Bachelorarbeit

Analyse des Spring-Frameworks Hateoas und Entwicklung einer Beispielapplikation

Heese, Andreas

geboren am 16. November 1995 in Lichtenstein

Matrikelnummer: 34397

Studiengang: Gesundheitsinformatik

Fakultät: Physikalische Technik/Informatik

Fachgruppe: Informatik

Betreuer: Prof. Dr. rer. nat. Wolfgang Golubski
Westsächsische Hochschule Zwickau
Dr.-Friedrichs-Ring 2A
08056 Zwickau, Deutschland

Abgabetermin: 27.04.2022

Inhaltsverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
HATEOAS	Hypermedia as the Engine of Application State
HTTP	Hypertext Transfer Protocol
JRE	Java Runtime Environment
JSON	JavaScript Object Notation
PIM	Product Information Management
POX	Plain Old XML
REST	Representational State Transfer
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language

Einleitung

Gegenstand

Gegenstand dieser Arbeit ist das Spring-Framework HATEOAS. Bei Spring handelt es sich um ein Java-Framework von VMware, dass Programmierschnittstellen für verschiedene Anwendungsgebiete, zum Beispiel für Datenbanken, Web- oder Cloudservices, zur Verfügung stellt. [1] Bei Hateoas handelt es sich im speziellen um ein Framework für die Entwicklung von REST-basierten Webservices nach dem HATEOAS-Prinzip. HATEOAS steht für „Hypermedia as the Engine of Application State“, das heißt der Zustand einer Anwendung wird allein über ein Hypermedium gesteuert, zum Beispiel über eine Sammlung von Links. [2] [3] Der Client muss dabei, abgesehen von einem generellen Verständnis von Hypermedien, keine Kenntnisse über den Server haben. Der Client und der Server können sich unabhängig voneinander entwickeln.

Motivation

Erste Entwicklungen von HATEOAS gab es zwar bereits 2012, es wurde allerdings erst 2019 in der Version 1.0 veröffentlicht. [4] Es handelt sich also um eine recht neue Technologie, bei der noch nicht hinreichend bekannt ist, ob sie bereits in Projekten Verwendung findet. Allerdings befindet es sich weiterhin stetig in Entwicklung, was die Frage aufwirft, wie relevant Hateoas ist. [5] Außerdem könnte das Thema für zukünftige Vorlesungen genutzt werden. Unabhängig davon ob bereits Software am Markt existiert oder nicht wird anhand einer Beispielapplikation das Konzept von HATEOAS anschaulich umgesetzt.

Probleme

P1 Es ist nicht hinreichend bekannt, ob HATEOAS bereits in der Praxis verwendet wird.

P2 Es ist nicht hinreichend bekannt, ob HATEOAS eine Relevanz besitzt.

P3 Es ist nicht hinreichend bekannt, was mit HATEOAS realisiert werden kann.

P4 Es ist nicht hinreichend bekannt, welche Vorteile HATEOAS gegenüber anderen Spring-Frameworks bietet.

P5 Es ist nicht hinreichend bekannt, welche Nachteile HATEOAS gegenüber anderen Spring-Frameworks bietet.

Ziele

Z1 Es wurde analysiert, ob HATEOAS in der Praxis verwendet wird. Wenn ja, wurde eine Auswahl von Projekten kurz vorgestellt.

Z2 Es wurde analysiert, ob Hateoas eine Relevanz besitzt.

Z3 Es wurde analysiert, was mit Hateoas alles realisiert werden kann.

Z4 Es wurde analysiert, welche Vorteile HATEOAS gegenüber einer normalen REST-Implementierung hat.

Z5 Es wurde analysiert, welche Nachteile HATEOAS gegenüber einer normalen REST-Implementierung hat.

Z6 Es wurde eine Beispielapplikation mit HATEOAS erstellt.

Methodik

Die Vorgehensweise bei der Erstellung dieser Bachelorarbeit wird aus zwei Teilen bestehen: einem Teil Literaturarbeit und einem Teil praktische Umsetzung.

Die Literaturarbeit befasst sich mit den Problemen P1 bis P5 und erfolgt qualitativ. Die Auswertung der Ergebnisse findet induktiv statt.

Die praktische Umsetzung befasst sich mit dem Ziel Z6 und erfolgt innerhalb des Programmierwerkzeugs Eclipse in der Sprache Java.

Die Umsetzung erfolgt im Rahmen einer Bachelorarbeit, deren Bearbeitungszeit 10 Wochen umfasst.

Zur besseren Lesbarkeit wird in dieser Arbeit das generische Maskulinum verwendet, obgleich alle Geschlechter mit einbegriffen sind.

Hauptteil

Die Schritte zu HATEOAS

Das Richardson Maturity Model

Leonard Richardson entwickelte ein Model, das die einzelnen Elemente eines REST-Ansatzes in 3 Schritte unterteilt. [6]

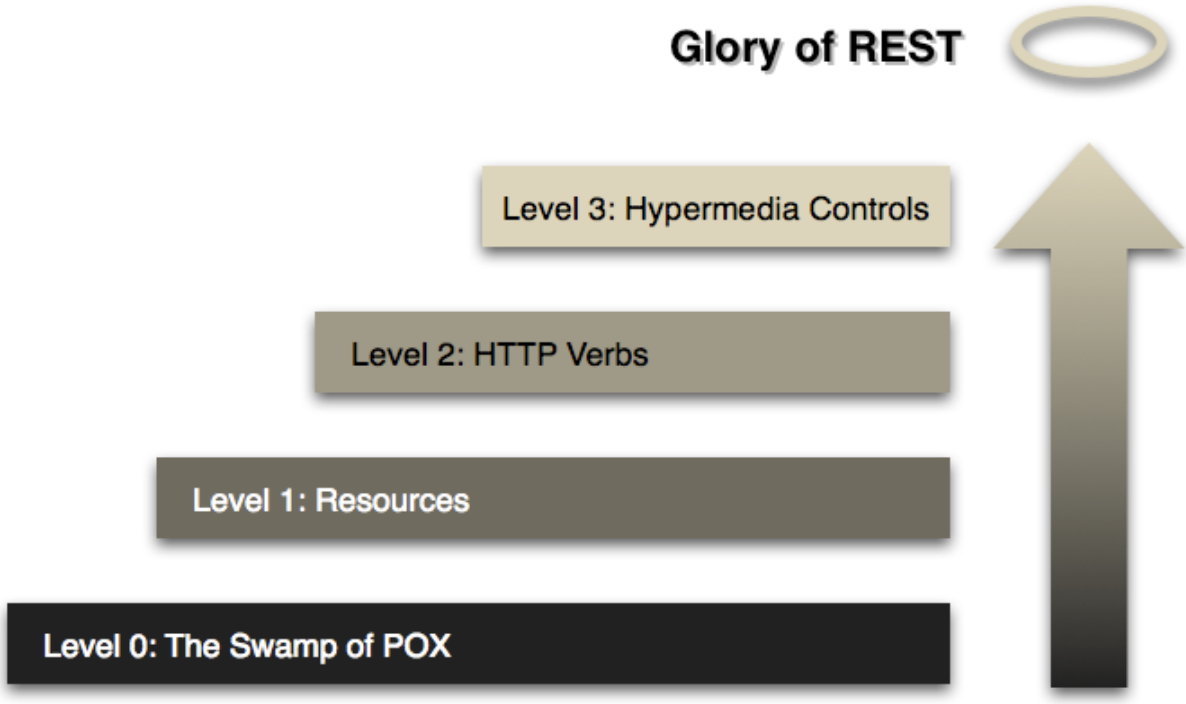

Abbildung 1 Schritte Richtung REST [7]

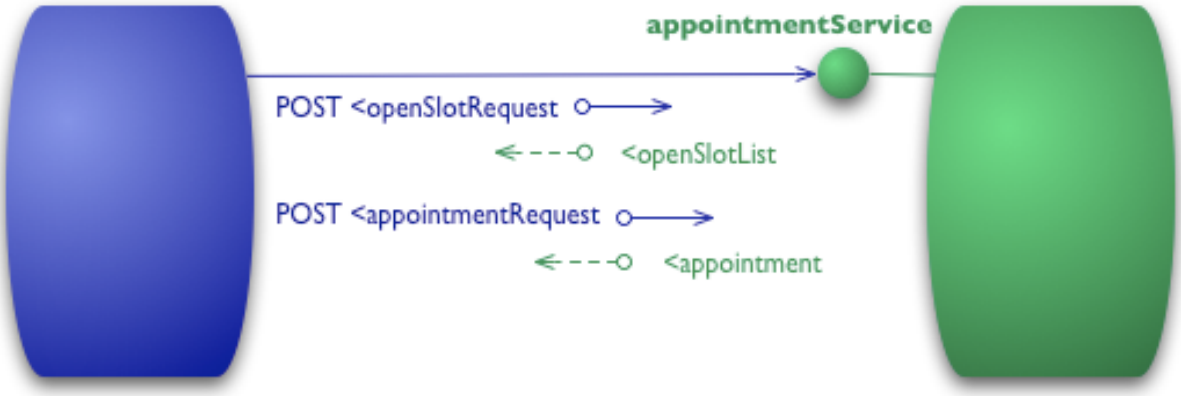

Abbildung 2 Eine Beispielinteraktion auf Ebene 0 [15]

Vor den 3 Schritten steht die erste Ebene 0. Diese hat er „The Swamp of POX“ genannt. HTTP wird in dieser Ebene lediglich als Transportsystem verwendet, ohne die Mechanismen des Internets zu verwenden. HTTP wird als Tunnel verwendet, um eigene Interaktionsmechanismen zu verwenden. Das folgende Beispiel einer Arztpraxis zeigt eine Terminbuchung beim Arzt. Über den URI-Endpunkt `/appointmentService` kann man alle Termine eines Arztes an einem bestimmten Tag abrufen. Der Server sendet alle freien Termine zurück. Anschließend kann man über `/appointmentRequest` einen freien Termin buchen. Wenn die Buchung erfolgreich getätigt werden kann, erhält man vom Server eine Antwort mit dem Statuscode `200 OK` zusammen mit den Daten des Termins. Schlägt die Buchung fehl, dann antwortet der Server mit dem Statuscode `200 OK` und der Nachricht `appointmentRequestFailure`.

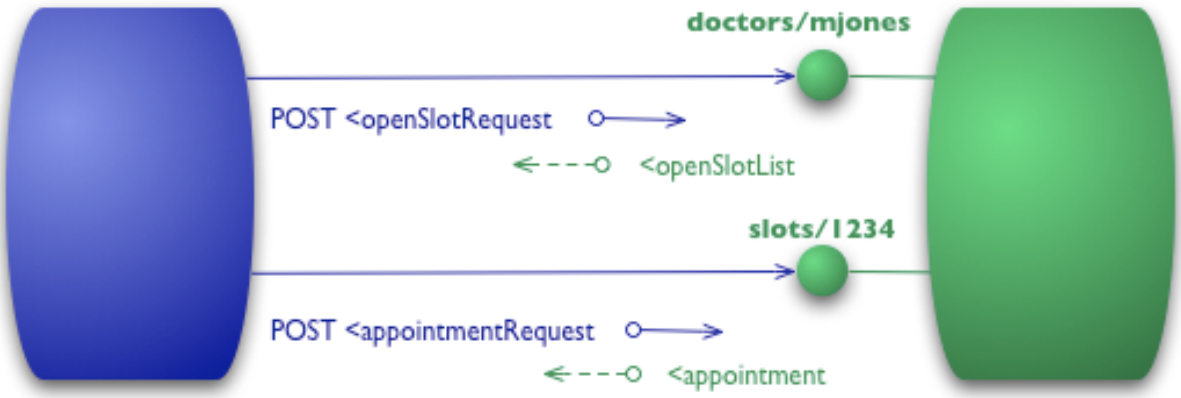

Abbildung 3 Ebene 1 fügt Ressourcen hinzu [8]

Auf der Ebene 1 führt man die Verwendung von Ressourcen ein. Anstatt die Anfragen an einen einzelnen Endpunkt zu senden, sendet man sie an unterschiedliche Ressourcen. Sendet man zum Beispiel eine POST-Anfrage zu der URI `/doctors/mjones`, so erhält man als Antwort dieselben Informationen wie zuvor bei Ebene 0, allerdings ist jeder freie Termin eine eigene Ressource mit einer Slot ID. Für eine Terminbuchung sendet man eine POST-Anfrage an `/slots/1234`.

