung von Oracle/-Sun Microsystems, entwickelt und spezifiziert. Die eigentliche Spezifikation [5] stammt aus dem Jahre 2009 und wird unter dem Java Specification Request [6] 311 geführt. Es wurden jedoch nicht alle Funktionalitäten in der Version 1 beziehungsweise 1.1 spezifiziert, was zur Folge hatte, dass am Januar 2011 eine Arbeitsgruppe zur Entwicklung von JAX-RS 2.0 gegründet wurde. Die wichtigste Neuerung war das Realisieren des *HATEOAS*-Prinzips. JAX-RS ist seit Java EE 6 Teil der Plattform. In Java SE sind die Schnittstellen standardmäßig nicht vorhanden und müssen separat als Abhängigkeiten geladen werden. Die Beschreibung des JSR 339 für JAX-RS 2.0 [7] beinhaltet noch viele weitere Informationen. Da es sich bei JAX-RS um eine vollwertige Spezifikation einer Schnittstelle handelt fällt diese relativ groß aus. Da jedoch zum Programmieren des Beispiels ein Framework verwendet wird, wird auf eine detaillierte Beschreibung der Spezifikation verzichtet. Nachgelesen werden kann die Spezifikation in Quelle [5]. Ebenso wird Jersey [8] als Referenzimplementierung genannt. Sie stellt das Basis-Framework für die beiden JAX-RS API's. Um die meisten spezifizierten Eigenschaften dieser Java-API zu verwenden wie heutzutage üblich auf Annotationen zurück gegriffen. Sie vereinfachen das Entwickeln und die Übersicht und sind Hauptinitiator um Clients und Server REST-konform zu gestalten. Im folgenden Beispiel wird eine Applikation mit der schon erwähnten Implementierung Jersey 2.0 umgesetzt. Es gibt natürlich viele weitere Frameworks, wie zum Beispiel JBoss oder Spring. jedoch ist zur Veranschaulichung das Leichtgewicht Jersey ausreichend.

2.2 Beschreibung der Beispielapplikation

Das Szenario in welcher die Applikation genutzt werden soll lautet wie folgt: Ein Nutzer möchte sich sein eigenen Film-Portfolio anlegen. Er würde die Funktion gerne von seinem Rechner aber auch von seinem Smartphone aus benutzen können. Er möchte sich alle Filme die in seiner Liste sind anzeigen lassen, nach einem bestimmten Eintrag suchen, einen Eintrag hinzufügen oder einen Eintrag löschen. Beschrieben wird nun die auf dem Applikations-Server vorhandene Implementierung.
Das Projekt ist als Maven-Projekt umgesetzt und nutzt eine Version von Java SE. Alle Jersey-Abhängigkeiten sind somit in der *pom.xml* definiert. Zusätzliche Abhängigkeiten sind *Jackson* und *Gson* zum mapping von Java-Objekten zum Json-Datenformat. Als Datenbankabhängigkeit wird die H2 InMemory-Datenbank verwendet. Da es sich nur um ein Beispiel handelt wurde kein eigenständiger Applikationsserver wie zum Beispiel Tomcat aufgesetzt. Stattdessen wurde auf

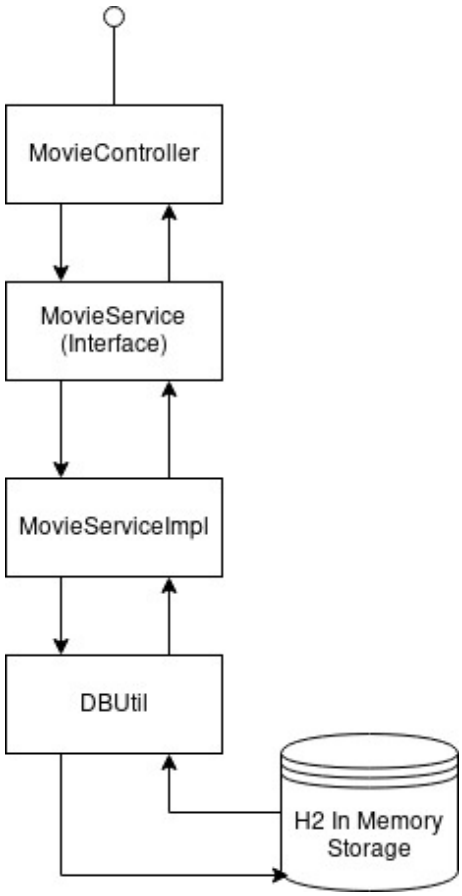

Abbildung 3. Schematischer Überblick zur Movie-Portfolio Applikation

die eigens dafür mitgebrachte JdkHttpServerFactory von Jersey zurückgegriffen. Diese erstellt zur Laufzeit einen entsprechenden HTTP-Server, sodass auch die Kommunikation via REST stattfinden kann. Die verwendete *pom.xml* findet sich im Anhang in Listing 7. Ein Überblick über den Aufbau der Applikation findet sich in Abbildung 3.

