RESTlos glücklich

# Hypermedia as the Engine of Application State – Spring Hateoas

**Phillip Ghadir**

*Das kürzlich veröffentlichte Spring Hateoas vereinfacht das Verlinken von Ressource-Repräsentationen und bietet Mechanismen zur Link-Erzeugung und zum Reverse-Routing. Dieser Artikel stellt vor, wie diese Konzepte mit Spring MVC zusammen arbeiten und die Entwicklung von Webanwendungen erleichtern.*

## HATEOAS und verlinkte Ressourcen

Wozu man ein zusätzliches Framework braucht, wenn es doch bereits patente Frameworks wie zum Beispiel Spring MVC [SpringMVC] gibt?

Wer hat nicht schon mal eine Webanwendung mit Spring-MVC gebaut und sie REST-konform (oder RESTful) genannt? Es gibt eine faszinierende Menge auch an jüngeren Programmierschnittstellen, die RESTful genannt werden, obwohl sie gegen eine entscheidende Architekturregel von REST verstoßen – den Hypertext-Constraint*: Der Client löst Zustandsänderungen aus, indem er Links folgt. In seiner Dissertation [Field00] nennt Roy Fielding dies *„Hypermedia as the engine of application state"*. Andere kürzen dies häufig mit HATEOAS ab.

Eine REST-konforme Anwendung ermöglicht beliebigen Clients die Interaktion so, wie ein Mensch Webseiten über einen Browser bedient: Nach dem Laden der Seite wählt er aus den dargestellten Informationen die relevanten aus und klickt den Link zur nächsten Seite/Aktion. Die Ressource-Repräsentationen enthalten Links, die dem Client die Menge sinnvoller Transitionen anbieten. Ein Client kann dann mit einer geeigneten Beschriftung der Links entscheiden, welche Zustandstransition ausgeführt werden soll.

Eine REST-Schnittstelle, die nicht auf dem Prinzip verlinkter Ressourcen basiert, ist keine REST-Schnittstelle. Das schreibt Roy Fielding in [Field08].

Es ist gang und gäbe, Links parallel mit Beschreibungen für den Menschen und die Maschine zu versehen. Während sich ein Mensch auf die zum Link gehörende (textuelle) Beschreibung konzentriert, verlässt sich eine Maschine eher auf das `rel`-Attribut. In beiden Fällen braucht den Client die URI selbst nicht zu interessieren.

Gängige Abstraktionen für Softwarebausteine, wie zum Beispiel Aggregat, Entität, persistentes Objekt oder Domänenklasse, bieten keine eigenen Konzepte an, um Ressource-Repräsentationen mit Links zu versehen, die sich automatisch aus der Implementierung ergeben. Sollen der Hypertext-Constraint nicht vernachlässigt werden und deshalb Links verwaltet und transportiert werden, braucht es entweder eine angemessene Abstraktion wie zum Beispiel `Ressource-Repräsentation` oder Links fließen direkt ins Domänenmodell ein.

Darüber hinaus fehlt in verschiedenen Back-End-Frameworks eine geeignete Abstraktion für das Erzeugen von Links. Selbst wenn ein Framework wie JAX-RS über URI-Templates verfügt und es erlaubt, mit Parametern versehene URIs mit Werten zu belegen und dadurch den konkreten URI zu erhalten, fehlt die Kopplung an die Struktur der Implementierung. Dies wird allgemein als Reverse-Routing bezeichnet.

Hier war vor Spring Hateoas immer die redundante Pflege von Links nötig: Zunächst konfiguriert man das URI-Mapping per RequestMapping-Annotation und dann baut man die URIs per String-Konkatenation zusammen.

Spring Hateoas ergänzt nun Spring MVC um die Möglichkeit, verlinkte Ressourcen zu realisieren, ohne dabei redundant die Zusammenhänge zwischen Implementierung und Links zu pflegen. Es ergänzt damit Spring MVC und bettet sich problemlos in eine Spring MVC-Anwendung ein. Abbildung 1 stellt die Abhängigkeiten der beteiligten Komponenten dar. Wer Spring MVC nutzt, hat praktisch alles, was Spring Hateoas noch benötigen könnte.

