FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Evaluierung des HATEOAS-Supports von Java Frameworks

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Alexander Rashed
Matrikelnummer 1325897

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Dr.-Ing. Stefan Schulte
Mitwirkung: Christoph Mayr-Dorn Ph.D.

Wien, 30. Juni 2016

Alexander Rashed	Stefan Schulte

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Evaluation of HATEOAS support by Java frameworks

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Alexander Rashed
Registration Number 1325897

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Dr.-Ing. Stefan Schulte
Assistance: Christoph Mayr-Dorn Ph.D.

Vienna, 30th June, 2016

Alexander Rashed Stefan Schulte

Erklärung zur Verfassung der Arbeit

Alexander Rashed
Lerchenfelderstraße 74 1/2, AT-1080 Vienna

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 30. Juni 2016

Alexander Rashed

Danksagung

Danke an meinen Bruder Philipp, ohne den ich nicht der wäre, der ich bin.

Acknowledgements

Thanks to my brother Philipp, without whom I wouldn’t be who I am.

Kurzfassung

Representational State Transfer (REST) ist ein Architektur-Stil, welcher als Post-hoc-Konzeptionierung der Architektur des World Wide Web beschrieben wurde. Dieser Architekturstil, welcher durch eine Menge an Bedingungen definiert ist, verspricht eine hoch skalierbare und veränderbare Software-Architektur. Allerdings unterscheidet er sich fundamental vom klassichen SOAP- oder RPC-basierten Ansatz. Vor allem bezüglich der Bedingung, dass Hypermedia den Applikations-Status vorantreiben muss (Hypermedia As The Engine Of Application State - HATEOAS), leiden Entwickler an fehlendem sachlichen Verständnis und an fehlender Framework-Unterstützung.

Ziel dieser Arbeit ist es, die beliebtesten Java REST Frameworks bezüglich deren HATEOAS-Unterstützung zu evaluieren. Zugunsten einer analytischen Vorgehensweise wird ein Reifemodell entwickelt. Eine Test-Anwendung und eine Menge an Tests werden spezifiziert um die Klassifikation der Frameworks in die Stufen des Reifemodells zu unterstützen.

Die Test-Anwendung und die Tests werden mit jedem einzelnen der ausgewählten Frameworks implementiert. Diese Implementierungen erlauben die Klassifikation der Frameworks in die Stufen des Reifemodells. Anschließend werden die Frameworks verglichen und die Ergebnisse der Evaluierung präsentiert.

Abstract

Representational State Transfer (REST) is an architectural style, described as a post-hoc conceptualization of the architecture of the World Wide Web. Following this style, which is defined as a set of constraints, promises to lead to a highly scalable and modifiable distributed software architecture. However, this architectural style differs completely from the traditional SOAP- or RPC-based approach. Especially when it comes to the constraint of hypermedia as the engine of application state (HATEOAS), developers suffer from a lack of understanding for the subject matter and framework-support.

The aim of this work is to evaluate the HATEOAS support of the most popular Java REST frameworks. In order to analytically evaluate the different frameworks, this thesis develops a new maturity model. A test application and a set of tests are specified to support the classification of the frameworks into the levels of the maturity model.

The test application and the tests are implemented using each of the selected frameworks. These implementations allow the classification of the frameworks into the levels of the maturity model. Afterwards the frameworks are compared and the evaluation results presented.

Contents

List of Figures

List of Tables

List of Listings

CHAPTER 1

Introduction

I am getting frustrated by the number of people calling any HTTP-based interface a REST API. [...] Is there some broken manual somewhere that needs to be fixed?
Roy T. Fielding

1.1 Motivation

Representational State Transfer (REST) is an architectural style initially described by Roy Fielding in his PhD thesis [1]. It is a framework of constraints and is meant to be serving as a post-hoc conceptualisation of the architecture of the World Wide Web. In times where applications are facing more and more clients and network environments become pervasive, the successful growth of the Web seems very promising to application developers. Therefore, the interest in the architectural style and the number of ”RESTful” services are high and still increasing [2].