2.3 Serverseite Beispielimplementierung

Im eben erwähnten Überblick lassen sich einige Information ableiten. Die Applikation exponiert eine Schnittstelle. Über diese kann Informationsaustausch in beide Richtungen stattfinden. Ebenso lässt sich erkennen, dass der *MovieSerivce* als Interface abgebildet ist und die eigentliche Implementierung nachgelagert realisiert wird. In dem Schemata wird das Erstellen des vorab erwähnten HTTP-Servers nicht abgebildet. Da der eingebettete Server von Jersey verwendet wird, fällt die Konfiguration ziemlich gering aus. Die Initialisierung wird in der *MainClass* beim starten der Anwendung direkt mit ausgeführt(vgl. Listing 2). Als Ressourcenkonfiguration wird die MovieResource-Klasse angegeben. Diese enthält die Pfade und Methoden die vom HTTP-Server initialisiert werden sollen, sodass Aufrufe dementsprechend geroutet werden können. Die einzelnen Bestandteile werden in Folgendem im Detail beschrieben.

```
{
    public static void main(String[] args)
        throws IOException, SQLException,
        ClassNotFoundException,
        URISyntaxException
    {
        System.out.println("Starting
            Application...");
        ResourceConfig rc = new
            ResourceConfig(MovieResource.
            class);
        JdkHttpServerFactory.
            createHttpServer(new URI("http://
            localhost:8080/rest"), rc);
        System.out.println("Application
            started...");
    }
}
```

Listing 2. MainClass: Startklasse der Applikation. Startet HTTP-Server und initialisiert die benötigten REST Resourcen

2.3.1 MovieResource

Die MovieResource, auch REST-Controller genannt, spiegelt die Schnittstelle programmatisch wider. Für alle Funktionalitäten werden hier die Pfade und die entsprechenden HTTP-Methoden festgelegt. Annotationen wie *@GET* oder *@POST* geben den zu verwendenden HTTP-Aufruf vor. Der Pfad setzt sich aus der Server-URL wie in Listing 2 zu sehen, ebenso aus den *@PATH*-Annotationen an der Controller-Klasse sowie den entsprechenden Methoden zusammen. In Listing 3 sind diese Annotationen zu sehen. Um den Controller anzusprechen lautet der Pfad **http://localhost:8080/rest/movies**. Bei der ersten gelisteten *GET*-Methode ist keine Erweiterung des Pfades nötig. Somit führt ein HTTP-GET auf *http://localhost:8080/rest/movie* dazu, dass eine Anfrage nach allen in der Datenbank befindlichen Filmen gestellt wird. Die zweite Methode trägt die Erweiterung *"movieId"*. Hierbei handelt es sich im einen in der URL codierten Parameter. Somit liefert ein HTTP-GET auf *http://localhost:8080/rest/movies/2* den Film mit der ID 2 aus der Datenbank, wie auch in Abbildung 4 zu sehen, beziehungsweise *null* wenn ein Film mit dieser ID nicht vorhanden wäre.

```
import javax.xml.bind.annotation.XmlElement;
import javax.xml.ws.RequestWrapper;
import java.util.List;
@Path( "movies" )
public class MovieResource
{
    private MovieService movieService = new
        MovieServiceImpl();
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response getAllMovies()
```

