Hochschule München

Fakultät für Informatik & Mathematik

Bereitstellen und Konsumieren von REST APIs mit Spring

FWP Aktuelle Technologien zur Entwicklung verteilter Java Anwendungen

Dozent: Herr Michael Theis

Verfasser: Manuel Wolf

Matrikelnummer: 69571816

Semestergruppe: IB6A

Inhaltsverzeichnis

Einleitung 4
Einführung 4
Spring Framework 4
Erklärung REST 4
Client-Server 4
Zustandslosigkeit 5
Caching 5
Einheitliche Schnittstelle 5
Mehrschichtige Systeme 5
Code-On-Demand 5
Http 6
http-Crud-Methoden 6
Statuscodes 6
Aufbau einer http-Nachricht 7
Bereitstellen einer REST API mit Spring 8
Aufsetzen eines Spring Projekts 8
Benötigte Dependencies 8
Erstes Build 9
Spring REST Architektur -MVC 9
Model 9
View 9
Controller 9
Service-Layer für Datenbankzugriff 9
Erstmals Daten von Server erhalten 9
Erstellen des Models 9
Erstellen des Controllers 10
Aufruf der Schnittstelle via Postman 11
Erstellen der Services 12
Informationen aus dem http-Request entnehmen 12
Pfadvariablen 12
Query 12
http-Body 13
Rest-Controller erstellen 13
http-Response bearbeiten 13
Alle Crud-Operationen anbieten 14

Fehlerbehandlungen 14
Manuelle Fehlerbehandlung 14
Exceptionhandler 15
Validierung mit der Java Validation API 16
Konsumieren einer REST API mit Spring 17
RestTemplate 17
Get-Operation 17
Post-Operation 17
Put-Operation 18
Delete-Operation 18
Rückgabe benutzerdefinierter Objekte 18
Abbildungsverzeichnis 19
Literaturverzeichnis 20

Einleitung

In einer Zeit, in der verteilte Anwendungen und Microservice-Architektur immer mehr an Beliebtheit gewinnen, ist es wichtig seine Programme für die Außenwelt zur Verfügung zu stellen. Dies darf jedoch keines Falls beliebig möglich sein. Es muss klar definierte Endpunkte geben, an denen Daten abgefragt werden können. Ebenso werden Anwendungen immer größer und vernetzter. Dabei bestehen Applikationen meist aus einzelnen, abgekapselten Komponenten, welche Daten nach außen weitergeben. Dies geschieht heutzutage in den meisten Fällen über REST Schnittstellen. Diese erlauben einen sehr einfachen und kontrollierten Zugriff auf Daten von externen Anwendungen. Ziel dieser Arbeit ist es, dem Leser, die Grundlagen von REST näher zu bringen. Zunächst werden einige Grundbegriffe und verwendete Frameworks sowie Technologien vorgestellt. Hierbei wird vor allem erläutert, was eine wirkliche Rest Schnittstelle ausmacht. Immer wieder kommt es vor, dass von REST Schnittstellen gesprochen wird, obwohl es sich eigentlich gar nicht darum handelt. Daraufhin wird die Umsetzung für die Bereitstellung einer REST API erklärt. Die theoretischen Erklärungen werden von einer praktischen Demo begleitet. Um immer ein direktes Beispiel zu liefern, welches das Verständnis erhöht, wird zu jedem theoretisch erklärten Schritt, ein Bild des Programmcodes hinzugefügt. Daraufhin wird eine zweite Anwendung realisiert, welche die Clientseite darstellt. Hier wird Schritt für Schritt erklärt, wie REST APIs mit Spring aufgerufen werden können und mit den zurückgegebenen Daten oder http-Informationen richtig umgegangen werden kann. In der Arbeit werden dabei ausschließlich die Grundlagen näher erläutert. Themen wie beispielsweise Security werden außenvor gelassen, da dieses Thema einen sehr großen einzelnen Themenblock darstellen würde. Ebenso beschränken sich die gesendeten Nachrichten auf Json Files. Das Senden, von anderen Dateiformaten oder Mediadaten wird nicht behandelt.

Einführung

Spring Framework

Das Spring Framework erleichtert das Erstellen von komplexen „Java Enterprise Applications". Es setzt auf Java EE auf und versucht die Entwicklung komplexer Anwendungen deutlich zu vereinfachen. Zunächst war das Ziel, einen besseren Umgang mit Enterprise Javabeans (EJB) zu gewährleisten. „EJB ist dabei eine moderne, verteilte Komponentenarchitektur."[1]

Erklärung REST

Bei Representational State Transfer (REST) handelt es sich um einen Architekturstil für verteilte Systeme, bei dem die Kommunikation zwischen den verschiedenen Teilnehmern über HTTP stattfindet. Dabei gibt es klar definierte Zugriffsmöglichkeiten auf die Schnittstelle, welche Daten anbietet. Diese werden über URI oder URLs aufgerufen. Rest folgt dabei sechs Prinzipien, die es einzuhalten gilt, damit von einer REST-Schnittstelle gesprochen werden kann.[2]

Client-Server

Eine Anforderung an REST ist die klare Trennung von Client und Server. Dadurch ist gewährleistet, dass sich Benutzeroberfläche und Datenverarbeitung und Speicherung, also Frontend und Backend, unabhängig voneinander weiter entwickeln können. Ebenso besteht ein Vorteil darin, dass Plattform unabhängig entwickelt werden kann, da verschiedene Benutzeroberflächen zur Verfügung gestellt werden können, welche im Hintergrund alle denselben Server aufrufen.