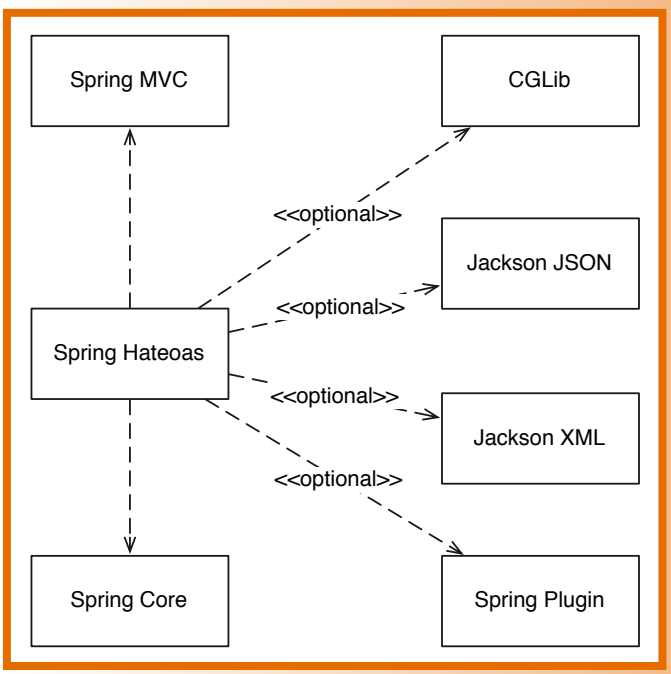


Abb. 1: Hateoas im Kontext von Spring MVC Web

*Heute ist der Begriff Hypermedia-Prinzip gebräuchlicher als Hypertext-Constraint. Für diesen Artikel ist letzterer aber treffender.

## Spring Hateoas im Projekt

Um Spring Hateoas im eigenen Projekt nutzen zu können, bindet man einfach das Java-Archiv spring-hateoas-<version>.jar ins Projekt ein. Maven-Nutzer fügen folgende Abhängigkeit in die pom.xml ein:

```
<dependency>
  <groupId>org.springframework.hateoas</groupId>
  <artifactId>spring-hateoas</artifactId>
  <version>0.4.0.RELEASE</version>
</dependency>
```

Das genügt bereits an Projektabhängigkeiten, um Links in die Ressource-Repräsentationen aufzunehmen.

## Ressource-Repräsentationen verlinken

Wenn eine Domänenklasse von **ResourceSupport** ableitet, erbt sie Methoden, mit denen Links zur Ressource hinzugefügt werden können. In diesem Kontext sind Links Tupel, bestehend aus einem Relationsnamen (dem **rel**-Attribut) und einer URI (dem **href**-Attribut).

```
public class KundenAkte extends ResourceSupport {
  public String kundenNr;
  public String nachname;
  public String vorname;
  public KundenArt kundenArt;

  public KundenAkte() {
  }
  public KundenAkte(Long kundenNr, String nachname,
  String vorname, KundenArt kundenArt) {
    // initialisiere alle Attribute ...
  }
  @Override
  public Link getId() { /* ... */ }
}
```

Listing 1: Beispiel einer Domänenklasse KundenAkte, die von ResourceSupport erbt

Das **extends ResourceSupport** in Zeile 1 von Listing 1 bewirkt, dass KundenAkten Links enthalten können. **ResourceSupport** definiert Methoden zum Hinzufügen und Abfragen von Links.

Angenommen, es gibt bereits zwei Link-Instanzen, referenziert durch die Attribute **theLinkToSelf** und **theLinkToTheDetails**, dann können wir diese einer KundenAkte beispielsweise so hinzufügen:

```
KundenAkte akte = new KundenAkte(4713L, "Potter",
  "Harry", KundenArt.VIP);
akte.add( theLinkToSelf );                  // Link für Relation self
akte.add( theLinkToTheDetails, "details" ); // Link zu den Details
```

Wird nun die KundenAkte über einen JSON-Marshaller ausgegeben, enthält das JSON automatisch die Liste mit den Links wie in Listing 2 dargestellt. Die **href**-Attribute zeigen Informationen zur Laufzeitumgebung: Der Server läuft auf **localhost:8080**. Die Webanwendung heißt **smh** (kurz für spring-mvc-hateoas-Beispiel). Der restliche URI-Pfad wird jeweils so konstruiert, wie im Folgenden beschrieben.

```
{
  "links":[
    {
      "rel":"details",
      "href":"http://localhost:8080/smh/customers/4713/details"
    },
    {
      "rel":"self",
      "href":"http://localhost:8080/smh/customers/4713"
    }],
  "kundenNr":4713,
  "nachname":"Potter",
  "vorname":"Harry",
  "kundenArt":"VIP"
}
```

Listing 2: JSON-Repräsentation der KundenAkte 4713

## Links definieren

Gelungen ist in Spring Hateoas das Erstellen von Links: Während andere Frameworks entweder mangelhafte Abstraktionen für die Konstruktion von Links anbieten, oder aber es nicht schaffen, die Informationen für das URI-Mapping an genau einer Stelle zu kapseln, ohne es beim Konstruieren von Links noch einmal redundant zu pflegen, erlaubt die Kombination von Spring MVC und Spring Hateoas die Link-Konstruktion wahlweise auf Basis von URI-Templates oder auch auf Basis der Implementierungsstruktur. Werden Links auf Basis der Implementierungsstruktur konstruiert, spricht man von Reverse-Routing.