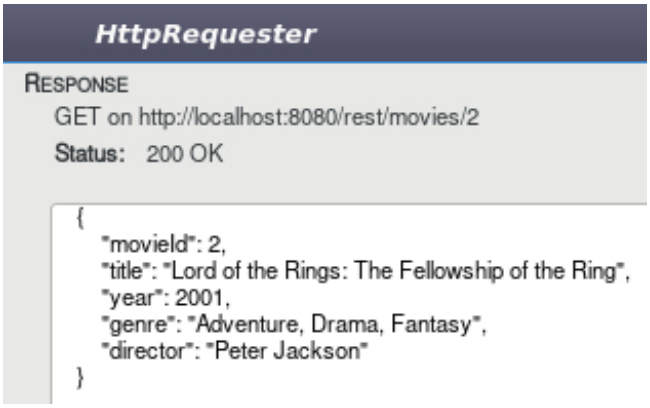

Abbildung 4. Response zum GET-Request auf http://localhost:8080/rest/movies/2

```
    {
        Gson gson = new Gson();
        System.out.println("Begin lookup for
             all movies in database");
        List<Movie> resultList =
            movieService.getAllMovies();
        return Response.ok(gson.toJson(
            resultList)).build();
    }
    @GET
    @Path("{movieId}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getMovieById(@PathParam(
        "movieId") long movieId)
    {
```

Listing 3. Ausschnitt aus der MovieController-Klasse

Als Rückgabewert wird in beiden Fällen eine HTTP-Response geliefert. Diese enthält bei einem GET-Request laut dem Standard(vgl .[9]) den HTTP-Code *200 OK*. Im Body werden die Objekte als Json geliefert. Die Annotation *@Produces* besagt, dass Json als Datenformat verwendet werden soll. Das Gson-Objekt ermöglicht es Java-Objekte in das entsprechende Json-Format zu bringen. Ohne die korrekte Formatierung wäre eine Serialisierung der Objekte nicht möglich und es würde zu Fehlern in der Übertragung kommen. Die Definition der weiteren Methoden können im vollständigen Code des MovieController in Listing 8 gefunden werden.
Somit gilt der Controller nur als Berührungspunkt mit der Außenwelt. Die Funktionalität wird durch den darunter liegenden Service realisiert.

2.3.2 MovieService-Interface

Der MovieService selbst wurde in diesem Beispiel als Interface implementiert. Durch diese Instanz zwischen Controller und Service-Implementierung wird eine gewissen Stabilität gegenüber dem Controller gewährleistet jedoch kann man die verwendete Implementierung beliebig austauschen sofern die im Interface festgelegten Methoden überschrieben werden. Dadurch ergibt sich eine Flexibilität gegenüber der Service-Implementierung.

```
import java.util.List;
public interface MovieService
{
    //retrieves all movies from db
    List<Movie> getAllMovies();
    //retrieves the specified movie from db
    Movie getMovieById(long id);
    //deletes the specified movie from db
    void deleteMovieById(long id);
    //creates a new movie in db
    void createMovie(Movie movie);
}
```

Listing 4. MovieService Interface des Movie-Portfolios

In Listing 4 sind die gewünschten Funktionalitäten abgebildet. Wie es für jede spezielle Anfrage eine exakte URL gibt, wird auch jede Funktionalität mit einer eigenen Implementierung abgedeckt.

2.3.3 MovieService-Implementierung

Hierbei handelt es sich um die konkrete Implementierung der dem Interface zugrunde liegenden Funktionen. Da diese Beispielanwendung sehr klein gehalten ist, ist die einzige Aufgabe das Lesen und Schreiben der Datenbank. Hierzu wird nur die entsprechende Datenbank-Instanz benötigt. Der Programm-Code findet sich in Listing 5.

```
import java.util.List;
public class MovieServiceImpl implements
    MovieService
{
    DBUtil dbUtil = DBUtil.getInstance();
    @Override
    public List<Movie> getAllMovies()
    {
        System.out.println("get for all
            movies is performed");
        //Here happens the Database
            interaction
        return dbUtil.getAllMovies();
    }
    @Override
    public Movie getMovieById(long id)
    {
        System.out.println("get for movie
            with id " + id + " is performed")
            ;
        //Here happens the Database
            interaction
        return dbUtil.getMovieWithId(id);
    }
    @Override
    public void deleteMovieById(long id)
    {
        System.out.println("delete for movie
             with id " + id + " is performed"
            );
```

```
        //Here happens the Database
            interaction
        dbUtil.deleteMovieWithId(id);
    }
    @Override
    public void createMovie(Movie movie)
    {
        System.out.println("new movie with
            id " + movie.getMovieId() + "is
            created");
        //Here happens the Database
            interaction
        dbUtil.createNewMovie(movie);
    }
}
```

Listing 5. MovieService Implementierung des Movie-Portfolios

2.3.4 DBUtil

Wie der Name schon verlauten lässt, handelt es sich hierbei um eine Hilfsklasse. Da keine konkrete Datenbank verwendet wird, ebenso wenig DAO-Klassen, werden die Lese- und Schreibvorgänge in einer Datenbank im Speicher des Rechners durchgeführt. Diese wird mit dem Anlegen einer DBUtil-Instanz angelegt und wird mit Beendigung des Programms zerstört. Um die Applikation funktionstüchtig zu implementieren wurde eine Art von Datenspeicher benötigt. Jedoch handelt es sich hierbei nicht unbedingt um einen relevanten Teil des REST Architekturstils. Die essenziellen Bestandteile können im Listing 9 im Anhang betrachtet werden.