---

[1] 10.5.1 Enterprise Java Beans (EJB): http://openbook.rheinwerk-verlag.de/it_handbuch/kap_10_konz_prog_005.html#8f42eaaf-0fb8-4830-9b1b-bc0fae3dd304

[2] Representational State Transfer (REST): https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Zustandslosigkeit

Die Zustandslosigkeit beschreibt, dass keine Daten, welche sich auf den Client beziehen oder für die Identifizierung der Anfrage dienen, auf dem Server gespeichert werden. Es wird vorausgesetzt, dass alle nötigen Informationen in jedem einzelnen Request mitgeschickt werden. Somit kann sichergestellt werden, dass jeder Request separat und unabhängig von anderen Requests funktioniert. Ebenso können Fehler leichter nachvollzogen werden, da sichergestellt ist, dass sich die Fehlerquelle auf diesen einen Request beschränkt, da keine Daten, welche möglicherweise einen anderen Ursprung haben, verwendet werden. Nachteil, dieses Architekturstils ist jedoch, dass eine bessere Netzwerk-Performance benötigt wird, da jede Anfrage erneut alle benötigten Daten schicken muss. Somit werden oft sich wiederholende Daten mitgeschickt.

Caching

Um die Netzwerkeffizienz zu verbessern, soll bei REST die Möglichkeit des Cachings bestehen. Somit wird vorausgesetzt, dass bei allen gesendeten explizit angegeben wird, ob es sich um zwischenspeicherbare oder nicht- zwischenspeicherbar Daten handelt.

Einheitliche Schnittstelle

Das zentrale Merkmal, welches REST von anderen netzwerkbasierten Stilen unterscheidet, ist, dass es sich um einheitliche Schnittstellen handelt. Hierbei wird das Prinzip der Allgemeingültigkeit verfolgt. Somit ist sichergestellt, dass ein bestimmter Request verbindlich immer das gleiche Ergebnis zurückliefert. Ebenso ist die Implementierung komplett von dem Service, welcher die Daten zur Verfügung stellt, entkoppelt. Dies unterstützt wieder die unabhängige Weiterentwicklung der verschiedenen Komponenten.

Mehrschichtige Systeme

Eine Anwendung soll aus verschiedenen hierarchischen Schichten zusammengesetzt werden und somit die Skalierbarkeit zu erhöhen, die Architektur zu vereinfachen und um in sich geschlossene Komponenten zu erstellen, welche unabhängig von anderen Schichten funktionieren. Dies erlaubt bei möglichen Änderungsanforderungen, flexibler agieren zu können und Schichten zu ändern oder ganz auszutauschen. Somit wird dem Client nur die nötige Schnittstelle angeboten. Alle dahinter liegenden Schichten bleiben ihm verborgen und diese können nicht separat aufgerufen werden.

Code-On-Demand

Bei Code-On-Demand handelt es sich um ein optionales Prinzip, welches es dem Client erlaubt, für bestimmte Funktionalitäten den Code herunterzuladen und gegebenenfalls selbst zu ändern oder zu erweitern. Somit kann Code auf die Bedürfnisse des Clients angepasst werden, ohne dass der Server alle möglichen Fälle abdecken muss.

Http

Für die Kommunikation zwischen eines Clients und einer REST API werden fast ausschließlich http und https verwendet. Hierfür bietet http einige Methoden, um Daten zu erhalten oder zu manipulieren. Die sogenannten CRUD-Methoden decken dabei die meisten Anwendungsfälle ab.

http-Crud-Methoden

http-Methode	Beschreibung
Get	Fordert Daten vom Server an
Post	Übermittelt Daten an den Server
Put	Ändert bestehende Daten auf dem Server. Put stellt sicher, ob es eine Datei bereits auf dem Server gibt. Ist dies der Fall, wird diese geändert. Ist dies nicht der Fall, dann wird eine neue Datei angelegt (Manchmal wird Patch verwendet)
Delete	Löscht Daten auf dem Server

Statuscodes

Auf eine http-Anfrage erhält der Client eine http-Antwort vom Server. Diese Antwort enthält immer einen http-Statuscode. Mit diesem Statuscode hat der Client die Möglichkeit, Informationen über den Erfolg seiner Anfrage, zu erhalten. Statuscodes sind in fünf Hauptgruppen unterteilt. Kategorisiert werden diese durch die erste Ziffer des Statuscodes.

Statuscode	Bedeutung
1xx	Informationen
2xx	Erfolgreiche Operation
3xx	Umleitung
4xx	Client Fehler
5xx	Server Fehler

Die meistbenutzten Statuscodes sind dabei:[3]

Statuscode	Nachricht	Bedeutung
200	Ok	Die Anfrage wurde erfolgreich bearbeitet
201	Created	Eine Ressource wurde erfolgreich an den Server geschickt und erstellt
203	Created	Das gesendete Objekt wurde erfolgreich erstellt. Meist wird zusätzlich eine Url, unter der das neue Objekt erreichbar ist, mitgeschickt
204	No Content	Die Anfrage wurde erfolgreich bearbeitet, jedoch enthält die Antwort keine Daten. Wird z.B oft bei DELETE verwendet
400	Bad Request	Die Anfrage wurde nicht richtig aufgebaut und dem Server fehlen Daten
401	Unauthorized	Die Anfrage benötigt eine Authentifizierung
403	Forbidden	Der Client hat nicht die nötigen Rechte, um eine Operation durchzuführen
404	Not Found	Die angeforderte Ressource wurde nicht gefunden
500	Internal Server Error	Serverseitig kommt es zu einem Fehler und die Anfrage kann nicht richtig bearbeitet werden

[3] *HTTP Status Codes*: https://www.restapitutorial.com/httpstatuscodes.html

Aufbau einer http-Nachricht

Eine http-Nachricht besteht aus einem http-Header und einem http-Body. Der http-Header erlaubt dem Sender zusätzliche Informationen zu übergeben. Grundsätzlich wird zwischen dem General Header, dem Request Header, dem Response Header und dem Entity Header unterschieden.
Der General Header kann sowohl bei Anfragen als auch bei Antworten mitgesendet werden und bezieht sich nicht auf geschickte Daten. Er enthält allgemeine Werte wie Datum oder Cache-Control mitgeschickt.
Der Request Header wird nur bei Anfragen mitgeschickt und enthält Informationen, wie Authentifizierung, Cookies, erlaubte Formate für die Antwort und generelle Informationen vom Sender.
Der Response Header wird bei Antworten hinzugefügt. Dieser enthält Werte, wie den Servernamen, den Ort, an dem sich der Server befindet oder den Statuscode.
Der Entity Header wird in Anfragen und Antworten benutzt. Dieser beschreibt den Inhalt des Bodys genauer und enthält Werte, wie die Länge des Inhalts, der Typ des Inhalts, wie zum Beispiel JSON oder Xml. Im http-Body befindet sich der eigentliche Inhalt, welcher an den Server geschickt oder von ihm gesendet wird.[4]