Zentrales Element ist hierbei die Klasse **ControllerLinkBuilder** aus dem Hateoas-Framework. Sie bietet verschiedene statische Methoden an, mit denen wir Links konstruieren können. Ein Link auf die Methode **getAkte** im **KundenController** (siehe Listing 3) kann man zum Beispiel einfach durch

```
linkTo(
  methodOn(KundenController.class )
    .getAkte( 1234L )
  ).withRel( "kundeDetail" );
```

zusammen bauen. Die Methode **linkTo** der Klasse **ControllerLinkBuilder** wird dazu vorab statisch importiert.

```
@Controller
@RequestMapping("/customers")
public class KundenController {
  @RequestMapping(value="/{id}", method=RequestMethod.GET,
     produces="application/json")
  public @ResponseBody KundenAkte getAkte(@PathVariable Long id) {
    /* ... */
  }
}
```

Listing 3: Auszug aus der Klasse KundenController, mit einer mit @RequestMapping annotierten Methode

Wir sehen hier, wie das Framework mit dem Aufruf von **methodOn( KundenController.class )** einen Proxy erzeugt, der dann direkt mit den Parametern aufgerufen wird, die wir im Link codieren möchten. Damit wird dann ein Link erzeugt, in dem per **withRel()**-Aufruf die Relation „kundeDetail“ gesetzt wird.

Der Aufrufparameter für **getAkte** – die Zahl 1234L – kann beliebig abhängig von der Signatur der Methode sein, zu der verlinkt werden soll. Standardmäßig wird auf den Argumenten **toString()** aufgerufen. Wenn ein Link auf den Controller genügt, schreibt man einfach:

```
linkTo(KundenController.class ).withSelfRel();
```

Das genügt, um einen Link mit Default-Relation zu erzeugen. Man braucht ihn nur noch zuzuweisen oder auszugeben.

Verfügt der Controller über ein parametrisiertes **RequestMapping**, wie zum Beispiel in Listing 4 dargestellt, benötigt man beim Aufruf von **linkTo** auch noch die erforderlichen Argumente:

```
linkTo( KundenAdressenController.class, kunde );
```

Ein kleiner Hinweis: Beim Erzeugen des Links werden nur die mit `@PathVariable` und `@RequestParam` annotierten Parameter ausgewertet und im Link verwertet.

```
@Controller
@RequestMapping("/customers/{id}/addresses")
public class KundenAdressenController {
  // ....
}
```

Listing 4: KundenAdressenController mit parametrisiertem RequestMapping

## Links erweitern

Häufig verwendet man ein URI-Schema, bei dem auf ein Detail zugegriffen wird, in dem an die URI der übergeordneten Ressource ein weiteres URI-Segment mit Slash angefügt wird. Das Anhängen von URI-Segmenten unterstützt Spring Hateoas direkt. Die Methode `slash` hängt ein Segment an den Link an. Für Instanzen, die `Identifiable` implementieren, ruft `slash` die Methode `getId()` auf dem Parameter auf und hängt diesen an. Für alle anderen Objekte wird `toString()` aufgerufen und das Ergebnis angehängt.

```
Link kundenDetailLink = kundenlisteLink.slash( kunde );
```

Angenommen, der Aufrufparameter `kunde` referenziert eine Instanz von `Kunde`, die `Identifiable` implementiert, wie in Listing 5 dargestellt. Dann ergäbe der obige Aufruf ein `http://localhost:8080/smh/customers/3`.

```
public class Kunde implements Identifiable<Long> {
  private Long id;

  public Kunde( Long id ) {
    this.id = id;
  }
  public Long getId() { return id; }
}
```

Listing 5: Domänenklasse Kunde implementiert Identifiable

Dank der Fluent-Programmierschnittstelle lassen sich beliebig verschachtelte Links sehr einfach konstruieren. Dies ist für Links innerhalb komplexer zusammengesetzter Ressourcen-Graphen sinnvoll:

```
res.add(
  linkTo( KundenController.class )
    .slash( kunde )
    .slash( "addresses" )
    .slash( kunde.getHauptadresse() )
    .withRel( "mainAddress" )
);
```

Die Verwendung von `slash` erfordert das Wissen um die URI-Struktur und bricht daher die Kapselung des URI-Mappings. Es bietet sich daher an, die Konstruktion von Links innerhalb des eigenen Systems nach fachlichen Gesichtspunkten geeignet zu kapseln. Für einfache Fälle bietet Spring Hateoas aber eine Alternative: `EntityLinks`, die sich auf die interne Softwarestruktur verlassen und die URI-Struktur nicht zu kennen brauchen.