2.4 Clientseite Beispielimplementierung

Grundlegend ist die Realisierung der REST-API durch die Implementierung der Server-Seite abgeschlossen. Eine Benutzung kann über den Browser, Kommandozeilenbefehle wie *curl* oder allen weiteren Programmen, welche es ermöglichen HTTP-Request zu verschicken, stattfinden. Ebenso besteht natürlich die Möglichkeit eine Java-Client programmatisch darzustellen, um innerhalb einer Applikation mit der REST-API zu interagieren. Auch hier bietet das Jersey-Framework wieder die nötigen Bordmittel. Beispielhaft kann eine Clientseitige GET-Anfrage nach einem besagten Film aus unserem Portfolio wie in Listing 6 aufgebaut sein.

```
...
private List<Movie> getMoviesResponse() {
try {
Client client = Client.create();
WebResource webResource = client.resource("
    http://localhost:8080/rest/movies");
ClientResponse response = webResource.accept
    ("application/json").get(ClientResponse.
    class);
if (response2.getStatus() != 200) {
throw new RuntimeException("Failed : HTTP
    error code : " + response.getStatus());
}
List<Movie> output = response.getEntity(new
    GenericType<List<Movie>>(){});
} catch (Exception e) {
e.printStackTrace();
}
}
...
```

Listing 6. Beispiel: Beispielhafte Java-Client Anfrage

Die Klassen Client, WebResource sowie ClientResponse werden alle durch das Jersey-Framework zur Verfügung gestellt. Der Client bekommt die URL für die geforderte Ressource, hier die Liste aller Filme. Durch das tragen der URL wird der Client zur WebResource. Diese bietet die Möglichkeit die entsprechenden Requests zu stellen. Als Antwort wird ausschließlich Json akzeptiert und mit dem folgenden Methodenaufruf *get(ClientResponse.class)* wird der HTTP-Request ausgeführt. Die ClientResponse-Klasse birgt die komplette Antwort, also HTTP-Header und Body. Somit können wie im Beispiel der HTTP-Status auf Richtigkeit geprüft werden bevor die Daten des Bodies verarbeitet werden. Der GenericType wird bei zum Beispiel bei allen Arten von Collections verwendet. Somit wird das erhaltene Json in die benötigte Java-Objekt Konstellation gebracht, hier eine Liste mit den Filmen. Dieser Zusatz ist beim Übertragen einfacher Daten oder einzelner Objekte nicht notwendig. Zu erwähnen ist weiterhin, dass der verwendete Datentyp Movie im Client nicht dem Datentyp vom Server entsprechen muss. Es müssen nicht alle Felder vorhanden sein, ebenso wenig ist der Datentyp auf nur die übertragenen Felder beschränkt. Dies bestätigt auch die *Uniform Interface* Richtlinie von Fielding. Jedoch muss der Bezeichner des Objekt-Attributs den Namen des JSON-Attributs tragen.

3. Zusammenfassung

Beschrieben wurde der REST Architekturstil. Es wurden grundlegenden Eigenschaften nach Fielding erläutert welche ein diese Architekturart mit sich bringen muss um auch als solche bezeichnet und genutzt zu werden. Mit diesen sechs Eigenschaften wurden nicht nur der Großteil der heute existierenden Web-Anwendungen sondern auch das HTTP selbst entwickelt. Es existieren viele Frameworks die es sich zur Aufgabe gemacht haben die JAX-RS API von Java weiter zu entwickeln und die Benutzung zu vereinfachen und zu erweitern. Anhand des Referenz-Frameworks Jersey 2.0 wurde eine beispielhafte Implementierung einer RESTful-API durchgeführt. Nochmals herauszustellen ist die Einfachheit mit welcher solch eine Schnittstelle realisiert werden kann. Hauptsächlich mit Annotationen und den gelieferten Klassen gelingt die Implementierung schnell und effektiv. Ebenso ist deutlich zu erkennen, dass der Aufbau einfach gehalten und leicht nachvollziehbar ist. Es wurden im Film-Portfolio Beispiel nicht alle Aspekte speziell beleuchtet, jedoch sind die essenziellen Notwendigkeiten, wenn auch nicht mit allen Voraussetzungen, wie die Umsetzung des *Uniform Interfaces*