[4] HTTP Header: https://developer.mozilla.org/de/docs/Web/HTTP/Headers

Bereitstellen einer REST API mit Spring

Im folgenden Abschnitt wird mit Spring ein Projekt Schritt für Schritt entwickelt, welches eine REST-Schnittstelle zur Verfügung stellt und anschließend von einem Client aufgerufen werden kann. Bei dem Bespielprojekt wird es sich um eine Anwendung handeln, die es erlaubt ein Telefonbuch zu verwalten. Hierfür gibt es verschiedene Einträge, die die nötigsten Informationen zu einer Person enthalten. Gespeichert werden die Daten im Hintergrund in einer lokalen Datenbank. Jedoch wird im Folgenden nicht weiter auf die Datenbank eingegangen, da es sich die Arbeit lediglich mit dem Bereitstellen der REST-Schnittstelle mit Spring beschäftigt.

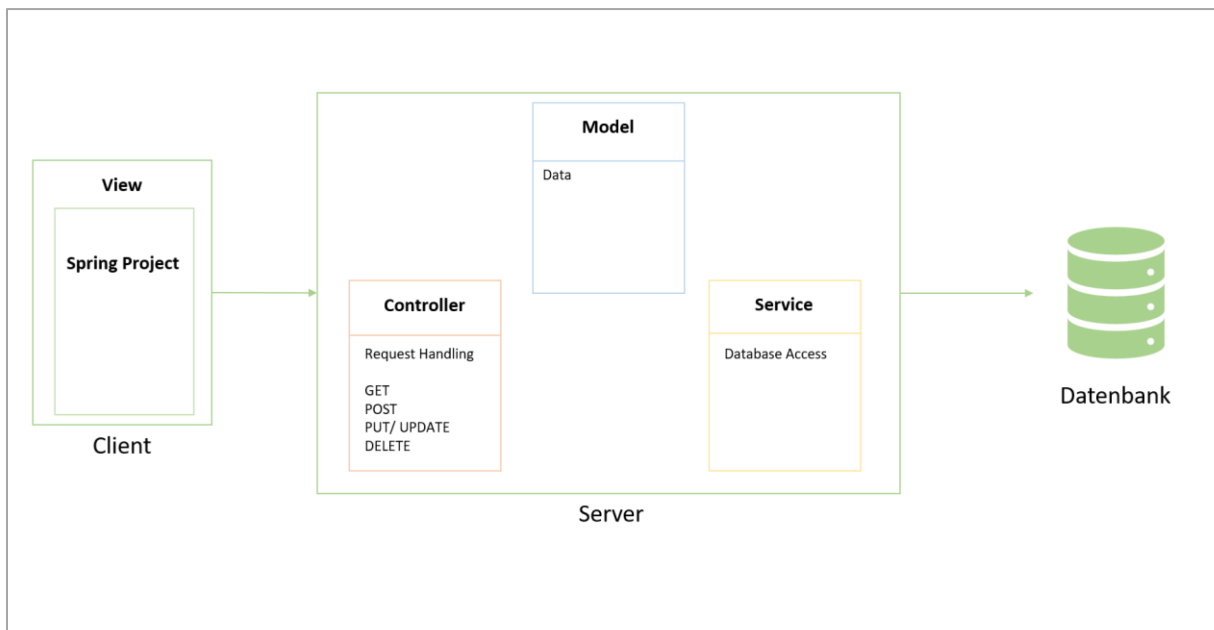

Abbildung 1: Aufbau des Projektes

Aufsetzen eines Spring Projekts

Zum erstellen eines Projektes wird der „Spring Initalizr“[5] verwendet. Bei dem folgenden Projekt handelt es sich um ein Maven-Projekt mit Java, welches sich auf der Website einstellen lässt. Danach wird der Applikation noch eine Gruppe und einen Namen gegeben. Das Ganze ich in ein Jar gebuildet. Nachdem die nötigen Dependencies für die Spring-Applikation ausgewählt wurden, kann das Projekt erstellt und heruntergeladen werden.

Benötigte Dependencies

Spring Web

Die einzige Dependency, die für das Bereitstellen einer Rest Schnittstelle benötigt wird, ist die „Web“-Dependency. Damit ist es möglich per Annotationen Rest-Services bereitzustellen.

Dependencies für Datenbankzugriff

Damit Daten, welche später an den Server gesendet werden, nicht verloren gehen, wird zudem die „JPA“-Dependency verwendet. Ebenso wird der „MariaDb-Java-Client“ verwendet, um Zugriff auf die Datenbank zu bekommen. Auf die genaue Funktionsweise wird im Folgenden nicht eingegangen, da sich die Arbeit lediglich mit dem Bereitstellen der REST API beschäftigt und nicht mit der Geschäftslogik oder dem Datenbankzugriff, welche im Hintergrund stattfinden.

[5] https://start.spring.io/

Erstes Build

Wenn das Projekt heruntergeladen wurde und es in einer beliebigen Ide geöffnet wurde, kann der Server mit dem Befehl „mvn spring-boot:run“ gestartet werden. Standardmäßig ist der Server nun unter „localhost:8080“ erreichbar. Im Moment wird jedoch noch eine Fehlerseite angezeigt, da die Url nicht gefunden wurde, da noch nichts weiter erstellt wurde, aber der Code kompiliert erfolgreich und der Server läuft.

Spring REST Architektur -MVC

Bei dem Projekt handelt es sich um eine Model-View-Controller-Architektur.

Model

Es wird ein „model“-Package erstellt, in dem alle Models erstellt werden. In diesem Package werden die Entitys, mit denen gearbeitet wird und welche in der Datenbank gespeichert werden, definiert.