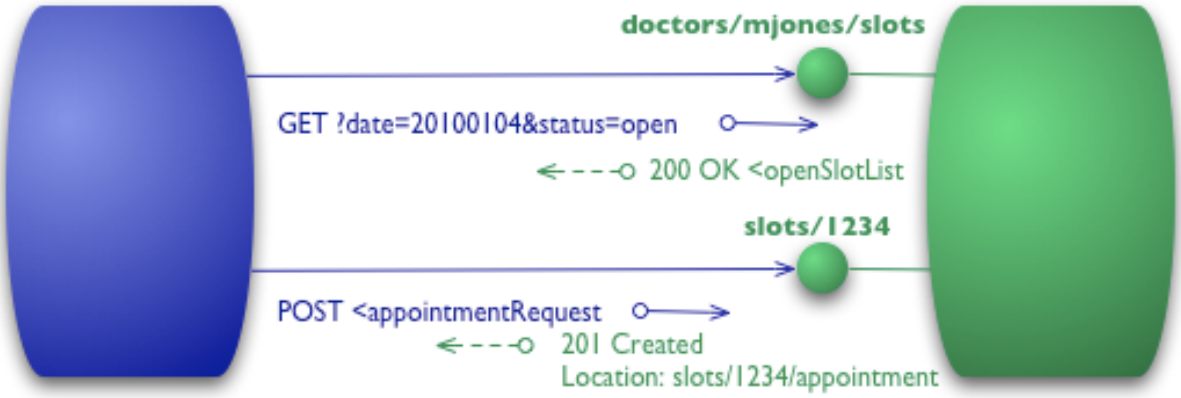

Abbildung 4 Ebene 2 fügt HTTP-Verben hinzu [9]

Auf Ebene 2 wird die Verwendung von HTTP-Verbs eingeführt. Bis jetzt wurden bei den Anfragen nur POST verwendet. Für die Anfrage nach freien Terminen verwendet man nun GET. Mit der GET-Anfrage `/doctors/mjones/slots?date=20100104&status=open` erhält man nun die gleiche Antwort wie im Beispiel der POST-Anfrage, allerdings ist GET eine viel sicherere Operation als POST, da sie den Status des Servers nicht ändert. Man kann so oft man will und in beliebiger Reihenfolge GET-Befehle ausführen. Außerdem eignet sich GET auch für das Caching der Webseite. Um einen Termin zu buchen benutzt man ein HTTP-Verb, was den Status des Servers verändern kann. Mit einer POST-Anfrage an `/slots/1234` kann man einen Termin buchen. Wenn die Terminbuchung erfolgreich war, bekommt man eine Antwort mit dem HTTP-Code `201 Created` und erhält zusätzlich einen Link zu der Ressource `slots/1234/appointment`, mit der man in Zukunft über eine GET-Anfrage den aktuellen Status der Terminbuchung später nochmal abrufen kann. Wenn bei der Terminbuchung etwas schiefgeht, erhält man als Antwort den HTTP-Code `409 Conflict`. Der Server antwortet also immer mit HTTP-Codes, anstatt wie zuvor immer mit `200 OK` zu antworten und dann lediglich innerhalb der Antwort dem Client mitzuteilen, dass die Terminbuchung fehlgeschlagen ist.

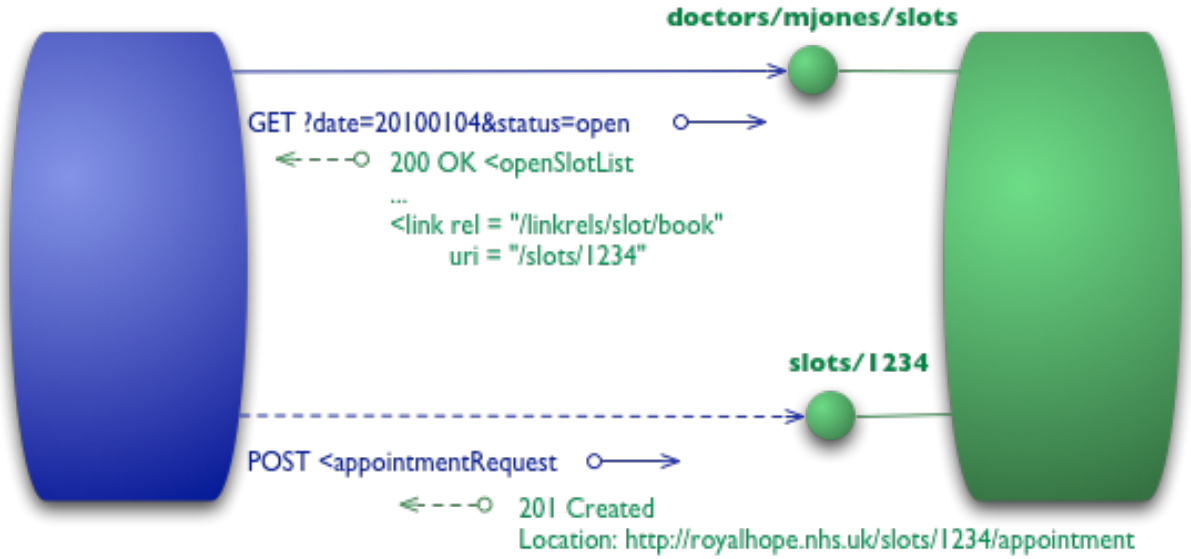

Abbildung 5 Ebene 3 fügt Hypermediensteuerung hinzu [10]

Ebene 3 fügt schließlich die Steuerung über Hypermedien hinzu. Mit der gleichen GET-Anfrage wie in Ebene 2 erhält man nun vom Server eine Liste mit weiterführenden Links zu Ressourcen, die der Client als nächstes anfragen kann. Der Server teilt einem so alle Möglichkeiten mit, welche Anfragen der Client als nächstes stellen kann oder wo sich zum Beispiel die Ressource für das Anlegen eines Termins befindet. So kann der REST-Server bei einer Änderung der Ressource einfach den neuen Link senden und der Client kann entsprechend darauf reagieren, ohne extra angepasst werden zu müssen.

Zu Z1: HATEOAS in der Praxis

Zu der Verwendung von HATEOAS in der Praxis lässt sich leider nicht allzu viel sagen. Lediglich drei Unternehmen konnten gefunden werden, die öffentlich kommunizieren, dass sie HATEOAS für ihr Unternehmen oder für ihr Produkt verwenden. Dass nur drei Unternehmen gefunden wurden, kann einerseits daran liegen, dass schlichtweg nicht viele Unternehmen HATEOAS nutzen. Auf der anderen Seite kann es aber auch daran liegen, dass die Unternehmen, die HATEOAS benutzen, es nicht öffentlich kommunizieren und nicht ihre genutzten Technologien veröffentlichen.

Novomind

Die Novomind AG ist ein in Hamburg ansässiges Unternehmen mit ungefähr 400 Mitarbeitern. Sie betreuen rund 250 Unternehmen auf der ganzen Welt, insbesondere in Europa und Asien. Sie bieten Software für Online-Shops, PIM und Marktplatzintegration. [11] Die API ihrer „iShop“-Software basiert auf dem HATEOAS-Prinzip. [12] Es handelt sich dabei um ein E-Commerce Shopsystem inklusive Weboberfläche. [13] Ihre andere Software namens „iPIM“ verwendet ebenfalls HATEOAS. Es handelt sich dabei um ein Product Information System, also ein System für die zentrale Erstellung, Verwaltung und Pflege aller Produktinformationen des Sortiments sowie Kunden- und Lieferantendaten. Außerdem lassen sich darüber Unternehmensprozesse abbilden. [14]

Yapeal

Bei Yapeal handelt es sich um ein Schweizer FinTech-Unternehmen. Es benutzt das HATEOAS-Prinzip für ihre mobile Banking Plattform. [15] Es wurde 2018 gegründet und ist eine Direktbank, bei der die Kontoverwaltung ausschließlich über eine App erfolgt. [16]

PayPal

Mit über 325 Millionen Kunden weltweit ist PayPal eines der größten Finanzdienstleister der Welt. [17] Das Unternehmen setzt vollständig auf HATEOAS für die Kommunikation des Browsers oder der App mit dem REST-Server. [18]

Zu Z2: Die Relevanz von HATEOAS

Die besseren REST-Services nutzen Spring HATEOAS

Nico Rimmele ist Senior Technical Consultant bei dem Unternehmen „Virtual7“. Er sagt: „Die besseren REST-Services nutzen Spring HATEOAS“. Mit HATEOAS kann man die Schnittstelle mit mehr Informationen Anreichern. Das Ziel ist eine Schnittstelle zu gestalten, die sich selbst beschreibt und die Kopplung zwischen Client und Server so gering wie möglich zu halten. [19]

Wer REST will, muss mit Hateoas ernst machen

Stefan Ullrich arbeitet als Softwarearchitekt bei der Hamburger Sparkasse und hat 15 Jahre Erfahrung mit Java. [20] Er sagt, dass beim Designen einer REST-Ressource ein UML-Zustandsdiagramm ein gutes Mittel sein kann. An dieser Stelle muss aber auch konsequent der REST-Server zustandsgesteuert umgesetzt werden. Aus dem UML-Diagramm kann man einfach die Zustandsübergänge ableiten. Er gibt Roy T. Fields recht, wenn er auf HATEOAS bei der Umsetzung einer REST-API besteht. [21]

REST APIs must be hypertext-driven

Roy Thomas Fielding ist der Mitbegründer von „Apache HTTP Server Project“. Außerdem ist er „Senior Principle Scientist“ bei Adobe. [22] Er stellte 6 Regeln auf die API-Designer anwenden sollen, bevor sie ihre API eine REST-API nennen. Vor allem der letzte Punkt ist sehr aussagekräftig: Eine REST-API sollte keine Vorkenntnisse über den Server haben müssen, bis auf die anfängliche URI. Von diesem Punkt an sollte der Status der Applikation ausschließlich über den Client durch die von dem Server gesendeten URIs geschehen. [23]

Ein Vergleich auf Stack Overflow

Durchsucht man Stack Overflow nach dem Tag „REST“ erhält man 89.545 Fragen mit diesem Tag. [24] Macht man das gleiche mit dem Tag „HATEOAS“ erhält man lediglich 628 Fragen zu diesem Thema. [25]

Zu Z3: Was kann mit Hateoas realisiert werden

Drei Beispiele wurden bereits in dieser Arbeit beschrieben. Ein weiteres Beispiel neben einem Shop- oder Banking-System ist eine Patientenverwaltung. Der praktische Teil dieser Arbeit beschäftigt sich mit einer Patientenverwaltung einer Arztpraxis, die Beispielhaft eine Stammdaten-, Termin- und Messdatenverwaltung beinhaltet. Prinzipiell kann man sagen, dass alles was auch jetzt schon mit REST umgesetzt werden kann, auch mit HATEOAS umgesetzt werden kann. Für jedes Projekt muss allerdings einzeln abgewogen werden, ob die Vorteile oder die Nachteile von HATEOAS überwiegen.