dargestellt. Alles in Allem wird klar wieso sich REST an vorderster Front zur Realisierung von *Web Services Architectures* befindet. Flexibilität, einfache Umsetzung sowie Skalierbarkeit sind nur einige Vorteile die REST für die Kommunikation zwischen Applikationen im WWW mit sich bringt. Diese breite Funktionalität bereitet den Weg für ein verteiltes Entwickeln von Anwendungen ebenso wie ein einfachen Anbinden von freien oder kommerziellen Diensten anderer Anbieter.

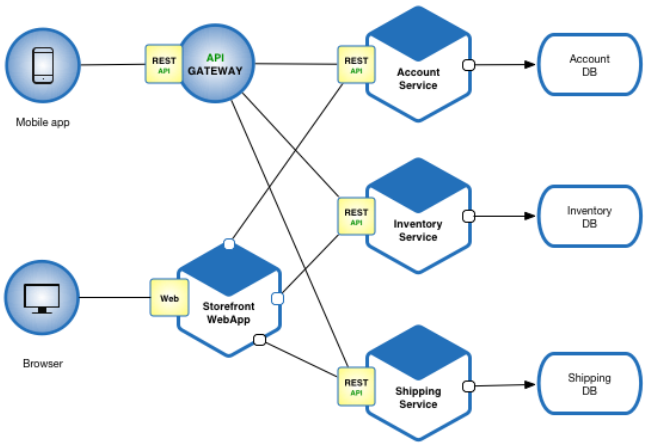

Abbildung 5. Aufbau einer Microservice-Architektur

3.1 Ausblick

Gerade das Nutzen schon vorhandener Dienste birgt ein interessantes Thema. Die Mircoservice-Architektur. Im Zuge dieser wird es sich zum Ziel gemacht, eigenständige Komponenten als unabhängige Services zu implementieren. Somit sollen viele einzelne Dienste entstehen die wiederum in unterschiedlichen Projekten genutzt werden können. Der Datenaustausch findet auch zwischen den einzelnen Services mit REST statt und um die Nutzung eines solchen Dienstes zu beanspruchen muss über die exponierte API kommuniziert werden. Einen kurzen Überblick gibt die Grafik 5. Viele grundlegende Informationen über den Aufbau einer Microservice-Architektur findet man in Quelle [10].

Literatur

[1] W3C Working Group. Web services architecture w3c working group note 11 february 2004. https://www.w3.org/TR/ws-arch/.

[2] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm, 2000.

[3] W3C Working Group. Soap version 1.2 part 1: Messaging framework (second edition). https://www.w3.org/TR/soap12/.

[4] W3C Working Group. Web services description language (wsdl) 1.1. https://www.w3.org/TR/wsdl.

[5] from Sun Microsystems Inc. Marc Hadley, Paul Sandoz. Jax-rs - java™ api for restful web services ("specification"). https://jsr311.java.net/nonav/releases/1.1/spec/spec.html.

[6] Java specification request. https://de.wikipedia.org/wiki/Java_Specification_Request.

[7] Jsr 339: Jax-rs 2.0: The java api for restful web services. https://jcp.org/en/jsr/detail?id=339.

[8] Jersey - restful web services in java. https://jersey.java.net/.

[9] Roy Thomas Fielding with W3C. Rfc 2616, hypertext transfer protocol –http/1.1, section 10, status code definitions. https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

[10] Pattern: Microservice architecture. http://microservices.io/patterns/microservices.html.

Erklärung zur Ausarbeitung

Hiermit erkläre ich, *Vorname Nachname (Matrikel)*, dass ich die vorliegende Ausarbeitung selbstständig und ohne fremde Hilfe angefertigt habe und keine anderen als in der Abhandlung angegebenen Hilfen benutzt habe; dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe. Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