View

Die View-Komponente dieses Projekts ist nicht in dem Projekt selbst. Da es REST eine klare Trennung der View, also des Clients und des Servers geben soll. Die Applikation, welche die View-Komponente darstellt, ist jene Applikation, welche die REST-Schnittstelle am Ende konsumiert. Wie diese erstellt wird, wird in Abschnitt „Konsumieren einer REST API mit Spring“ erklärt.

Controller

Für die Controller-Ebene wird ebenso ein eigenes „controller“-Package, in dem sich alle Controller der Anwendung befinden, erstellt. Die Controllerklasse beinhaltet die Methoden, welche durch externe Anwendungen, mit Hilfe der bestimmten URIs aufgerufen werden können.

Service-Layer für Datenbankzugriff

Da die Controller-Ebene, welche letztendlich die einzige Ebene ist, welche für den Client sichtbar ist, stets von der Geschäftslogik getrennt sein sollte, wird eine Service-Ebene eingeführt, welche die Geschäftslogik und in Fall, dieses Projektes auch den Datenbankzugriff enthält. In großen Anwendungen können diese zwei Ebenen erneut aufgeteilt und die Logik- bzw. Service-Ebene von der Data-Access-Ebene getrennt werden. Um die Modularität und die Möglichkeiten der Anpassungen des Service-Layers zu bieten, werden stets Interfaces benutzt. So kann etwas separat auf der Service-Ebene geändert werden, ohne den Aufruf im Controller oder in anderen Services ändern zu müssen. Da in dieser Anwendung Spring JPA verwendet wird und es keiner weiteren Logik bedarf, als die Datenbankzugriffe, entsprechen die Service-Klassen gleich den Data-Access-Klassen

Erstmals Daten von Server erhalten

Erstellen des Models

Um Daten vom Server zu erhalten oder zu manipulieren, müssen zuerst die Models, also die Daten, welche gespeichert bzw. dem Client zur Verfügung gestellt werden, definiert werden. Da es sich bei diesem Beispielprojekt um ein Telefonbuch handelt, wird ein Telefonbucheintrag zurückgegeben. Dieser enthält eine ID, Vor- und Nachname sowie die Telefonnummer. Dafür wird im model-Package eine neue Klasse „PhonebookEntry“ erstellt.

```
public class PhonebookEntry {

    int id;
    String Prename;
    String Name;
    String PhoneNumber;


    public PhonebookEntry() {
    }

    public PhonebookEntry(int id, String prename, String name, String phoneNumber) {
        this.id = id;
        Prename = prename;
        Name = name;
        PhoneNumber = phoneNumber;
    }
```

Abbildung 2: Telefonbucheintag des Beispielprojektes

Erstellen des Controllers

Um einen solchen Telefoneintrag über eine Schnittstelle zur Verfügung zu stellen wird eine Controller-Klasse erstellt. In diesem Fall der „PhonebookController“. Damit dieser auch als Controller definiert ist und von externen Anwendungen aufgerufen werden kann, werden Annotationen benötigt. Die Controller-Klasse wird mit der „@RestController“-Annotation versehen. Somit weiß Spring, dass es sich um einen Restcontroller handelt. Zu diesem Zeitpunkt wäre der Controller bereits unter der Adresse *„http://localhost:8080/“* erreichbar. Jedoch ist dieser Name nicht sehr gut gewählt, da keinerlei Informationen, ob es sich bei dieser URL um eine Schnittstelle handelt, noch welche Daten von dieser abgefragt werden können, zur Verfügung gestellt werden. Spätestens, wenn zwei Controller existieren, kommt es zu Komplikationen, da mehrere Controller unter der gleichen URL verfügbar wären. Um jedem Controller eine eindeutige und sprechende URL zu geben, wird jedem Controller mit Hilfe der „@RequestMapping“-Annotation eine URL zugewiesen. Häufig beginnt die URL mit dem Wort „api“, damit der Aufrufer weiß, dass es sich bei dieser Adresse um eine Schnittstelle handelt. Darauf folgt oft die Versionsnummer der Schnittstelle. Daraufhin beginnt der eigentliche Name, um diesen Controller zu spezifizieren. In dem Fall des PhonebookControllers wäre daher eine Annotation wie *„@RequestMapping("api/v1/phonebook")“* sinnvoll. Nun ist der Controller unter *„http://localhost:8080/api/v1/phonebook“* erreichbar.

Jetzt müssen Methoden definiert werden, die der Client aufrufen kann. Hier wird eine ganz normale Java-Methode benötigt mit der Besonderheit, dass diese mit einer „@GetMapping()“ Annotation erweitert wird. Innerhalb der Klammern, kann definiert werden unter welcher URL diese spezifische Methode erreichbar ist. Dies ist hilfreich, wenn mehrere Get-Methoden innerhalb eines Controllers angeboten werden. Im Beispiel wird zunächst eine Methode unter der URL „test“ angeboten. Zu Testzwecken ist der Rückgabewert der Methode zunächst das bereits erstellte Model „PhonebookEntry“. Dies ist normal nicht der Fall und wird später mit der Rückgabe einer http-Response ersetzt, was es erlaubt, dem Client genauere Informationen für die http-Kommunikation zukommen zu lassen.

```java
package com.spring.restApi.SpringRestApi.controller;

import com.spring.restApi.SpringRestApi.model.PhonebookEntry;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("api/v1/phonebook")
public class PhonebookController {


    @GetMapping("test")
    public PhonebookEntry getPhoneNumber(){
        return new PhonebookEntry( id: 1,  prename: "Herbert",  name: "Bauer",  phoneNumber: "016478875221");

    }
}
```

Abbildung 3: PhonebookController erster Test

Aufruf der Schnittstelle via Postman

Nun kann die Applikation gestartet werden und ist aufrufbar. Um dies zu testen, gibt es Tools wie Postman[6] oder Insomnia[7]. In diesem Beispiel wird Postman verwendet. In Postman können verschiedene http-Requests an einen Server geschickt werden. Hierfür kann eingestellt werden, um welche Art von Request (zum Beispiel Get oder Post) es sich handelt und gegebenenfalls den http-Header oder http-Body nach Belieben definiert werden. Um die Methode aus dem PhonebookController aufzurufen, wird mit Postman ein Get-Request an die URL *„http://localhost:8080/api/v1/phonebook/test“* geschickt.