Zu Z4: Die Vorteile von Hateoas

Erkundbare API

Da man bei einer Antwort immer alle folgenden Ressourcen mitgeschickt bekommt, kann man sich von Ressource zu Ressource durch die API bewegen. Dadurch kann man sehr einfach nachvollziehen, wie ein REST-Server aufgebaut ist, ohne sich groß in Dokumentationen einlesen zu müssen. [26]

Eingeschlossene Dokumentation

Eine Dokumentation der API kann Kontextabhängig über eine Ressource erfolgen. [26]

Einfache Client Logik

Da der Client lediglich den einzelnen Links folgen muss, muss der Client kein Verständnis über den Aufbau des REST-Servers haben. Er muss sich nicht anpassen, wenn Ressourcen sich ändern und er muss auch nicht umständlich berechnen, ob zum Beispiel eine bestimmte Schaltfläche angezeigt werden muss oder nicht. Er muss lediglich prüfen, ob der entsprechende Link vorhanden ist. [26]

Auslagern der Daten auf einen anderen Server

Wenn man die Daten von einem Server auf einen anderen auslagern will, wäre es sehr aufwändig den Client zu ändern, wenn dieser nur hartcodierte URLs enthalten würde. Durch die lose Kopplung zwischen Client und Server, die durch HATEOAS entsteht, ist dies allerdings kein Problem. [26]

Versionierung der API

Dadurch, dass jeder Client eine individuelle Liste an Links erhält, können mehrere Versionen der gleichen API nebeneinander existieren. Clients, die auf eine ältere Version angewiesen sind, erhalten die Links zu der alten, kompatiblen Version der API. Clients, die bereits eine neuere Version nutzen können, erhalten dementsprechend die Links zu der neuen Version. [26]

Mehrere Implementierungen der gleichen API

Da die Clients keine hartcodierten Links enthalten, kann dieselbe API gleichzeitig auf mehreren Servern gehostet werden. Dadurch ändern sich lediglich die einzelnen Links, jedoch nicht deren Verbindungen untereinander. Der Client kann also auch damit umgehen was ohne HATEOAS so nicht möglich wäre. [26]

Autorisierung des Benutzers

Um zu überprüfen, ob ein Benutzer eine Aktion durchführen kann, müsste bei einem REST-Server ohne HATEOAS die Überprüfung auf Client-Seite stattfinden. Entweder, indem der Client eine Anfrage an den Server stellt, ob die jeweilige Aktion durch den Nutzer ausgeführt werden kann oder indem der Server direkt einen „Flag“ mitschickt, ob die jeweilige Aktion durchgeführt werden kann. Bei HATEOAS genügt es zu überprüfen, ob für die Aktion ein jeweiliger Link vorhanden ist. Der REST-Server hat in dem Fall die Berechtigungsüberprüfung bereits übernommen und schickt dem Client nur Links, die der Nutzer auch aufrufen darf. [27]

Wartbarkeit

Sollte in der Entwicklung eines REST-Servers einmal der Aufbau der URLs verändert werden, sind auf der Client-Seite keine Änderungen nötig, da in dem Client keine hartcodierten URLs zu finden sind und der Client einfach die neuen URLs aus den Antworten des Servers bezieht. [27]

Tracking

Man kann auf dem REST-Server genau nachvollziehen, von welcher Ressource der Nutzer kommt und welchen Link er als nächstes aufgerufen hat. Dadurch erhält man durch HATEOAS automatisches Tracking, ohne dass der Nutzer es verhindern kann. [27]

Zu Z5: Die Nachteile von HATEOAS

Tracking

Die Eigenschaft, dass man mit HATEOAS eine nicht verhinderbare Möglichkeit bekommt, die Nutzer zu tracken, kann auch von Unternehmen missbraucht werden. [27]

Keine etablierten Standards

Es gibt keine standardisierte Form wie die Links innerhalb der JSON-Antwort dargestellt werden sollen. Je nach genutztem Framework gibt es verschiedene Darstellungsformen, was wiederum die Interoperabilität beeinträchtigt. [15]

Zu wenig Libraries

Je nach verwendeter Programmiersprache und Framework kann es schwierig werden, eine passende HATOEAS-Library zu finden. [15]

Mehr Bandbreite

Dadurch, dass bei jeder Antwort alle Links mit angehängt werden, werden sehr viele Daten erzeugt. Selbst wenn die eigentlichen Nutzdaten der Antwort sehr klein sind, die angehängten Links können dafür sorgen, dass die Nachricht trotzdem sehr groß wird. Außerdem erhält der Client sehr viele Daten, die er vielleicht gar nicht braucht. [15] [27]

Mehr Aufwand

Dadurch, dass der Server die komplette Kontrolle über den Ablauf hat, muss natürlich auch mehr bei der Entwicklung bedacht werden. Man muss bei der Ressource überlegen, mit welchen anderen Ressourcen sie in Verbindung steht und unter welchen Umständen der Client die Ressourcen aufrufen können soll. Mit HATEOAS wird also die Entwicklung der Serverseite deutlich komplexer. [15] [27]

Zu Z6: Die Beispielapplikation

Das Beispielprogramm von Spring.io

Die Beispielapplikation wurde entwickelt unter macOS Monterey mit der Versionsnummer 12.3.1 und innerhalb des Programmierwerkzeugs „Eclipse“ mit der Versionsnummer 4.23.0 M2. Diese lässt sich kostenlos über die Webseite der „Eclipse Foundation“ herunterladen. [28] Die Java Runtime Environment hat die Version 17.0.2. Mithilfe des Guides unter https://spring.io/guides/gs/rest-hateoas/ kann man ein einfaches Testprogramm erstellen. Zunächst einmal kann man sich über die Webseite „Spring Initializr“ https://start.spring.io ein bereits vorkonfiguriertes Projekt erstellen lassen. Man wählt aus, ob man „Maven“ oder „Gradl“ und welche Sprache man benutzen will. Bei den „Dependencies“ fügt man HATEOAS hinzu und erhält damit ein vorkonfiguriertes Projekt. Danach klickt man auf „Generate“, entpackt den heruntergeladenen Ordner und öffnet das Projekt mit Eclipse.

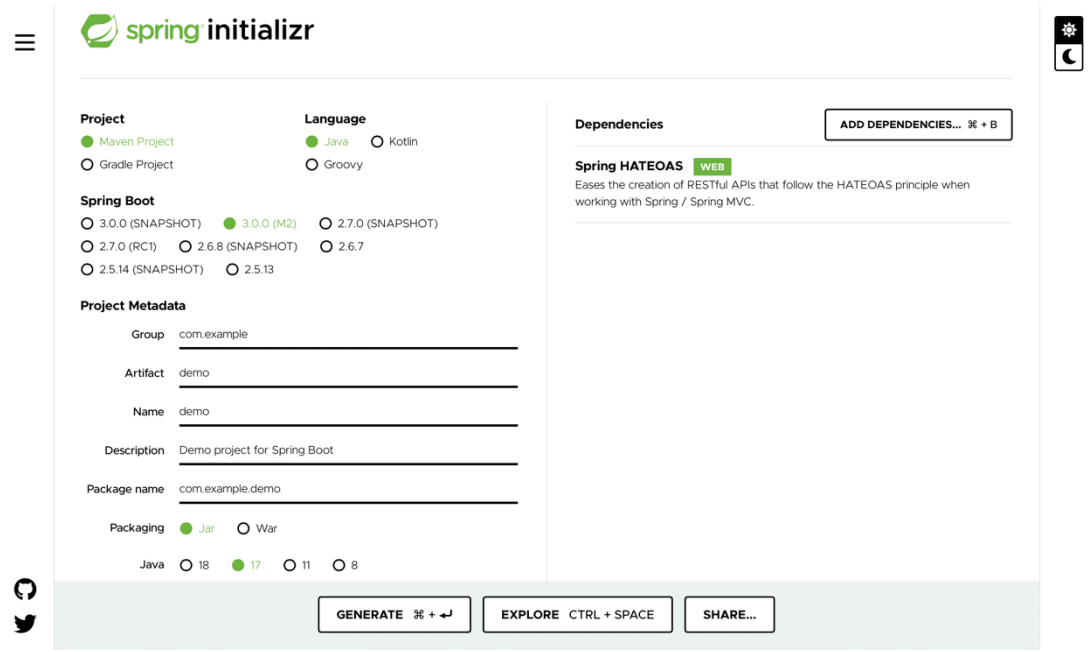

Abbildung 6 Spring Initializr [29]

Laut dem Guide soll man folgenden Block in die pom.xml einfügen

```
<dependency>
  <groupId>com.jayway.jsonpath</groupId>
  <artifactId>json-path</artifactId>
  <scope>test</scope>
</dependency>
```

Allerdings erhält man einen Error beim Ausführen des Testprogramms.

Wenn man die Zeile

```
<scope>test</scope>
```

entfernt wird das Programm korrekt ausgeführt.

Als erstes erstellt man eine Klasse, die eine Ressource repräsentiert.

```
package com.example.resthateoas;

import org.springframework.hateoas.RepresentationModel;
import com.fasterxml.jackson.annotation.JsonCreator;
import com.fasterxml.jackson.annotation.JsonProperty;

public class Greeting extends RepresentationModel<Greeting>
{

	private final String content;

	@JsonCreator
	public Greeting(@JsonProperty("content") String content)
	{
		this.content = content;
	}

	public String getContent()
	{
		return content;
	}
}
```

[3]

Dadurch, dass die Klasse eine Erweiterung von `RepresentationModel` ist, kann sie genutzt werden um sie als Ressource zu verwenden. Die Anmerkung `@JsonCreator` ist ein Indikator für „Jackson“, einer Library um Java Objekte in JSON darzustellen und anders herum. [30] Über die Anmerkung `@JsonProperty` markiert man die Parameter der Klasse und bestimmt den Namen innerhalb der JSON-Datei. Über die get-Methode erhält Jackson den jeweiligen Wert der Variable.

Als nächstes wird der REST-Controller erstellt. Dieser bearbeitet die Anfragen an den Server und stellt eine Antwort zu Verfügung.

```
package com.example.resthateoas;

import static
org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.linkTo;
import static
org.springframework.hateoas.server.mvc.WebMvcLinkBuilder.methodOn;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class GreetingController
{

      private static final String TEMPLATE = "Hello, %s!";

      @GetMapping("/greeting")
      public HttpEntity<Greeting> greeting(
      @RequestParam
      (value = "name", defaultValue = "World") String name)
      {

            Greeting greeting =
            new Greeting(String.format(TEMPLATE, name));


            greeting.add(linkTo(methodOn(GreetingController.class)
            .greeting(name))
            .withSelfRel());

            return new ResponseEntity<>(greeting, HttpStatus.OK);
      }
}
```

[3]

Die Klasse wird mit der Anmerkung `@RestController` versehen. Mit der Anmerkung `@GetMapping` stellt man sicher, dass die HTTP-GET-Anfragen an die URL `/greeting` auf die Methode `greeting()` abgebildet werden. Mit der Anmerkung `@RequestParam` verbindet man die Variable `name` in der HTTP-Anfrage mit dem Parameter `name` in der Methode. Außerdem kann man festlegen, ob die Variable `required`, also ob sie erforderlich ist, oder ob es eine `defaultValue`, also einen Standardwert gibt. In diesem Fall ist die Variable automatisch `not required`. Es wird ein `Greeting`-Objekt erstellt, an das dann die HATEOAS-Links angehängt werden. `linkTo()` und `methodOn()` sind beides statische Methoden der Klasse `ControllerLinkBuilder`, die in das Projekt automatisch durch Spring importiert wurden. Mit der Methode `linkTo()` hängt man einen Link an, mit der Methode `methodOn()` erzeugt man einen Aufruf der Methode `greeting()` innerhalb derselben Methode. Hierdurch wird ein `LinkBuilder`-Objekt erstellt, dass die Möglichkeit hat die Ressource zu erstellen.