_____

Unterschrift

1. Listings

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/
             maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>RestProject</groupId>
    <artifactId>rest-example</artifactId>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>org.glassfish.jersey.containers</groupId>
            <artifactId>jersey-container-grizzly2-http</artifactId>
            <version>2.25.1</version>
        </dependency>
        <dependency>
            <groupId>org.glassfish.jersey.containers</groupId>
            <artifactId>jersey-container-grizzly2-servlet</artifactId>
            <version>2.25.1</version>
        </dependency>
        <dependency>
            <groupId>org.glassfish.jersey.containers</groupId>
            <artifactId>jersey-container-jdk-http</artifactId>
            <version>2.25.1</version>
        </dependency>
        <dependency>
            <groupId>org.glassfish.jersey.containers</groupId>
            <artifactId>jersey-container-simple-http</artifactId>
            <version>2.25.1</version>
        </dependency>
        <dependency>
            <groupId>org.glassfish.jersey.containers</groupId>
            <artifactId>jersey-container-jetty-http</artifactId>
            <version>2.25.1</version>
        </dependency>
        <dependency>
            <groupId>org.glassfish.jersey.containers</groupId>
            <artifactId>jersey-container-jetty-servlet</artifactId>
            <version>2.25.1</version>
        </dependency>
        <dependency>
            <groupId>com.fasterxml.jackson.jaxrs</groupId>
            <artifactId>jackson-jaxrs-json-provider</artifactId>
            <version>2.9.0.pr1</version>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <version>1.4.193</version>
        </dependency>
        <dependency>
            <groupId>com.google.code.gson</groupId>
            <artifactId>gson</artifactId>
            <version>2.8.0</version>
        </dependency>
    </dependencies>
</project>
```

Listing 7. POM.xml der Beispielanwendung

```
import com.google.gson.Gson;
import javax.ws.rs.*;
import javax.ws.rs.core.GenericEntity;
import javax.ws.rs.core.GenericType;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.ws.RequestWrapper;
import java.util.List;
@Path( "movies" )
public class MovieResource
{
    private MovieService movieService = new MovieServiceImpl();
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Response getAllMovies()
    {
        Gson gson = new Gson();
        System.out.println("Begin lookup for all movies in database");
        List<Movie> resultList = movieService.getAllMovies();
        return Response.ok(gson.toJson(resultList)).build();
    }
    @GET
    @Path("{movieId}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response getMovieById(@PathParam("movieId") long movieId)
    {
        Gson gson = new Gson();
        System.out.println("Begin lookup for movie with id " + movieId + " in database");
        return Response.ok(gson.toJson(movieService.getMovieById(movieId))).build();
    }
    @DELETE
    @Path("{movieId}")
    public Response deleteMovieById(@PathParam("movieId") long movieId)
    {
        System.out.println("Begin lookup for movie with id " + movieId + " in database");
        movieService.deleteMovieById(movieId);
        return Response.ok().build();
    }
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response createMovie(Movie movie)
    {
        System.out.println("Create new movie in database");
        movieService.createMovie(movie);
        return Response.status(Response.Status.CREATED).build();
    }
}
```

Listing 8. Vollständiger Code des MovieControllers

```
public class DBUtil
{
    private static Connection conn;
    private static DBUtil instance;
    private DBUtil()
    {
        try
        {
            System.out.println("Setup DB...");
            DataSource ds = JdbcConnectionPool.create("jdbc:h2:mem:test;DB_CLOSE_DELAY=-1",
                "user", "password");
            conn = ds.getConnection();
            conn.createStatement().executeUpdate("CREATE TABLE movies(id INTEGER(20) PRIMARY
                KEY, title VARCHAR2(50), director VARCHAR2(50), genre VARCHAR2(100), year
                INTEGER(4) )");
            conn.commit();
            createBackToTheFuture();
            createLotR();
            createStarWars();
            createTerminator2();
            createHitchhikersGuide();
            System.out.println("Table created, 5 entries written");
        }
        catch(Exception e)
        {
            e.fillInStackTrace();
        }
    }
   public List<Movie> getAllMovies()
   {
       List<Movie> result = new ArrayList<Movie>();
       try
       {
           ResultSet resultSet = conn.createStatement().executeQuery("SELECT * FROM movies
               ORDER BY id");
           while (resultSet.next())
           {
               Movie m = new Movie();
               m.setMovieId(resultSet.getLong("id"));
               m.setTitle(resultSet.getString("title"));
               m.setDirector(resultSet.getString("director"));
               m.setGenre(resultSet.getString("genre"));
               m.setYear(resultSet.getInt("year"));
               result.add(m);
           }
       }
       catch(SQLException e)
       {
           e.printStackTrace();
       }
       return result;
   }
}
```

Listing 9. Auszüge aus der DBUtil Klasse