http://localhost:8080/api/v1/phonebook/test

GET | http://localhost:8080/api/v1/phonebook/test | Send | Save

Params | Authorization | Headers (7) | Body | Pre-request Script | Tests | Cookies | Code | Comments (0)

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body | Cookies | Headers (3) | Test Results | Status: 200 OK | Time: 14 ms | Size: 202 B | Download

Pretty | Raw | Preview | JSON


```json
{
    "id": 1,
    "name": "Bauer",
    "phoneNumber": "016478875221",
    "prename": "Herbert"
}
```


Abbildung 4: Ergebnis des ersten Get-Requests

Vom Server wird daraufhin eine http-Response geschickt, welche im Body den Telefonbucheintrag, welcher in der Applikation erstellt wurde, enthält. Des Weiteren wird der Statuscode 200 Ok angezeigt, was bedeutet, dass alles funktioniert.

[6] https://www.getpostman.com/downloads/
[7] https://insomnia.rest/download/

Erstellen der Services

Damit die Daten nicht jedes Mal neu erstellt werden müssen, läuft im Hintergrund eine lokale MariaDB-Datenbank. Dafür wird Spring JPA verwendet. Um diese Dependency zu benutzen, wird ein Interface benötigt, welches von einem CrudRepository erbt. Dies erlaubt einfache Datenbankzugriffe und alle nötigen Crud-Operationen lassen sich ausführen.

```
public interface PhonebookService extends CrudRepository<PhonebookEntry, Integer> {

}
```

Abbildung 5: PhonebookService-Interface

Damit die PhonebookEntrys in der Datenbank gespeichert werden, müssen diese, mit einer „@Entity"-Annotation, versehen werden. Des Weiteren wird eine ID definiert.

Hinweis: Groß- und Kleinschreibung der Variablennamen wird beachtet. Werden diese groß geschrieben, findet Spring JPA diese nicht automatisch.

```
@Entity
public class PhonebookEntry {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;
    private String prename;
    private String name;
    private String phoneNumber;
```

Abbildung 6: PhonebookEntry Entity

Informationen aus dem http-Request entnehmen

Pfadvariablen

Eine Möglichkeit, um Informationen aus dem http-Request zu erhalten, sind die sogenannten Pfadvariablen. Diese werden direkt in der URL angegeben. Ein Beispiel dafür wäre die Abfrage, eines ganz bestimmten Objektes, mit Hilfe der ID. Im hier genannten Beispiel wäre eine mögliche Abfrage *„http://localhost:8080/api/v1/phonebook/1"*. Um nun Zugriff zu diesem Wert zu bekommen, muss die „@GetMapping"-Annotiation angepasst werden. Hierfür werden die nötigen Variablen in geschweifte Klammern der Url hinzugefügt. Um diese nun zu verwenden, wird in den Parametern der Methode ebenso eine Variable definiert. Dieser wird eine „@PathVariable" Annotation hinzugefügt.

```
@GetMapping("/{id}")
public PhonebookEntry getPhonebookEntryById(@PathVariable Integer id){
    return phonebookService.findById(id).get();
}
```

Abbildung 7: Get-Methode mit Pfadvariable

Query

Eine weitere Möglichkeit Daten aus dem Request zu erhalten, ist aus dem Query String, welcher an die Url angehängt wird. Getrennt wird dieser String von der eigentlichen URL mit einem „?". Daraufhin folgen die Variablennamen mit den entsprechenden Werten, sogenannte Key-Value-Pairs. Ein möglicher http-Request wäre *„http://localhost:8080/api/v1/phonebook?vorname=Kai&name=Vogt"*. Um einen solchen Aufruf zu ermöglichen, müssen der Get-Methode alle nötigen Parameter übergeben werden und mit einer „@RequestParam"-Annotation versehen werden. Wird ein Parameter mit einer

solchen Annotation deklariert, ist dieser automatisch verpflichtend, bei jedem Aufruf anzugeben. Falls dies nicht gewünscht ist, kann der Annotation ein Wert „required = false“ hinzugefügt werden. Standardmäßig gleicht der Name des Parameters, dem Namen des Schlüssels in der Query. Wenn dieser jedoch einen benutzerdefinierten Namen erhalten soll, so kann der Annotation ein „name“-Attribut hinzugefügt werden.

```
@GetMapping()
public Iterable<PhonebookEntry> getPhonebookEntries(@RequestParam(name = "vorname", required = false) String prename){

}
```

Abbildung 8:Get-Methode mit Queryvariablen

http-Body

Daten können ebenso dem Body entnommen werden. Diese Methode wird jedoch nicht bei Get-Operationen verwendet. Bei Post- oder Put-Operationen hingegen, können dort Objekte übergeben werden, die dann verarbeitet werden können. Dieser Body kann verschiedene Daten enthalten. Für die Kommunikation, zwischen Client und Server, ist heutzutage JSON am üblichsten. Es können jedoch auch XML, Html, Bilder und weiteres im Body mitgeschickt werden. Um die Daten aus dem Body verwenden zu können, wird ein Parameter angegeben, welcher mit eine „@RequestBody“-Annotation deklariert wird. Dieser Parameter kann nun deutlich komplexer sein, als ein String oder ein Integer aus der Query oder den Pfadvariablen. Es können ebenso Objekte sein, die mit dem JSON-File aus dem Body gemappt werden.

```
@PostMapping()
public PhonebookEntry createPhonebookEntry(@RequestBody PhonebookEntry entry){
    return phonebookService.save(entry);
}
```

Abbildung 9:Daten aus dem http-Body entnehmen

Rest-Controller erstellen

Zunächst wird der zuvor erstellte Phonebook-Service per Dependency Injection im Controller hinzugefügt. Dafür wird die Deklaration des Services mit einer „@Autowired“-Annotation versehen.

```
@Autowired
PhonebookService phonebookService;
```