Mit `withSelfRel()` fügt man einen selbst referenzierenden Link zu der Antwort hinzu. Die Methode gibt eine `ResponseEntity` zurück, die das soeben erstellte `Greeting`-Objekt inklusive der angehängten Links und einem HTTP-Status-Code enthält.

Die Main-Methode befindet sich in der automatisch erstellten Klasse `RestHateoasApplication`.

```
package com.example.resthateoas;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class RestHateoasApplication {

	public static void main(String[] args) {
		SpringApplication.run(RestHateoasApplication.class, args);
	}

}
```

[3]

Nachdem man die Anwendung gestartet hat, kann man in seinem Browser oder mit einem beliebigen REST-Client die Seite unter http://localhost:8080/greeting aufrufen. Für diese Arbeit wurde der REST-Client „Postman“ unter der Version 9.13.0 benutzt. Als Antwort im JSON-Format erhält man

```
{
    "content": "Hello, World!",
    "_links": {
        "self": {
            "href": "http://localhost:8080/greeting?name=World"
        }
    }
}
```

Den Parameter für den Namen kann man ändern, in dem man an die URL `?name=` anhängt. Zum Beispiel http://localhost:8080/greeting?name=Andreas gibt als Antwort

```
{
    "content": "Hello, Andreas!",
    "_links": {
        "self": {
            "href": "http://localhost:8080/greeting?name=Andreas"
        }
    }
}
```

Erweiterung der Beispielapplikation zu einer Patientenverwaltung

Die Beispielapplikation wird erweitert, um die Funktionalität einer Patientenverwaltung bereit zu stellen. Dazu zählten Patientenstammdaten, die in diesem Fall auf die Daten: ID, Vorname, Nachname, Geburtsdatum, Straße, Ort und Postleitzahl beschränkt werden. Zu einem Patienten kann man Messdaten hinzufügen, in diesem Fall wurden sie beschränkt auf Daten einer Blutdruckmessung mit den Werten: systolischer Blutdruck, diastolischer Blutdruck und Datum und Uhrzeit der Messung. Außerdem kann man einen Termin für den Patienten anlegen mit Datum und Uhrzeit.

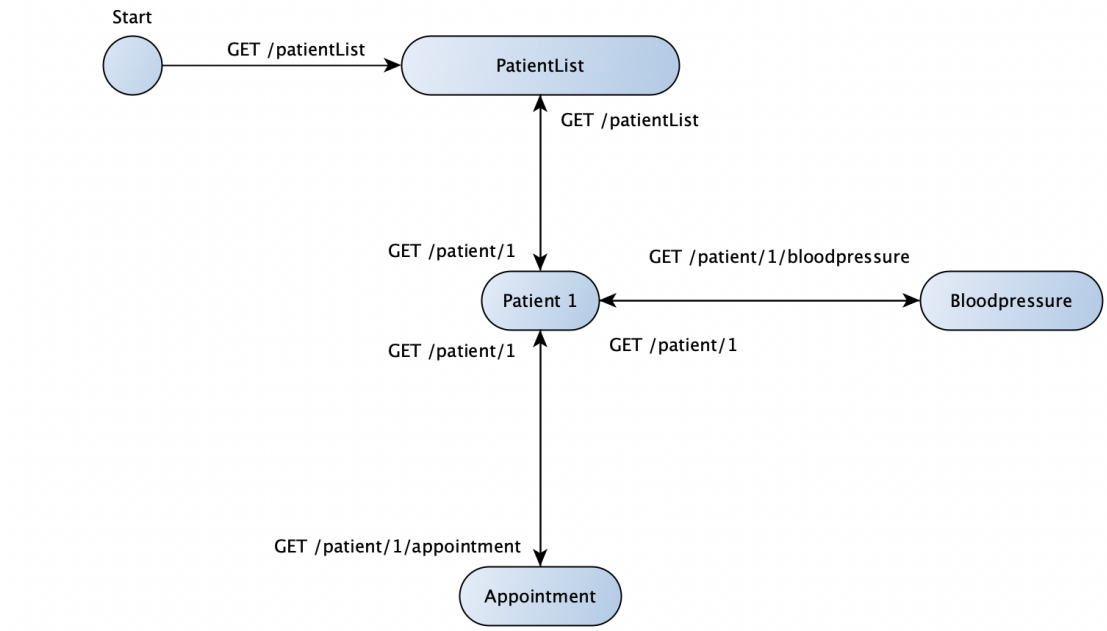

Abbildung 7 Flussdiagramm des REST-Servers

Zunächst wird entsprechend der Vorlage aus der Beispielapplikation die Klasse `Patient` mit folgenden Variablen erstellt.

```
public class Patient extends RepresentationModel<Patient> {
    private int id;
    private String firstName;
    private String lastName;
    private Date birthdate;
    private String street;
    private String city;
    private String zipCode;
    private final SimpleDateFormat DATEFORMAT = new
SimpleDateFormat("dd.MM.yyyy");

    private BloodPressure bloodPressure;

    private Appointment appointment = null;
```

Der Konstruktor enthält analog zur Beispielapplikation die Anmerkungen `@JsonCreator` und `@JsonProperty` für Jackson.

```
@JsonCreator
public Patient(@JsonProperty("id") int id,
@JsonProperty("firstName") String firstName,
@JsonProperty("lastName") String lastName,
@JsonProperty("birthDate") String birthDate,
@JsonProperty("street") String street,
@JsonProperty("city") String city,
@JsonProperty("zipCode") String zipCode) {
      this.id = id;
      this.firstName = firstName;
      this.lastName = lastName;
      try {
          this.birthdate = DATEFORMAT.parse(birthDate);
      } catch (ParseException e) {
          e.printStackTrace();
      }
      this.street = street;
      this.city = city;
      this.zipCode = zipCode;

      bloodPressure = new BloodPressure();
    }
```

Es wurde noch ein zweiter Konstruktor erstellt, der als Paramater ein `Patient`-Objekt erhält und daraus eine Kopie erstellt, in dem er alle Variablen des übergebenen Objekts ausließt und daraus ein neues `Patient`-Objekt erstellt. Wozu dieser Konstruktor gebraucht wird, wird später noch deutlich.

```
    public Patient(Patient patient) {
      this.id = patient.getId();
      this.firstName = patient.getFirstName();
      this.lastName = patient.getLastName();
      this.birthdate = patient.getBirthdate();
      this.street = patient.getStreet();
      this.city = patient.getCity();
      this.zipCode = patient.getZipCode();
      this.bloodPressure = patient.getBloodPressure();
      this.appointment = patient.getAppointment();
    }
```

Für jede Variable wurde eine get-Methode erstellt. Interessant hierbei sind die get-Methoden für das Geburtsdatum, den Blutdruck und dem Termin.

```
@JsonIgnore
public Date getBirthdate() {
  return birthdate;
}

@JsonIgnore
public BloodPressure getBloodPressure() {
  return bloodPressure;
}

@JsonIgnore
public Appointment getAppointment() {
  return appointment;
}
```

Wenn man nicht will, dass JSON einen Wert einer Variablen darstellt, kann man die entsprechende get-Methode mit der Anmerkung `@JsonIgnore` versehen. Jackson ignoriert daraufhin die entsprechende get-Methode und zeigt den Wert der Variable in der Server-Antwort nicht an. Für das Geburtsdatum wird eine andere get-Methode genutzt, die einen String des `Date`-Objekts zurückgibt und nicht das eigentliche `Date`-Objekt. `BloodPressure` und `Appointment` muss man an dieser Stelle nicht anzeigen, da dafür eigene Klassen für deren Darstellung erstellt wurden.

Als nächstes wird die Controller-Klasse von `Patient` erstellt. Sie nimmt Anfragen des Clients entgegen und generiert eine Antwort.

```
@RestController
@RequestMapping("/patient")
public class PatientController {
```

Mit der Anmerkung `@RequestMapping` sind alle Abbildungen von URLs relativ zu dieser URL.

```
@GetMapping("/{id}")
    public HttpEntity<Patient> patient(@PathVariable int id) {
      Patient patientCopy = null;
      for (Patient p : PatientAdministrationApplication.getPatientList()) {
          if (p.getId() == id) {
            patientCopy = new Patient(p);
          }
      }
```

Die Methode nimmt Anfrage über die URL `/{id}` entgegen und erzeugt eine `ResponseEntity`. Der Wert in der geschweiften Klammer ist ein Platzhalter für die ID des Patienten. Die Methode nimmt den Wert für die ID aus dem Platzhalter und durchsucht die Patientenliste nach dem passenden Patienten. Anschließend wird über den zweiten Konstruktor der Klasse `patient` eine Kopie des Patienten erstellt.

An diese Kopie werden anschließend die Links angehängt.

```
patientCopy.add(linkTo(methodOn(BloodPressureController.class)
.bloodPressure(id)).withRel("bloodPressure"));

      if (patientCopy.getAppointment() != null) {
patientCopy.add(linkTo(methodOn(AppointmentController.class)
.appointment(id)).withRel("appointment"));
      } else {
patientCopy.add(linkTo(methodOn(AppointmentController.class)
.addNewAppointment(id, null))
.withRel("addNewAppointment"));
      }

patientCopy.add(linkTo(methodOn(PatientController.class)
.deletePatient(id)).withRel("deletePatient"));

patientCopy.add(linkTo(methodOn(PatientController.class)
.patient(id)).withSelfRel());

      return new ResponseEntity<>(patientCopy, HttpStatus.OK);
    }
```

Es werden Relationen für den Blutdruck, den Termin und für das Löschen des Patienten erzeugt. Durch if-Abragen kann bestimmt werden, welche Links angehängt werden sollen und welche nicht. Wenn zum Beispiel der Patient kein `Appointment`-Objekt besitzt, wird statt der Ressource des Termins die Ressource zum Erstellen eines Termins an den Patienten angehängt. Man hängt die Links an die Kopie des Patienten an, nicht an das eigentliche Objekt. Ansonsten würde bei jeder Anfrage die Links erneut angehängt werden und nie gelöscht werden.

```
@PostMapping("/")
public HttpEntity<Patient> postPatient(@RequestBody ObjectNode objectNode){
      int id = objectNode.get("id").asInt();
      String firstName = objectNode.get("firstName").asText();
      String lastName = objectNode.get("lastName").asText();
      String birthDate = objectNode.get("birthDate").asText();
      String street = objectNode.get("street").asText();
      String city = objectNode.get("city").asText();
      String zipCode = objectNode.get("zipCode").asText();

      Patient newPatient = new Patient(id, firstName, lastName,
      birthDate, street, city, zipCode);

      PatientAdministrationApplication.getPatientList().add(newPatient);
```

Durch `@PostMapping` wird festgelegt, dass diese Methode die POST-Anfragen entgegennimmt. Mit der Anmerkung `@RequestBody` legt man fest, dass die entsprechende Variable über einen Request-Body gesendet wird. Über das `objectNode` kann der Request-Body ausgewertet werden und die einzelnen Werte ausgelesen werden.