While these promises are very attractive, they come with a price. A developer has to fully understand the described architectural style in order to correctly apply it. Especially when it comes to *Hypermedia As the Engine of Application State* (HATEOAS), one of the key constraints of REST, the design differs completely from the traditional RPC and WS* approach. That is to say that not only the protocol but the whole design and usage of the newly created Application Programming Interface (API) changes on server- as well as on client-side.

In order to address the needs of REST service developers, countless frameworks for all major programming languages have been created. Especially for Java, a language known for its extensibility regarding open source frameworks, it might be challenging to keep an overview of the frameworks and finally select the one which seems to be best.

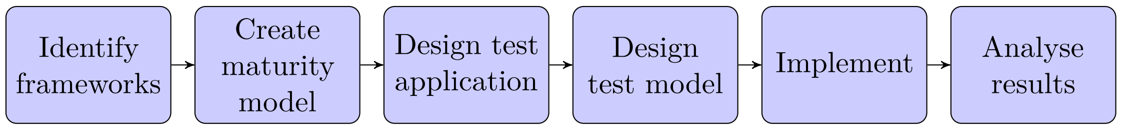

Figure 1.1: Methodological approach

1.2 Problem statement

Even though there are currently plenty of Java frameworks for REST service development available, many of them lack to support the developers in fulfilling the HATEOAS constraint. Without that constraint, however, many of the benefits, achieved by the loose coupling which comes with HATEOAS, are lost. As Roy Fielding himself stated in a popular blog post: ”REST APIs must be hypertext driven” [3].

1.3 Aim of the work

The aim of this work is to evaluate the most popular Java REST frameworks in order to find out how well they support developers in implementing HATEOAS compliant RESTful services. The evaluation is based on a particular maturity model. The frameworks are classified by implementing a test application and test scenarios matching that maturity model.

1.4 Methodological approach

The methodological approach of this work is shown in figure 1.1. It covers the following areas:

- At first, the frameworks that require evaluation have to be identified. The exercise aims at those Java frameworks that have the highest impact in the Java community. This means not only assessing concrete framework implementations but also the official standard for Java REST APIs.

- In order to classify how well the different frameworks assist in implementing REST services which fulfil the HATEOAS principle, we will create a maturity model. This maturity model consists of several levels. Each level represents a different goal which has to be achieved to receive all the benefits promised by the HATEOAS principle.

- Afterwards, an example application is designed. Even though the application should be as simple as possible, it has to be complex enough to create different scenarios covering all levels of the maturity model.

- On top of this application, a test model is created. The created tests will cover the different levels of the maturity model. Also some aspects of an API will be identified in order to analyse the simplicity of usage of the different frameworks.
- The test application will then be implemented in all of the selected frameworks. In order to analyse the adaptability of the different implementations, there will be two stages. At first, the initial server- and client-implementations are created. In the second stage, several aspects on the server-side are changed. This will have an impact on the client- as well as on the server-side.
- Finally, the results of the tests in the different stages and different aspects of the different APIs will be mapped to the maturity model. Based on that data we will identify the current status, capabilities and problems of the selected frameworks.

1.5 Structure of the work

In chapter 2 the REST architectural style and particularly HATEOAS will be explained in detail. At first the origins and basics of REST are described. Afterwards, the different constraints that have to be fulfilled by RESTful applications and especially HATEOAS are examined. The chapter also explains the basics and importance of domain application protocols as well as the currently established maturity model for REST applications. In the end of the chapter, the benefits and the problems developers have to face when it comes to the proper implementation of REST applications are illustrated.

Chapter 3 presents the actual evaluation. The different steps of the methodological approach, listed in section 1.4, are described in detail. Following the framework selection a maturity model is created. Based on this maturity model, a test application and tests matching the model are designed. In the end the tests and the application are implemented using the different frameworks.

In chapter 4 the results are presented and discussed. It also discusses open issues of the frameworks and evaluation.

Chapter 5 gives a summary of the evaluation and possible future areas of exploration are discussed.