Abbildung 10: Service per Dependency Injection bereitstellen

http-Response bearbeiten

Zurzeit sind die Rückgabewerte der Methoden immer das Objekt PhonebookEntry. Dieses Objekt wird automatisch in den Body der http-Response geschrieben. Es ist jedoch nicht möglich, die http-Response weiter anzupassen. Beispielsweise Statuscodes, die später noch bei der Fehlerbehandlung wichtig werden oder Werte, welche als Information im Header hinzugefügt werden, können nicht hinzugefügt werden. Um dies zu ermöglichen, wird der Rückgabewert in eine „ResponseEntity“ geändert. Mit dieser Entity kann der http-header oder der Statuscode nach Belieben angepasst werden.

```java
@GetMapping("/{id}")
public ResponseEntity<PhonebookEntry> getPhonebookEntryById(@PathVariable Integer id){
    HttpHeaders headers = new HttpHeaders();
    headers.add( headerName: "Custom-Header",  headerValue: "Test-Value");

    ResponseEntity response = new ResponseEntity(phonebookService.findById(id).get(), headers, HttpStatus.valueOf(200));
    return response;
}
```

Abbildung 11: http-Response mit ResponseEntity

Alle Crud-Operationen anbieten

Um die verschiedenen Crud-Operationen anbieten zu können, müssen die Methoden mit den entsprechenden Annotationen versehen werden.

- @GetMapping()
- @PostMapping()
- @PutMapping()
- @DeleteMapping()

```java
@GetMapping("/{id}")
public ResponseEntity<PhonebookEntry> getPhonebookEntryById(@PathVariable Integer id){
    ResponseEntity response = new ResponseEntity(phonebookService.findById(id).get(), HttpStatus.valueOf(200));
    return response;
}

@PostMapping()
public ResponseEntity<PhonebookEntry> createPhonebookEntry(@RequestBody PhonebookEntry entry){
    return ResponseEntity.ok(phonebookService.save(entry));
}

@PutMapping()
public ResponseEntity<PhonebookEntry> updatePhonebookEntry(@RequestBody PhonebookEntry entry){
    return ResponseEntity.ok(phonebookService.save(entry));
}

@DeleteMapping("/{id}")
public ResponseEntity<Void> updatePhonebookEntry(@PathVariable Integer id){
    phonebookService.deleteById(id);
    return new ResponseEntity<Void>(HttpStatus.valueOf(204));
}
```

Abbildung 12: Crud-Operationen

Fehlerbehandlungen

Da REST APIs meist für fremde Clients angeboten werden, die, bis auf die Schnittstellenendpunkte, nichts über den Aufbau der Schnittstelle wissen, ist es wichtig dem Aufrufer möglichst viele nützliche Informationen mitzugeben, damit dieser seine Anfragen richtig aufbaut und benutzt. Hierbei hilft die Rückgabe entsprechender Fehlercodes. Hierbei ist es wichtig die Anfrage zunächst zu validieren und bei falscher Anfrage, den Aufrufer darauf hinzuweisen, dass die http-Anfrage nicht komplett oder falsch aufgebaut ist. Dazu zählen Syntaxfehler, Werte außerhalb des zulässigen Bereichs, oder fehlende Werte, die benötigt werden, wie zum Beispiel ein Authentifizierungstoken.

Ebenso ist es wichtig Exceptions zu behandeln. Kommt es im Programmablauf zu Fehlern, ist es wichtig diesen zu Klassifizieren und dem Client mitzuteilen.

Manuelle Fehlerbehandlung

Im folgenden Beispiel werden die Fehler behandelt, die auftreten können, wenn man, mit Hilfe einer ID, ein bestimmtes Objekt vom Server aus einer Datenbank abfragen möchte. Hierbei funktioniert der Aufruf wie bisher mit einer Pfadvariable.

Zunächst muss der Aufruf validiert werden. Bei der ID in der Datenbank handelt es sich um einen positiven Integerwert. Somit können negative Zahlen von vornherein ausgeschlossen werden. Ebenso muss es sich um eine Zahl und keine Zeichenkette oder anderes handeln. Dies wird in diesem Fall automatisch erkannt, da ein Fehler zurückgeschickt wird, dass ein String nicht in einen Integer konvertiert werden kann. Falls es sich um einen nicht zugelassenen Wert handelt, wird eine http-Response mit dem Fehlercode 400 für Bad Request gesendet.

Ebenso kann es dazu kommen, dass der Client ein Objekt abfragen möchte, welche sich nicht in der Datenbank befindet. Dafür wird für gewöhnlich ein 404 Not Found Statuscode zurückgesendet. Jedoch wird im Moment eine Exception geworfen, wenn es ein Objekt nicht in der Datenbank gibt. Somit endet der Aufruf mit einem Fehler und ein Statuscode 500 Internal Server Error wird zurückgegeben. Um dies zu behandeln müssen die geworfenen Exceptions gefangen und behandelt werden.

```
@GetMapping("/{id}")
public ResponseEntity<PhonebookEntry> getPhonebookEntryById(@PathVariable Integer id){
    if(id <= 0 || id == null)
    {
        return ResponseEntity.badRequest().build();
    }
    try
    {
        return ResponseEntity.ok(phonebookService.findById(id).get());
    }
    catch (NoSuchElementException e)
    {
        return ResponseEntity.notFound().build();
    }
}
```

Abbildung 13: Fehlerbehandlung

Exceptionhandler

Falls es zu komplexeren Operationen im Programm kommt, kann es zu sehr vielen verschiedenen Fehlern kommen, die alle einzeln behandelt werden müssen. Dadurch kann die eigentliche Get-Methode sehr unübersichtlich werden. Um dies zu vermeiden, kann die Behandlung von Fehlern ausgelagert werden. Entweder in eine eigene Methode oder in eine separate Klasse, welche global verfügbar ist. In diesem Beispiel reicht eine Auslagerung in eine Methode innerhalb des Controllers aus.