Beispielhaft sieht ein `RequestBody` im JSON-Format folgendermaßen aus:

```
{
    "id": 4,
    "firstName": "Monika",
    "lastName": "Meier",
    "birthDate": "01.09.1993",
    "street": "Ringstrasse 525",
    "city": "Hamburg",
    "zipCode": "22767"
}
```

Analog wie bei der Methode `getPatient()` werden auch hier die einzelnen Links angehängt.

```
patientCopy.add(linkTo(methodOn(BloodPressureController.class)
.bloodPressure(id)).withRel("bloodPressure"));

patientCopy.add(linkTo(methodOn(AppointmentController.class)
.appointment(id)).withRel("appointment"));

patientCopy.add(linkTo(methodOn(PatientController.class)
.deletePatient(id)).withRel("deletePatient"));

patientCopy.add(linkTo(methodOn(PatientController.class)
.patient(id)).withSelfRel());

      return new ResponseEntity<>( patientCopy, HttpStatus.CREATED);
    }
```

Abweichend dazu geben wir den passenden HTTP-Status `Created` zurück.

Abschließend wird die Methode `deletePatient()` zum Löschen eines Patienten erstellt.

```
@DeleteMapping("/{id}")
public HttpEntity<Patient> deletePatient(@PathVariable int id) {
      Patient patient = null;

      for (Patient p : PatientAdministrationApplication.getPatientList()) {
          if (p.getId() == id) {
            patient = p;
          }
      }

      PatientAdministrationApplication.getPatientList().remove(patient);

      return new ResponseEntity<>(patient, HttpStatus.OK);
    }
```

Analog zu der Klasse `Patient` werden die anderen Klassen für `Appointment` und `BloodPressure` und ihre entsprechenden Controller-Klassen erstellt. Außerdem gibt es die Klassen `PatientList` und `patientListController`, die über die Index-Adresse des Servers, sprich der Adresse http://localhost:8080/ erreicht werden können und alle Patienten inklusiver ihrer URIs ausgegeben.

Die Klasse, die die Main-Methode enthält, erzeugt auch die Testdaten.

```
@SpringBootApplication
public class PatientAdministrationApplication {
private static ArrayList<Patient> patientList = new ArrayList<Patient>();

public static void main(String[] args) {

	initialize();

	SpringApplication.run(PatientAdministrationApplication.class, args);
}

public static void initialize() {
Patient patient1 = new Patient(1, "Andreas", "Heese", "16.11.1995", "Innere
Schneeberger Straße 23", "Zwickau",
		"08056");
patient1.getBloodPressure().addBloodPressure(120, 80, "02.02.2022 um 14:00
Uhr");
patient1. getBloodPressure().addBloodPressure(130, 70, "17.02.2022 um 13:30
Uhr");

Patient patient2 = new Patient(2, "Maik", "Hansen", "23.04.1967", "Baumweg
7", "Köln", "50667");
Patient patient3 = new Patient(3, "Linda", "Mueller", "02.06.1983",
"Postweg 34", "Dresden", "01867");

patientList.add(patient1);
patientList.add(patient2);
patientList.add(patient3);

Appointment appointment = new Appointment("04.04.2022 um 13:00 Uhr");
patient1.setAppointment(appointment);
}

/**
*
* @return patientList that contains all the Patient objects
*/
public static ArrayList<Patient> getPatientList() {
	return patientList;
    }

}
```

Nachdem man den Rest-Server gestartet hat, kann man zum Beispiel über einen REST-Client Anfragen an den Server senden.

Testweise wird eine get-Anfrage an die URL http://localhost:8080/ gesendet, die die Daten aller Patienten ausgibt.


```
{
    "patientList": [
        {
            "id": 1,
            "firstName": "Andreas",
            "lastName": "Heese",
            "street": "Innere Schneeberger Straße 23",
            "city": "Zwickau",
            "zipCode": "08056",
            "birthdateString": "16.11.1995",
            "_links": {
                "self": {
                    "href": "http://localhost:8080/patient/1"
                }
            }
        },
        {
            "id": 2,
            "firstName": "Maik",
            "lastName": "Hansen",
            "street": "Baumweg 7",
            "city": "Köln",
            "zipCode": "50667",
            "birthdateString": "23.04.1967",
            "_links": {
                "self": {
                    "href": "http://localhost:8080/patient/2"
                }
            }
        },
    "_links": {
        "addNewPatient": {
            "href": "http://localhost:8080/patient/"
        },
        "self": {
            "href": "http://localhost:8080/"
        }
    }
}
```

Hier erhält man auch die URI jedes einzelnen Patienten und kann diese ebenfalls mit einer POST-Anfrage aufrufen, wie zum Beispiel `http://localhost:8080/patient/1`. Als Antwort erhält man die JSON-Repräsentation des Patienten mit der ID 1 inklusiver seiner Links.


```
{
    "id": 1,
    "firstName": "Andreas",
    "lastName": "Heese",
    "street": "Innere Schneeberger Straße 23",
    "city": "Zwickau",
    "zipCode": "08056",
    "birthdateString": "16.11.1995",
    "_links": {
        "bloodPressure": {
            "href": "http://localhost:8080/patient/1/bloodpressure"
        },
        "appointment": {
            "href": "http://localhost:8080/patient/1/appointment"
        },
        "deletePatient": {
            "href": "http://localhost:8080/patient/1"
        },
        "self": {
            "href": "http://localhost:8080/patient/1"
        }
    }
}
```

Das Testen des REST-Servers

Getestet wurde der REST-Server mit JUnit-Tests. Die Testklasse wird mit der Anmerkung `@SpringBootTest` versehen und über `classes=` gibt man denNamen der Klasse an, die die Main-Methode enthält.

```
@SpringBootTest(classes = PatientAdministrationApplication.class)
public class PatientAdministrationApplicationTests {
	@Autowired
	private WebApplicationContext wac;

	private MockMvc mockMvc;
```

Über die Methode `setup()` wird mit Hilfe eines `MockMvc`-Objekts eine Testinfrastruktur bereitgestellt. [31] Mit der Anmerkung `@BeforeEach` wird sichergestellt, dass diese Methode vor jedem Test ausgeführt wird.

```
@BeforeEach
public void setup() throws Exception {
	mockMvc = MockMvcBuilders.webAppContextSetup(wac).build();
    }
```

Über die Anmerkung `@Test` legt man eine Methode fest, die den Test ausführt, hier Beispielhaft die Methode `getPatient()`. Über `mockMvc.perform()` lässt man das Mock-Objekt etwas ausführen. Über `MockMvcRequestBuilders.get()` erstellt man eine get-Anfrage an die Adresse `/patient/1` und legt den `contentType` der Anfrage fest. Anschließend lässt sich mit der Methode `andExpect()` bestimmte Merkmale der Antwort überprüfen, zum Beispiel ob der HTTP-Status `200Ok` oder der Inhalt der Antwort einem vorher festgelegten String entspricht.

```
@Test
public void getPatient() throws Exception {
	mockMvc.perform(MockMvcRequestBuilders.get("/patient/1").contentType(
	MediaType.APPLICATION_JSON_VALUE))
	.andExpect(MockMvcResultMatchers.status().isOk())
	.andExpect(MockMvcResultMatchers.content()
	.json(JSONPatientOutputString));
    }
```

Die Entwicklung des Clients

Für den Client wurde das Spring Framework „Thymeleaf“ verwendet. Bei Thymeleaf handelt es sich um eine Template-Engine. Man erstellt HTML-Seiten mit Platzhaltern, die zur Laufzeit mit Daten gefüllt werden können. [32] Als Startunkt eignet sich auch hier die Seite Spring Initializr. Dort fügt man unter „Dependencies“ Spring Web und Thymeleaf hinzu.

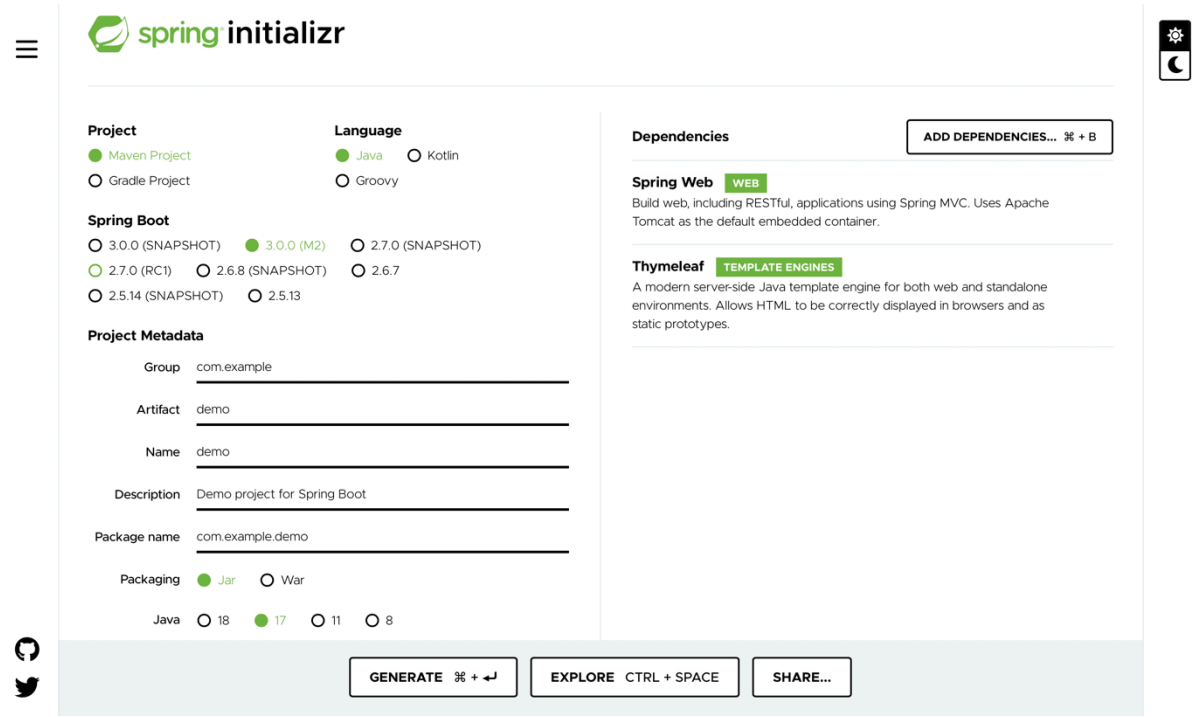

Abbildung 8 Spring Initializr [29]

Als Ausgangspunkt dient die Klasse `PatientAdministrationClientApplication`.
Diese enthält die Main-Methode und die URL des entsprechenden REST-Servers, die in diesem Fall localhost mit dem Standard-Port 8080 entspricht.

```
@SpringBootApplication
public class PatientAdministrationClientApplication {

// URL of the corresponding RestServer
public static final String RestServer = "http://localhost:8080";

public static void main(String[] args) {
SpringApplication.run(PatientAdministrationClientApplication.class, args);

}

public String getRestServer() {
      return RestServer;
    }

}
```

Da ohne Änderung dieser Thymeleaf-Server ebenfalls den Port 8080 nutzen würde, muss man ihn manuell ändern. Dies geschieht indem man in der Datei `application.properties`, die automatisch im Projekt vorhanden ist, die Zeile `server.port=8081` einfügt.

Die Startseite des Clients besteht aus einer Auflistung aller Patienten mit der ID und den Namen des Patienten zusammen mit einer Schaltfläche, um alle Daten des Patienten anzuzeigen. Der Controller dieser Seite ist die Klasse `PatientListClient`.

```
@Controller
public class PatientListClient {

    @Value("${server.port}")
    private String serverPort;

    private String restServer =
PatientAdministrationClientApplication.RestServer;
```

Durch die Anmerkung `@Value(„${server.port}“)` kann man den Server-Port aus der Datei `application.properties` auslesen.