## Zusammenspiel mit Spring MVC-Controllern

Neben der Möglichkeit, sich in den Link-Buildern direkt auf das konfigurierte Mapping an den Controllern und Controller-Methoden zu beziehen, gibt es eine weitere Option: Spring MVC-Controller können mit `@ExposesResourceFor( ... )` annotiert werden, um zu zeigen, dass sie für die Verwaltung von Ressourcen eines bestimmten Typs zuständig sind. Solange es für jede Domänenklasse nur genau einen Controller gibt, der für deren Verwaltung zuständig ist, kann man Links bauen, ohne den Controller zu kennen, der die Ressource exponiert.

Um nun die generischen Methoden verwenden zu können, muss man `EntityLinks` in der Spring MVC-Konfiguration aktivieren. Dazu annotiert man die Konfigurationsklasse mit der Annotation `@EnableEntityLinks`. Damit werden alle mit `@Controller` annotierten Klassen nach der Annotation `@ExposesResourceFor` durchsucht und registriert, sodass dafür sogenannte `EntityLinks` zur Verfügung stehen. Der Aufruf von

```
// in der Controller-Klasse deklariert
@Autowired EntityLinks entityLinks;

// irgendwo in einer Methode der Controller-Klasse
Link link = entityLinks.linkToSingleResource(Adresse.class, adressId);
```

erzeugt dann Links auf die Ressource-Adresse. Der URI setzt sich dann zusammen aus dem URI der Webanwendung, dem URI-Segment, das sich aus dem RequestMapping des Controllers ergibt, der mit `@ExposesResourceFor( Adresse.class )` annotiert ist, und der `adressId`:

```
http://localhost:8080/smh/addresses/7021
```

Die Registrierung von Controllern per `@ExposesResourceFor` funktioniert nur, wenn für eine Ressource nur jeweils ein Controller annotiert wird.

## Weiterführende Informationen

Spring Hateoas ist übersichtlich (s. [Hateoas]). In dem Readme werden auch die hier nicht weiter beschriebenen Bestandteile vorgestellt: die Abbildung auf XML oder der `ResourceAssemblerSupport`. Resource-Assembler dienen dazu, für Domänenobjekte Ressourcen bzw. Collection-Ressourcen zu erzeugen.

Das separate Beispiel-Projekt [RestBucks] zeigt, wie Spring MVC und Spring Hateoas zusammenspielen können. Im Package `org.springsource.restbucks.payment.web` ist auch die Abwägung zwischen dem Kapseln der Link-Erzeugung in einer eigenen Klasse und der direkten Verwendung der Klasse `EntityLinks` zu erkennen. Die Klasse `PaymentLinks` kapselt das Erzeugen von Links. Dennoch verwendet der `PaymentController` teilweise auch direkt EntityLinks, um einfache Links zu erzeugen.

## Fazit

Spring Hateoas ist eine Ergänzung zu Spring MVC, mit der das Realisieren von vernetzten Ressourcen wartungsfreundlicher werden kann. Trotz des noch frühen Stadiums verfügt es bereits über Abstraktionen für den manuellen Zusammenbau von URIs sowie die automatische Konstruktion von URIs auf Basis von Annotationen innerhalb des Back-Ends.

Das Zusammenspiel mit JAX-RS wurde hier nicht weiter betrachtet, sollte aber problemlos funktionieren. Dadurch stünde dann neben den URI-Erzeugungsmechanismen aus Spring MVC und Hateoas auch noch der JAX-RS-URI-Builder zur Verfügung. Damit wird sowohl das Definieren von URIs als auch das Verlinken von Ressourcen einfacher.

Derzeit ist die Version 0.4.0 verfügbar. Auch in 2013 ist Fieldings Kritik an vielen REST-konformen Schnittstellen gültig (s. [Field08]). Mit Spring Hateoas wird es nun leichter, den Hypertext-Constraint des REST-Stils umzusetzen.

## Literatur und Links

**[Field00]** R. T. Fielding, Architectural Styles and the Design of Network-based Software Architectures, Dissertation an der University of California, Irvine, 2000,
`http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`
**[Field08]** R. T. Fielding, REST APIs must be hypertext-driven,
`http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven`
**[Hateoas]** SpringSource, Spring Hateoas auf github,
`https://github.com/SpringSource/spring-hateoas`
[**RestBucks]** Spring Restbucks, Oliver Gierke,
`https://github.com/olivergierke/spring-restbucks`
**[SpringMVC]** SpringSource, Spring MVC Web,
`http://static.springsource.org/spring/docs/3.2.x/spring-framework-reference/html/mvc.html`

**Phillip Ghadir** baut am liebsten tragfähige, langlebige Softwaresysteme. Er ist Mitglied der Geschäftsleitung bei innoQ und hat sich früh auf Architekturen für verteilte, unternehmenskritische Systeme spezialisiert. Darüber hinaus ist er Mitbegründer und aktives Mitglied des iSAQB, des International Software Architecture Qualification Board.
E-Mail: phillip.ghadir@innoq.com