Als Beispiel wird die Post-Methode verwendet. Im Moment ist es möglich, leere Telefoneinträge zu erstellen oder Werte freizulassen. Um dies zu vermeiden, wird erst geprüft, ob alle benötigen Werte vorhanden sind, um danach den Eintrag in die Datenbank aufzunehmen. Ist dies nicht der Fall wird eine Exception geschmissen. In diesem Fall eine „ValidationException“. Um diese Exception zu behandeln wird eine neue Methode innerhalb des Controllers angelegt. Diese Methode wird mit einer „@ExceptionHandler“-Anntotation versehen und es wird festgelegt, auf welche Exceptions diese Methode reagieren soll. Innerhalb dieser Methode wird dann die http-Response mit dem Fehler zurückgeschickt.

```java
@PostMapping()
public ResponseEntity<PhonebookEntry> createPhonebookEntry(@RequestBody PhonebookEntry entry) throws ValidatorException {
    if(entry.getId() == 0 && entry.getName() != null && entry.getPrename() != null && entry.getPhoneNumber() != null)
        return ResponseEntity.ok(phonebookService.save(entry));
    else {
        throw new ValidatorException("Entry cannot be created");
    }
}

@ExceptionHandler(ValidatorException.class)
ResponseEntity<String> exeptionHandler(ValidatorException e){
    return new ResponseEntity(e.getMessage(), HttpStatus.BAD_REQUEST);
}
```

Abbildung 14: Ausgelagerte Fehlerbehandlung

Validierung mit der Java Validation API

Bei dieser At der Validierung wird direkt in der Modelklasse festgelegt welche Werte zugelassen sind. Die Annotationen werden über das gewünschte Attribut geschrieben.

Liste der möglichen Validierungen

- @NotNull
- @AssertTrue
- @Min / @Max
- @Digits
- @NotEmpty
- @NotBlank
- @Positive
- @Size

Nun kann die Validierung, welche unter Umständen eine lange Liste an If-Bedingungen wäre, weggelassen werden und der Parameter, welcher aus dem Body entnommen wird, wird mit @Valid versehen.
Daraufhin wird eine Exceptionhandler Methode angeboten, die alle Validierungsfehler an den Client zurücksendet. Hierfür wird eine eigene „FieldErrorMessage“-Klasse erstellt, welche die Attribute „Field“ und „Message“ besitzt. Die Klasse gibt an, welches Attribut betroffen ist und welche Voraussetzungen für dieses Feld erfüllt werden müssen. Die Werte werden aus der Exception genommen und auf die „FieldErrorMessage“ gemappt.

```java
@PostMapping("/valid")
public ResponseEntity<PhonebookEntry> createPhonebookEntryJavaValidation(@Valid @RequestBody PhonebookEntry entry) throws ValidatorException {
        return ResponseEntity.ok(phonebookService.save(entry));
}

@ResponseStatus(HttpStatus.BAD_REQUEST)
@ExceptionHandler(MethodArgumentNotValidException.class)
List<FieldErrorMessage>ValidExcetionHandler(MethodArgumentNotValidException e){
    List<FieldError> fieldErrors = e.getBindingResult().getFieldErrors();
    List<FieldErrorMessage> fieldErrorMessages = fieldErrors.stream()
            .map(fieldError -> new FieldErrorMessage(fieldError.getField(), fieldError.getDefaultMessage()))
            .collect(Collectors.toList());
    return  fieldErrorMessages;
}
```

Abbildung 15: Java Validation API

Konsumieren einer REST API mit Spring

Bis jetzt ersetzte Postman die Clientseite der Client-Server-Kommunikation. Im Folgenden Abschnitt wird ein Spring-Java-Projekt entwickelt, welches den Client darstellt und die zuvor erstellte REST Schnittstelle aufruft und Daten vom Server erhält und manipuliert. Bei dem Beispielprojekt ist nach einer MVC-Architektur aufgebaut. Auf die Komponenten Controller und die View, wird im Folgenden nicht eingegangen, da sie für das Abschicken und Empfangen von http-Nachrichten irrelevant sind. Es wird sich lediglich auf eine Serviceklasse beschränkt, in denen alle 4 Crud-Operationen durchgeführt werden. Um dies zu realisieren, wird erneut die Spring Boot Web Dependency benötigt.

RestTemplate

Beim RestTemplate handelt es sich um eine Klasse aus der Web Dependency. Diese Klasse ermöglicht es http-Anfragen zu senden und Antworten direkt zu empfangen.

Get-Operation

Um eine Get-Operation durchzuführen, wird die Methode „getForObject“ des RestTemplates aufgerufen. Als Parameter werden die Url und der erwartete Rückgabetyp zurückgegeben. Um besser mit den empfangenen Daten arbeiten zu können, wird eine Model Klasse erstellt, die der Modelklasse der Schnittstelle gleicht. Sollten die Werte, aus dem Body der http-Response denen, der Modelklasse gleichen, so werden die Werte automatisch auf die Attribute gemappt.

```java
public PhonebookEntry getEntryById(int id)
{
    final String uri = "http://localhost:8081/api/v1/phonebook/" + id;
    RestTemplate restTemplate = new RestTemplate();

    PhonebookEntry result = restTemplate.getForObject(uri, PhonebookEntry.class);
    return result;
}
```

Abbildung 16: RestTemplate – GetObject

Post-Operation

Um eine Post-Operation durchzuführen, kann die Methode „postForObject“ aufgerufen. Hierbei wird zusätzlich, zu zur Url und dem Rückgabewert, das zu sendende Objekt als Parameter übergeben. Als Rückgabewert wird wieder das gesendete Objekt zurückgegeben.

```java
public PhonebookEntry createEntry(PhonebookEntry entry)
{
    final String uri = "http://localhost:8081/api/v1/phonebook";

    RestTemplate restTemplate = new RestTemplate();
    PhonebookEntry result = restTemplate.postForObject( uri, entry, PhonebookEntry.class);
    return result;
}
```

Abbildung 17: RestTemplate - PostObject

Eine weitere Möglichkeit, ist das Aufrufen der Methode „postForLocation“. Hierbei wird nicht das Objekt zurückgegeben, sondern die Url, unter der das Objekt ab sofort bei der Schnittstelle abgefragt werden kann.