Für das GET-Mapping erstellt man die Methode `patientList()`. Der Parameter `Model` muss nicht selbst beim Aufruf übergeben werden, sondern wird von Spring automatisch erstellt. Dem `Model`-Objekt kann man Daten und Objekte hinzufügen, auf die man später innerhalb des HTML-Templates Zugriff hat.

```
@GetMapping("/patientlist")
    public String patientList(Model model) {
try {
```

Anschließend erzeugt man eine HTTP-Anfrage an den REST-Server

```
// Create HTTP Client
HttpClient httpClient = HttpClientBuilder.create().build();

// Create new postRequest with aforementioned URL
HttpGet getRequest = new HttpGet(restServer);

// Execute your request and catch response
HttpResponse response = httpClient.execute(getRequest);
```

Die Antwort des Servers liest man aus und erstellt daraus ein JSON-Objekt, das dem `Model`-Objekt hinzugefügt wird.

```
// Read the response from the REST server, create a
//JSON object out of it and
// adds it to the model.
BufferedReader br = new BufferedReader(new
InputStreamReader((response.getEntity().getContent())));
JSONObject jo = new JSONObject(br.readLine());
model.addAttribute("jo", jo);
```

Man liest die Links aus der Antwort und erstellt daraus eine Map.

```
// Get the data under the JSON path _links and creates a Mao out of it
JSONObject linksObj = (JSONObject) jo.get("_links");
Map links = linksObj.toMap();
Iterator<Map.Entry> itr = links.entrySet().iterator();
```

Über diese Map iteriert man und erzeugt die passenden Link-Daten für Thymeleaf. Einmal der Link für den Thymeleaf-Server, der einfach auf den Namen der Relation endet, zum Beispiel `addNewPatient`. Aus dem Link für die Ressource wird die JSON-Spezifische Syntax entfernt und dieser wird anschließend in Base64 codiert. Das alles wird in einer neuen Map namens `linkCleanTh` gespeichert. Diese enthält den Namen der Relation als Schlüssel und ein String-Array bestehend aus dem Thymeleaf-Link und der Base64 codierten Ressource auf dem REST-Server.

```
// A map out of Thymelead links without the JSON specific syntax like
"href="
Map<String, String[]> linkCleanTh = new HashMap<String, String[]>();

// Iterate over the Link map
while (itr.hasNext()) {
	Map.Entry pair = itr.next();
	String key = pair.getKey().toString();

	// Create a Thymeleaf link with the serverPort of the REST server
	// and the key at the end, which is the name of the URI in the JSON
	// path of the JSON response from the REST server.
	String thString = "http://localhost:" + serverPort + "/" + key;

	// Link of the HATEOAS resource without the JSON specific syntax
	String haString = pair.getValue().toString()
	.replace("{href=", "").replace("}", "");

	// Base64 encoding of HATEOAS resource
	String haString64 = Base64.getUrlEncoder()
	.encodeToString(haString.getBytes(StandardCharsets.UTF_8
	.toString()));

	linkCleanTh.put(key, new String[] { thString, haString64 });

}
```

Für die Daten der einzelnen Patienten erstellt man für jeden Parameter eine ArrayList.

```
ArrayList<Integer> idArray = new ArrayList<Integer>();
ArrayList<String> firstNameArray = new ArrayList<String>();
ArrayList<String> lastNameArray = new ArrayList<String>();
ArrayList<String> patientLinkArray = new ArrayList<String>();
```

Die einzelnen Patienten werden ausgelesen und in einem JSON-Array gespeichert.

```
JSONArray joArray = jo.getJSONArray("patientList");
```

Über die einzelnen Objekte in dem JSON-Array iteriert man, liest die Werte aus und speichert sie in der entsprechenden Array-List für die Patientendaten. Außerdem liest man auch die URI für die einzelnen Patienten aus, die genauso in Base64 codiert werden.

```
// Iterate over the patientList, get id, firstName, lastName
//and links from each
// Patient object and add each of them to the corresponding ArrayList.
for (int i = 0; i < joArray.length(); i++) {
            JSONObject patientObj = joArray.getJSONObject(i);

            idArray.add(patientObj.getInt("id"));
            firstNameArray.add(patientObj.getString("firstName"));
            lastNameArray.add(patientObj.getString("lastName"));

            JSONObject patientLinksObj = (JSONObject)
            patientObj.get("_links");

            Map patientLinks = patientLinksObj.toMap();

            Iterator<Map.Entry> patientLinkitr =
            patientLinks.entrySet().iterator();

            while (patientLinkitr.hasNext()) {
                Map.Entry pair = patientLinkitr.next();

                String key = pair.getKey().toString();

                // Get the URI of the self reference and add it to the
                //patientLinkArray
                if (key.equals("self")) {
                  // Link of the HATEOAS resource
                  String haString =
                  pair.getValue().toString().replace("{href=",
                  "").replace("}", "");

                  // Base64 encoding of HATEOAS resource
                  String haString64 = Base64.getUrlEncoder()
                  .encodeToString(haString
                  .getBytes(StandardCharsets.UTF_8.toString()));

                  patientLinkArray.add(haString64);
                }

            }
        }
```

Abschließend fügt man alles dem `Model`-Objekt hinzu.

```
model.addAttribute("idArray", idArray);
model.addAttribute("firstNameArray", firstNameArray);
model.addAttribute("lastNameArray", lastNameArray);
model.addAttribute("patientLinkArray", patientLinkArray);
```

Der Rückgabewert entspricht dem Namen des HTML-Templates, das angezeigt werden soll. Im Falle eines Fehlers die Seite `error`, ansonsten die Seite `patientList`.

```
catch (ClientProtocolException e) {
        e.printStackTrace();
        // When an exception occurs, the Thymeleaf page error is displayed.
          return "error";
      } catch (IOException e) {
        e.printStackTrace();
        // When an exception occurs, the Thymeleaf page error is displayed.
          return "error";
      }
      return "patientList";
    }
}
```

Die HTML-Seiten erstellt man im Ordner `src/main/resources/templates`.

Innerhalb der HTML-Seite kann man dann auf die Objekte zugreifen, die man zuvor dem `Model`-Objekt hinzugefügt hat.

```
<table>
      <tr>
            <th>Id</th>
            <th>Vorname</th>
            <th>Nachname</th>
      </tr>
      <tr th:each="i : ${#numbers.sequence( 0, idArray.size()-1, 1)}">
            <td th:text="${idArray.get(i)} " />
            <td th:text="${firstNameArray.get(i)} " />
            <td th:text="${lastNameArray.get(i)} " />
            <td>
            <form th:action="@{/patient}" method="get">
                  <input type="hidden" name="URI"
th:value="${patientLinkArray.get(i)}" />

                  <button type="submit">Zum Patienten</button>
            </form>
      </td>
</tr>
</table>
```

Mit `th:each` kann man über eine Map oder ein Array iterieren. Hier wird es als For-Schleife verwendet. Über `th:text` kann man einen Wert als Text anzeigen lassen. Mit dem $-Symbol kann man auf die Objekte und Daten im `Model`-Objekt zugreifen. Die Schaltfläche zum Patienten wird über eine Form gelöst. Über `input type="hidden"` lässt sich eine Variable an die URL anhängen. Der Name der Variable ist in diesem Fall „URI“ und der Wert der Variable ist die Base64 codierte URI für den Patienten.

Die Seite, die Thymeleaf erstellt, enthält dann die korrekten Daten.

```
<table>
	<tr>
		<th>Id</th>
		<th>Vorname</th>
		<th>Nachname</th>
	</tr>
	<tr>
		<td >1</td>
		<td >Andreas</td>
		<td >Heese</td>
		<td>
		<form action="/patient" method="get">
			<input type="hidden" name="URI"
value="aHR0cDovL2xvY2FsaG9zdDo4MDgwL3BhdGllbnQvMQ==" />

			<button type="submit">Zum Patienten</button>
		</form>
		</td>
	</tr>
	<tr>
		<td >2</td>
		<td >Maik</td>
		<td >Hansen</td>
		<td>
		<form action="/patient" method="get">
			<input type="hidden" name="URI"
value="aHR0cDovL2xvY2FsaG9zdDo4MDgwL3BhdGllbnQvMg==" />

			<button type="submit">Zum Patienten</button>
		</form>
		</td>
	/tr>
</table>
```

Im Browser angezeigt wird es dann folgendermaßen.

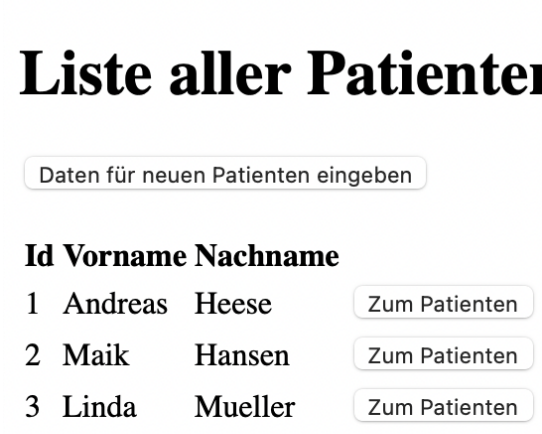

Abbildung 9 Liste aller Patienten

Für alle GET-Anfragen gibt es eine Klasse `ThymeleafPageClient`. Diese unterschiedet sich insofern von der Klasse `PatientListClient`, dass die URL für das GET-Mapping einen Platzhalter für die entsprechende Thymeleaf-Seite enthält.

```
@GetMapping("/{page}")
    public String thymeleafPage(@PathVariable String page,
          @RequestParam(value = "URI", required = true) String
hateoasURI64, Model model)
```

Außerdem kann diese Methode einen Parameter namens „URI“ empfangen, die die Base64 codierte Ressource auf dem REST-Server enthält.

Die Ressource wird wieder decodiert und dann über eine HTTP-Anfrage aufgerufen.

```
// Decode the parameter from Base64 back to a String, which represents the
// URI resource
      byte[] decodeByte = Base64.getUrlDecoder().decode(hateoasURI64);
      String hateoasURI = new String(decodeByte, StandardCharsets.UTF_8);

      try {
          // Create HTTP Client
          HttpClient httpClient = HttpClientBuilder.create().build();

          // Create new PostRequest with aforementioned URL
          HttpGet getRequest = new HttpGet(hateoasURI);

          // Execute your request and catch response
          HttpResponse response = httpClient.execute(getRequest);
```

Anschließend wird wieder aus der Antwort ein JSON-Objekt erzeugt und zusammen mit den einzelnen Links dem `Model`-Objekt hinzugefügt.

In dem HTML-Template „Patient“ kann man mit `th:each` über die Links iterieren und jeden Link einzeln auswerten. Die Form enthält diesmal `th:if`, worüber man abfragen kann, ob eine bestimmte Relation in den Links vorhanden ist. Wenn dies der Fall ist, wird die Form ganz normal erstellt, wenn nicht wird die Form gar nicht erst im HTML-Code eingefügt.

```
<table>
      <tr th:each="link: ${links}">
            <form th:if="${link.key.equals('bloodPressure')}"
      th:action="@{${link.value[0]}}" method="get">
            <input type="hidden" name="URI" th:value="${link.value[1]}" />

                  <button type="submit">Blutdruck</button>
            </form>

            <form th:if="${link.key.equals('appointment')}"
th:action="@{${link.value[0]}}" method="get">
                  <input type="hidden" name="URI"
th:value="${link.value[1]}" />

                  <button type="submit">Termin</button>
            </form>
      </t>
</table>
```

Alle anderen Anfragen wie POST und GET werden in separaten Klassen behandelt. Für das Erstellen eines Patienten kann man die Daten innerhalb einer Form über Eingabefelder eingeben und diese werden dann ausgelesen und ebenfalls als URL-Variablen mitgesendet.

```
<table>
      <tr th:each="link: ${links}">
            <form th:if="${link.key.equals('addNewPatient')}"
th:action="@{${link.value[0]}}" method="post">
            <table>
                  <tr>
                        <td>
                              <label for="Id">Id :</label>
                        </td>
                        <td>
                              <input type="text" name="id" id="id"/>
                        </td>
                        </tr>
                  <tr>
                        <td>
                              <label for="firstName">Vorname :</label>
                        </td>
                        <td>
                              <input type="text" name="firstName"
                              id="firstName"/>
                        </td>
                  </tr>
                  <tr>
                        <td>
                              <label for="lastName">Nachname :</label>
                        </td>
                        <td>
                              <input type="text" name="lastName"
                              id="lastName"/>
                        </td>
                  </tr>
            </table>
            <input type="hidden" name="URI" th:value="${link.value[1]}" />

                  <button type="submit">Neuen Patienten hinzufügen</button>
            </form>
      </tr>
 </table>
```

Für die Anfrage an den Server erstellt man aus den einzelnen Parametern für den Patienten ein JSON-Objekt und erzeugt anschließend einen JSON-String daraus, den man als `StringEntity` der Anfrage POST-Anfrage hinzufügt. Die `StringEntity` entspricht in dem Fall dem `RequestBody`.

```
@PostMapping("/addNewPatient")
public String addNewPatient(
@RequestParam int id, @RequestParam String firstName, @RequestParam String
lastName, @RequestParam String birthDate, @RequestParam String street,
@RequestParam String city, @RequestParam String zipCode, @RequestParam(value =
"URI", required = true) String hateoasURI64, Model model) {
      try {
          // Decode the parameter from Base64 back to a String,
          //which represents the URI resource
          byte[] decodeByte = Base64.getUrlDecoder().decode(hateoasURI64);
          String hateoasURI = new String(decodeByte, StandardCharsets.UTF_8);

          // Create HTTP Client
          HttpClient httpClient = HttpClientBuilder.create().build();

          // Create new postRequest with aforementioned URL
          HttpPost postRequest = new HttpPost(hateoasURI);

          JSONObject joPost = new JSONObject();

          joPost.put("id", id);
          joPost.put("firstName", firstName);
          joPost.put("lastName", lastName);
          joPost.put("birthDate", birthDate);
          joPost.put("street", street);
          joPost.put("city", city);
          joPost.put("zipCode", zipCode);

          // Create a StringEntity object from a JSON object that
          //can be added to a PostRequest as RequestBody
          StringEntity entity = new StringEntity(joPost.toString(),
          ContentType.APPLICATION_JSON);

          postRequest.addHeader("content-type",
          ContentType.APPLICATION_JSON.toString());

          // Sets the aforementioned StringEntity as RequestBody
          postRequest.setEntity(entity);
```

Für das Löschen eines Patienten erzeugt man eine `DeleteRequest` und kann somit die Ressource einfach löschen.

```
    @PostMapping("/deletePatient")
    public String deletePatient(@RequestParam(value = "URI", required =
true) String hateoasURI64, Model model) {
      try {
          // Decode the parameter from Base64 back to a String,
          //which represents the URI resource
          byte[] decodeByte = Base64.getUrlDecoder().decode(hateoasURI64);
          String hateoasURI = new String(decodeByte,
          StandardCharsets.UTF_8);

          // Create HTTP Client
          HttpClient httpClient = HttpClientBuilder.create().build();

          // Create a new deleteRequest
          HttpDelete deleteRequest = new HttpDelete(hateoasURI);

          // Execute the request
          httpClient.execute(deleteRequest);
      } catch (ClientProtocolException e) {
          e.printStackTrace();
          // When an exception occurs, the Thymeleaf page error is
displayed.
          return "error";
      } catch (IOException e) {
          e.printStackTrace();
          // When an exception occurs, the Thymeleaf page error is
displayed.
          return "error";

      }
      return "redirect:/patientlist";

    }
```

In diesem Fall wird keine Seite an sich zurückgegeben, sondern eine Umleitung zu der URL `/patientList`. Da der Patient gelöscht wurde kann man keinen Patienten mehr anzeigen und wird umgeleitet zu der Startseite.

Das Testen des Thymeleaf-Servers

Das Testen des Thymeleaf-Servers verläuft ähnlich wie bei dem REST-Server. Man nutzt ebenfalls ein Mock-Objekt und die einzelnen Test-Methoden sehen auch sehr ähnlich aus.

Um zum Beispiel zu testen, ob erfolgreich ein neuer Patient erstellt wurde, kann man überprüfen, ob die Weiterleitung auf eine andere Seite funktioniert und ob die Größe des `idArrays` stimmt.

```
@Test
 public void addNewPatientClient() throws Exception {
mockMvc.perform(MockMvcRequestBuilders.post("/addNewPatient")
.contentType(MediaType.APPLICATION_JSON_VALUE)
.param("id", "4").param("firstName", "Monika")
.param("lastName", "Meier")
.param("birthDate", "01.09.1993")
.param("street", "Ringstrasse 525")
.param("city", "Hamburg")
.param("zipCode", "22767")
.param("URI", addNewPatientLink64))
.andExpect(MockMvcResultMatchers.status().is3xxRedirection())
.andExpect(MockMvcResultMatchers.redirectedUrl("/patientlist"));


mockMvc.perform(MockMvcRequestBuilders.get("/patientlist").contentType(Medi
aType.APPLICATION_JSON_VALUE))
.andExpect(MockMvcResultMatchers.status().isOk())
.andExpect(MockMvcResultMatchers.view().name("patientList"))
.andExpect(MockMvcResultMatchers.model()
      .attribute("idArray", Matchers.hasSize(3)));
}
```

Man muss allerdings beachten, dass man bei der automatischen Konfiguration durch Spring-Boot zwei Klassen für die Erstellung von JSON-Objekten hat. Deswegen muss man in der `pom.xml` eine von beiden Klassen exkludieren.

```
<dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
      <exclusions>
            <exclusion>
                  <groupId>com.vaadin.external.google</groupId>
                  <artifactId>android-json</artifactId>
            </exclusion>
      </exclusions>
</dependency>
```

Vergleich mit anderen Projekten

Auf GitHub gibt es bereits ein Beispielprojekt für einen HATEOAS-Server inklusive eines Thymeleaf-Servers als Client.

Hier beispielhaft gezeigt eine GET-Methode. Anstand das man die URI über die URL-Variable von Seite zu Seite mitschickt, wird hier innerhalb der Methode über ein `Traverson`-Objekt der Link zur entsprechenden Ressource auf dem Server gesucht, in dem man den Link zu `employee` folgt. Dadurch erhält man zur Laufzeit vom REST-Server den Link zu der Ressource.

```
/**
 * Get a listing of ALL {@link Employee}s by querying the remote services'
root URI, and then
 * "hopping" to the {@literal employees} rel.
 *
 * NOTE: Also create a form-backed {@link Employee} object to allow
creating a new entry with
 * the Thymeleaf template.
 *
 * @param model
 * @return
 * @throws URISyntaxException
 */
@GetMapping
public String index(Model model) throws URISyntaxException {

      Traverson client = new Traverson(new URI(REMOTE_SERVICE_ROOT_URI),
MediaTypes.HAL_JSON);
      CollectionModel<EntityModel<Employee>> employees = client
            .follow("employees")
            .toObject(new ResourcesType<EntityModel<Employee>>(){});

      model.addAttribute("employee", new Employee());
      model.addAttribute("employees", employees);

      return "index";
}
```

[33]

Probleme bei der Entwicklung

Bei der Klasse `BloodPressure` wurde ein anderer Ansatz versucht als bei den restlichen Klassen. Hier gibt es nur ein `BloodPressure`-Objekt, das eine ArrayList aus `BloodPressureValue`-Objekten enthält, die allerdings keine eigene Ressource sind. Dadurch muss für das Löschen der Zeitpunkt der Messung verwendet werden, da diese auch eindeutig für jede Messung ist.

```
<table style="width:35%">
	<tr>
		<th>Zeitpunkt der Messung</th>
		<th>Systole in mmHg</th>
		<th>Diastole in mmHg</th>
	</tr>
	<tr th:each="joArray: ${jo.get('bloodPressureValueArray')}">
		<p th:each="joMap: ${joArray}">
		<td th:text="${joMap.get('dateAndTimeString')} " />
		<td th:text="${joMap.get('systole')} " />
		<td th:text="${joMap.get('diastole')} " />
		<td
			<form th:if="${links.containsKey('deleteBloodPressure')}"
		th:action="@{'http://localhost:8081/deleteBloodPressure'}"
		method="post">

			<input type="hidden" name="dateAndTime"
		th:value="${joMap.get('dateAndTimeString')} "/>

		<div th:object="${links.get('deleteBloodPressure')}" >
			<input type="hidden" name="URI" th:value="*{[1]} " />


				<button type="submit">Löschen</button>
			</form>
		</td>
		</p>
	</tr>
</table>
```

Man muss also einmal über das JSON-Array `bloodPressureValueArray` iterieren und dann innerhalb des Arrays über die JSON-Map aus den einzelne Blutdruckdaten iterieren. Um innerhalb der Form auf das Link-Objekt zugreifen zu können, kann man innerhalb des Tags `<div>` ein Thymeleaf-Objekt erzeugen, auf das man dann mit dem *-Symbol zugreifen kann. [34]

In der Methode für das Löschen des Blutdruckwertes muss man dann statt `DeleteRequest` `PostRequest` benutzen, da man an einen `DeleteRequest` keinen `ResponseBody` hinzufügen kann.

```
// Create a new postRequest. Although deleteRequest exists, postRequest
// is used here because we need to add an entity object to it and
// that only works with a postRequest.
HttpPost postRequest = new HttpPost(hateoasURI);

JSONObject joPost = new JSONObject();
joPost.put("dateAndTime", dateAndTime);

// Create a StringEntity object from a JSON object that can be added to a
// PostRequest as RequestBody
StringEntity entity = new StringEntity(joPost.toString(),
ContentType.APPLICATION_JSON);

postRequest.addHeader("content-type",
ContentType.APPLICATION_JSON.toString());

postRequest.setEntity(entity);
```

Im Browser kann man allerdings die Schaltfläche zum Löschen des Blutdruckwerts nicht anklicken. Auch wird der Link für die form-action im Quellcode im Browser nicht farbig markiert, als würde der Browser den Link nicht erkennen. Dieses Problem konnte leider nicht behoben werden. Vermeiden lassen würde sich das Problem wahrscheinlich, in dem aus den einzelnen Blutdruckwerten auch jeweils eigene Ressourcen erstellt und man dann einfach, wie bei den Patienten, die Ressource löschen kann.

Ein weiteres Problem ist, wenn man keine oder falsche Daten beim Erstellen eines Termins eingibt, zwar die Fehlerseite angezeigt wird, allerdings trotzdem beim Patienten hinterlegt ist, dass er einen Termin hat, was zu Fehlern führt, wenn man den Termin eines Patienten anzeigen lassen will. Auch dieses Problem konnte nicht gelöst werden.

Schlussteil

Eigene Erfahrungen während der Entwicklung

Abschließend möchte ich die Pro- und Kontraargumente zu HATEOAS mit meiner eigenen Erfahrung während der Entwicklung vergleichen.

Erkundbare API

Das empfand ich als sehr angenehm. Weniger, weil ich mich in einen bereits vorhanden REST-Server einarbeiten musste, sondern es hat mir bei der Erstellung des Servers geholfen, da ich mir so sehr einfach den REST-Server vorstellen konnte und dadurch auch einfacher erstellen konnte. Und ich kann auch nachvollzeihen, warum es so einfacher ist, sich in einen Server einzuarbeiten.