Put-Operation

Bei der Put-Operation handelt es sich um eine Methode ohne Rückgabewert. Hierfür reicht es die Methode „put“ mit der Url und dem zu änderndem Objekt anzugeben.

```
public void changeEntry(PhonebookEntry entry)
{
    final String uri = "http://localhost:8081/api/v1/phonebook";
    RestTemplate restTemplate = new RestTemplate();
    restTemplate.put( uri, entry);
}
```

Abbildung 18: RestTemplate - Put

Delete-Operation

Die Delete-Operation wird durchgeführt, indem die „delete“ Methode mit der Url als Parameter aufgerufen wird. Diese besitzt ebenfalls keinen Rückgabewert.

```
public void deleteEntry(int id)
{
    final String uri = "http://localhost:8081/api/v1/phonebook/"+id;
    RestTemplate restTemplate = new RestTemplate();
    restTemplate.delete(uri);

}
```

Abbildung 19: RestTemplate - Delete

Rückgabe benutzerdefinierter Objekte

Mit den zuvor beschriebenen Möglichkeiten ist es lediglich möglich zuvor definierte Objekte zu erhalten. Desweitern wird das Objekt, welches als Rückgabewert angegeben wurde, direkt zurückgegeben. Wird nun beispielsweise eine Liste von Objekten erwartet, dann muss ein anderer Aufruf genutzt werden. Dafür wird die Exchange-Methode des RestTemplates aufgerufen. Hier gibt es die Möglichkeit, mit Hilfe der „ParameterizedTypeReference<>“ Klasse, benutzerdefinierte Rückgabewerte zu erstellen. Hierfür wird der Rückgabewert als ein neues Objekt der Klasse definiert. Diese hat als Rückgabewert ein Objekt der Klasse „ResponseEntity“. Mit dieser ist nun der Zugriff auf die ganze http-Response gewährleistet. Beispielsweise kann so zunächst der Statuscode überprüft werden und nur wenn dieser dem erwarteten Wert entspricht, der normale Programmverlauf fortgeführt werden.

```
public List<PhonebookEntry> getAllEntries()
{
    List<PhonebookEntry> result = null;

    final String uri = "http://localhost:8081/api/v1/phonebook";
    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<List<PhonebookEntry>> response = restTemplate.exchange(
            uri,
            HttpMethod.GET,
            requestEntity: null,
            new ParameterizedTypeReference<List<PhonebookEntry>>(){});

    if(response.getStatusCode().value() == 200)
        result = response.getBody();
    return result;
}
```

Abbildung 20: RestTemplate - Exchange-Methode

Abbildungsverzeichnis

Abbildung 1: Aufbau des Projektes ... 8
Abbildung 2: Telefonbucheintag des Beispielprojektes ... 10
Abbildung 3: PhonebookController erster Test ... 11
Abbildung 4: Ergebnis des ersten Get-Requests ... 11
Abbildung 5: PhonebookService-Interface ... 12
Abbildung 6: PhonebookEntry Entity ... 12
Abbildung 7: Get-Methode mit Pfadvariable ... 12
Abbildung 8:Get-Methode mit Queryvariablen ... 13
Abbildung 9:Daten aus dem http-Body entnehmen ... 13
Abbildung 10: Service per Dependency Injection bereitstellen ... 13
Abbildung 11: http-Response mit ResponseEntity ... 14
Abbildung 12: Crud-Operationen ... 14
Abbildung 13: Fehlerbehandlung ... 15
Abbildung 14: Ausgelagerte Fehlerbehandlung ... 16
Abbildung 15: Java Validation API ... 16
Abbildung 16: RestTemplate – GetObject ... 17
Abbildung 17: RestTemplate - PostObject ... 17
Abbildung 18: RestTemplate - Put ... 18
Abbildung 19: RestTemplate - Delete ... 18
Abbildung 20: RestTemplate - Exchange-Methode ... 18

Literaturverzeichnis

Building a RESTful Web Service. (kein Datum). Von Spring.io: https://spring.io/guides/gs/rest-service/ abgerufen

Fejér, A. (07. 04 2019). *Using Spring ResponseEntity to Manipulate the HTTP Response*. Von Baeldung: https://www.baeldung.com/spring-response-entity abgerufen

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures.* Von ics.uci.edu: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm abgerufen

Gupta, L. (kein Datum). *Spring RestTemplate – Spring REST Client Example*. Von Howtodoinjava: https://howtodoinjava.com/spring-restful/spring-restful-client-resttemplate-example/ abgerufen

Hansen, B. (09. 12 2016). *Pluralsight*. Von Spring Fundamentals: https://app.pluralsight.com/library/courses/spring-fundamentals/table-of-contents abgerufen

HTTP Status Codes. (kein Datum). Von restapitutorial: https://www.restapitutorial.com/httpstatuscodes.html abgerufen

Kersken, S. (28. 08 2011). *Openbook Rheinwerk-verlag.* Von IT-Handbuch für Fachinformatiker: http://openbook.rheinwerk-verlag.de/it_handbuch/kap_10_konz_prog_005.html#8f42eaaf-0fb8-4830-9b1b-bc0fae3dd304 abgerufen

Krischan, S. (25. 01 2019). Von developer.mozilla: https://developer.mozilla.org/de/docs/Web/HTTP abgerufen

Pratt, M. (08. 11 2018). *Get and Post Lists of Objects with RestTemplate*. Von https://www.baeldung.com/spring-rest-template-list: https://www.baeldung.com/spring-rest-template-list abgerufen

Spring Framework Overview. (31. 03 2019). Von docs.spring.io: https://docs.spring.io/spring/docs/5.1.6.RELEASE/spring-framework-reference/overview.html#overview abgerufen

Tilkov, S. (12. 3 2009). *REST – Der bessere Web Service?* Von http://jaxenter.de: http://jaxenter.de/rest-der-bessere-web-service-8988 abgerufen

Van rijn, P. (10. 10 2018). *Spring REST: Getting Started*. Von Pluralsighgt: https://app.pluralsight.com/library/courses/spring-rest/table-of-contents abgerufen

Webb, P., Syer, D., & Long, J. (kein Datum). *Spring Boot Reference Guide*. Von docs.spring.io: https://docs.spring.io/spring-boot/docs/2.1.3.RELEASE/reference/htmlsingle/ abgerufen