Einfache Client Logik

Diesen Punkt kann ich auch bestätigen. Dadurch, dass man einfach nur überprüfen muss, ob ein Link vorhanden ist um daraufhin zum Beispiel eine Schaltfläche anzuzeigen oder nicht, sorgt dafür, dass man sich die gesamte Logik auf Client-Seite spart. Durch die lose Kopplung konnte man auch gut den REST-Server getrennt von dem Client entwickeln. Allerdings habe ich am Anfang der Entwicklung keine Vorlage für einen REST-Client gefunden und es fiel mir schwer auf die Idee mit den Base64 codierten URIs zu kommen und diese zu implementieren. Im Nachhinein würde ich aber sagen, dass das durchaus eine sinnvolle Lösung ist, um die lose Kopplung zu ermöglichen.

Wartbarkeit

Ich habe während der Entwicklung auch Ressourcen auf dem REST-Server geändert und ich war froh darüber, dass der Client davon unberührt blieb.

Keine etablierten Standards

Diesen Punkt kann ich persönlich für mein Projekt nicht nachvollziehen, da es für Java mit Spring ein sehr umfangreiches Framework gibt, das einem viel Arbeit bei der Erstellung des REST-Servers, des Thymeleaf-Servers und der Tests abnimmt.

Mehr Bandbreite

Diesen Punkt kann ich durchaus verstehen, da durch die Links sehr viel Overhead erzeugt wird. Allerdings hat mich das bei diesem kleinen Projekt in der Entwicklung nicht beeinflusst.

Mehr Aufwand

Diesen Punkt kann ich teilweise nachvollziehen. Natürlich hat man den Mehraufwand, dass man immer alle möglichen Links an die Antwort anhängen muss und wenn man zum Beispiel eine neue Ressource erstellt muss man drauf achten, auch diese an alle relevanten Objekte anzuhängen. Allerdings würde ich behaupten, dass der Aufwand ohne HATEOAS auf Client-Seite alles anpassen zu müssen ähnlich hoch sein wird. Ich würde also sagen es ist nicht mehr Aufwand, allerdings auch nicht weniger, vorausgesetzt man hat von Anfang an HATEOAS genutzt. Ein Projekt auf HATEOAS umzustellen, lohnt sich wahrscheinlich nur in den wenigstens Fällen.

Zusammenfassung und Ausblick

Abschließend lässt sich sagen, dass HATEAS eine sehr spannende Entwicklung ist, die wahnsinnig viel Potential hat. Auch die Meinung von vielen Entwicklern in der Branche deckt sich damit. Allerdings fällt auf, dass sich das nicht mit der Anzahl an Unternehmen deckt, die HATEOAS nutzen. Das kann daran liegen, dass die Unternehmen nicht öffentlich kommunizieren, welche Technologien sie verwenden, aber vielleicht ist es in der Praxis wirklich noch nicht so oft vertreten. Auch wenn man sich Kommentare in Foren durchließt, ist die Stimmung auch eher gegen HATEOAS, weil es sich nicht lohnen würde oder es zu kompliziert sei. Allerdings weiß man bei kurzen anonymen Kommentaren meistens nicht, warum derjenige diese Meinung vertritt und inwiefern er auch schon mit HATEOAS gearbeitet hat. Auch wenn HATEOAS auch Nachteile hat, sind einige davon auch nur der geringeren Verbreitung geschuldet. Das Potenzial ist auf jeden Fall vorhanden, dass HATEOAS nicht nur eine weitere Art ist, wie man einen REST-Server aufbauen kann, sondern der Standard für REST-Server wird.

Anlagen

Inhalt der beigefügten CD

- Die Bachelorarbeit als Word- und als PDF-Dokument
- Der Quellcode zum Projekt „PatientAdministration“ inklusive aller Testklassen und ausführbarer Jar-Datei
- Der Quellcode zum Projekt „PatientAdministrationClient“ inklusive aller Testklassen, HMTL-Templates und ausführbarer Jar-Datei
- Alle genutzten Internetquellen als PDF
- Alle genutzten Buchquellen als PDF
- Alle selbsterstellten Bildschirmfotos als PNG

Literaturverzeichnis

[1] „Spring,“ VMware, 2022. [Online]. Available: https://spring.io. [Zugriff am 25 April 2022].

[2] „HATEOAS Wikipedia,“ Wikipedia, 27 Oktober 2021. [Online]. Available: https://en.wikipedia.org/wiki/HATEOAS. [Zugriff am 25 April 2022].

[3] „Getting Started | Building a Hypermedia-Driven RESTful Web Service,“ VMware, 2022. [Online]. Available: https://spring.io/guides/gs/rest-hateoas/#initial. [Zugriff am 25 April 2022].

[4] A. Bernat und A. Wołowiec , „How to Build Hypermedia API with Spring HATEOAS - Grape Up,“ Grape Up, 2022. [Online]. Available: https://grapeup.com/blog/how-to-build-hypermedia-api-with-spring-hateoas/#. [Zugriff am 25 April 2022].

[5] „Maven Repository: org.springframework.hateoas » spring-hateoas,“ MvnRepository, 2022. [Online]. Available: https://mvnrepository.com/artifact/org.springframework.hateoas/spring-hateoas. [Zugriff am 25 April 2022].

[6] M. Fowler, „Richardson Maturity Model,“ 18 März 2010. [Online]. Available: https://martinfowler.com/articles/richardsonMaturityModel.html. [Zugriff am 25 April 2022].

[7] L. Richardson, „Step towards REST,“ 18 März 2010. [Online]. Available: https://martinfowler.com/articles/images/richardsonMaturityModel/overview.png. [Zugriff am 25 April 2022].

[8] L. Richardson, „Level 1 adds resources,“ 18 März 2010. [Online]. Available: https://martinfowler.com/articles/images/richardsonMaturityModel/level1.png. [Zugriff am 25 April 2022].

[9] L. Richardson, „Level 2 addes HTTP verbs,“ 18 März 2010. [Online]. Available: https://martinfowler.com/articles/images/richardsonMaturityModel/level2.png. [Zugriff am 25 April 2022].

[10] L. Richardson, „Level 3 adds hypermedia controls,“ 18 März 2010. [Online]. Available: https://martinfowler.com/articles/images/richardsonMaturityModel/level3.png. [Zugriff am 25 April 2022].

[11] „Das Unternehmen novomind AG Hamburg,“ novomind AG, 2022. [Online]. Available: https://www.novomind.com/de/unternehmen/. [Zugriff am 25 April 2022].

[12] „HATEOAS | novomind developer documentation,“ novomind AG, 2022. [Online]. Available: https://apps.novomind.com/developer/ishop/rest/hateoas.html. [Zugriff am 25 April 2022].

[13] „Shopsystem für modernen Omnichannel-Commerce - novomind iShop,“ novomind AG, 2022. [Online]. Available: https://www.novomind.com/commerce/novomind-ishop/. [Zugriff am 25 April 2022].

[14] „PIM-System für ihre Produktdaten im Onlineshop - novomind iPIM,“ novomind AG, 2022. [Online]. Available: https://www.novomind.com/de/pim-system/novomind-ipim-pim-software/. [Zugriff am 25 April 2022].

[15] „Using HATEOAS with REST APIs - 3ap Engineering Blog,“ 3ap, 2022. [Online]. Available: https://engineering.3ap.ch/post/using-hateoas-with-rest/. [Zugriff am 25 April 2022].

[16] „Yapeal Review 2022 – New Swiss Digital Bank - The Poor Swiss,“ The Poor Swiss, 2022. [Online]. Available: https://thepoorswiss.com/yapeal-review/. [Zugriff am 25 April 2022].

[17] „PayPal - Über uns - PayPal,“ https://www.paypal.com/de/webapps/mpp/about, 2022. [Online]. [Zugriff am 26 April 2022].

[18] „API responses,“ PayPal, 2022. [Online]. Available: https://developer.paypal.com/api/rest/responses/. [Zugriff am 26 April 2022].

[19] N. Rimmele, „Die besseren REST-Services nutzen Spring HATEOAS,“ Java aktuell, Januar 2021. [Online]. Available: https://backoffice.doag.org/formes/pubfiles/12736401/docs/Publikationen/Java-Aktuell/2021/01-2021/01_2020-Java_aktuell-Nico_Rimmele-Die_besseren_REST-Services_nutzen_Spring_HATEOAS.pdf. [Zugriff am 25 April 2022].

[20] „Stefan Ullrich,“ 2022. [Online]. Available: https://entwickler.de/experten/stefan-ullrich. [Zugriff am 26 April 2022].

[21] S. Ullrich, „Wer REST will, muss mit HATEOAS ernst machen,“ entwickler.de, 2022. [Online]. Available: https://entwickler.de/api/wer-rest-will-muss-mit-hateoas-ernst-machen/. [Zugriff am 26 April 2022].

[22] R. T. Fielding, „Roy T. Fielding,“ 12 April 2021. [Online]. Available: https://roy.gbiv.com. [Zugriff am 26 April 2022].

[23] „REST APIs must be hypertext-driven » Untangled,“ 2011. [Online]. Available: https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven. [Zugriff am 26 April 2022].

[24] „Newest 'rest' Questions - Stack Overflow,“ Stack Exchange Inc, 2022. [Online]. Available: https://stackoverflow.com/questions/tagged/rest. [Zugriff am 26 April 2022].

[25] „Newest 'hateoas' Questions - Stack Overflow,“ Stack Exchange Inc, 2022. [Online]. Available: https://stackoverflow.com/questions/tagged/hateoas?tab=Newest. [Zugriff am 26 April 2022].

[26] „SOA Bits and Ramblings: Selling the benefits of hypermedia in APIs,“ 6 Dezember 2013. [Online]. Available: http://soabits.blogspot.com/2013/12/selling-benefits-of-hypermedia.html. [Zugriff am 25 April 2022].

[27] „HATEOAS - The Good, the Bad and the Ugly,“ Code Scare, 2018. [Online]. Available: https://varunpatil.me/blog/posts/hateoas/. [Zugriff am 26 April 2022].

[28] „Eclipse Downloads | The Eclipse Foundation,“ Eclipse Foundation, 2022. [Online]. Available: https://www.eclipse.org/downloads/. [Zugriff am 26 April 2022].

[29] „Spring Initializr,“ VMware, 2022. [Online]. Available: https://start.spring.io. [Zugriff am 26 April 2022].

[30] „Jackson | Jackson Tutorial - javatpoint,“ Javatpoint, 2021. [Online]. Available: https://www.javatpoint.com/jackson. [Zugriff am 26 April 2022].

[31] W. Golubski, Entwicklung verteilter Anwendungen Mit Spring Boot & Co, Zwickau: Springer Vieweg, 2019.

[32] „Thymeleaf,“ The Thymeleaf Team, [Online]. Available: https://www.thymeleaf.org. [Zugriff am 26 April 2022].

[33] „spring-hateoas-examples/api-evolution at main · spring-projects/spring-hateoas-examples · GitHub,“ 20 Oktober 2020. [Online]. Available: https://github.com/spring-projects/spring-hateoas-examples/tree/main/api-evolution. [Zugriff am 26 April 2022].

[34] „Thymeleaf th/object and Asterisk Syntax *{ },“ o7planning.org, März 2014. [Online]. Available: https://o7planning.org/12385/thymeleaf-th-object-and-asterisk-syntax. [Zugriff am 26 April 2022].

[35] L. Richardson, „An example interaction at Level 0,“ 18 März 2010. [Online]. Available: https://martinfowler.com/articles/images/richardsonMaturityModel/level0.png. [Zugriff am 25 April 2022].

Abbildungsverzeichnis

Selbstständigkeitserklärung

Selbstständigkeitserklärung gem. § 14 Absatz 5 BPO

Hiermit versichere ich, Andreas Heese, dass ich die vorliegende Bachelorarbeit mit dem Titel

Analyse des Spring-Frameworks Hateoas und Entwicklung einer Beispielapplikation

selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

.....	.....
Ort, Datum	Unterschrift