CHAPTER 2

Representational State Transfer

2.1 Fundamentals

The success of the World Wide Web is incontrovertible and still ongoing. During the design process of HTTP 1.1 [4], Roy Fielding developed a core set of constraints, principles and properties in order to defend his design choices [5]. This initial model, which can be seen as a post-hoc conceptualization of the architecture of the World Wide Web, would later end in his PhD thesis, ”Architectural Styles and the Design of Network-based Software Architectures”, introducing and elaborating the *REpresentational State Transfer* (REST) architectural style for distributed hypermedia systems [1, chapter 5] [6].

In his dissertation, Fielding describes REST as a framework of constraints applied to the different roles and elements as well as their interactions within a distributed hypermedia system. It is an abstract definition without any details regarding the component implementation or even protocol syntax. This means that REST is actually not strictly related to HTTP, even though Fielding himself applies REST to HTTP within his thesis as well as other literature most commonly associated them.

The name ”Representational State Transfer” describes how a proper REST-application should behave: ”a network of web pages (a virtual state-machine), where the user progresses through the application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use” [1, chapter 6.1, page 109].

After the publication of Fielding’s dissertation, it took several years until *RESTful (Web-) services* - as those services which are implemented according to the REST architectural style are often called - gained mainstream popularity and became a serious competitor to the traditional SOAP-/RPC-based web service model. Even though it is not possible to generally say that one style is better than the other, it became evident over time that REST provides advantages in terms of several architectural properties like scalability and loose coupling (as explained in section 2.5).

2.2 REST constraints

Fielding describes the architecture of the Web (and thereby REST) by defining a set of constraints applied to the elements within the architecture. The constraints described in this section can be considered the core of the REST architectural style. Due to the common association with HTTP and to show an actual implementation, the constraint descriptions in this section also explain how they are applied to HTTP and URI. However, this listing is not complete, as Fielding defines more constraints - e.g. about separation of concerns, caching or code-on-demand - in his work.

2.2.1 Resources and resource identification

In REST, the key abstraction of information is a *resource*. The formal definition of a resource is as follows:

Definition ”A *resource* R is a temporally varying membership function $MR(t)$, which for time t maps to a set of entities, or values, which are equivalent.” [1, section 5.2.1.1]

Any information that is used within an application is considered a resource. Any resource can be the target of a hypertext reference (hyperlink, or just link). This concept of a resource is explicitly not limited to static data. Information with a high degree of variance in their value over time is considered a resource as well. Only the semantics of a resource has to be static. This distinction is necessary to allow the independent identification and referencing of the information.

Example The ”employee of the month” of a coffee shop is a mapping ($MR(t)$) whose value changes potentially every month. A mapping to ”employee #1337” on the other hand is static. Those two mappings, even if they map to the same value at the same time, are different resources, because the semantics of those two mappings are different.

Every single resource has to be uniquely and stable identifiable. These identifiers should also be global in order to allow for dereferencing them regardless of their context.

Application In the typical Web architecture *Uniform Resource Identifiers* (URI) [7] are used to identify the resources.

2.2.2 Uniform interface

One of the most central features of REST in comparison to other network-based architectural styles is the definition of a uniform interface between its components.

Definition All interactions with resources are using a *uniform interface*. This uniform interface supports those interactions by providing a general and functionally sufficient set of methods.

This constraint differs completely from the traditional SOAP-/RPC-based web service model, in which the set of methods that can be called is defined for every single web service. RESTful services expose a set of resources and clients can only interact with all the resources by using the uniform interface (or a subset of it).

Example The uniform interface constraint is a formulation of the software engineering principle of generality. A typical example is the creation of an interface for similar classes in object oriented programming. Instead of having a class ”Car” with the method ”drive” and a class ”Truck” with the method ”transport”, one would create an interface ”Vehicle” for both classes, defining the method ”drive”.

By applying that principle, the architecture is simplified, the visibility of interactions is improved and the service implementation is being decoupled from its interface. The downside, however, is a decreased efficiency, because information has to be transferred in a more standardized form rather than in a highly efficient, application specific form.

Application In order to describe the application of the uniform interface constraint in HTTP, it is necessary to define two important properties:

Definition ”Methods can also have the property of ”*idempotence*” in that (aside from error or expiration issues) the side-effects of N $>$ 0 identical requests is the same as for a single request.” [4, section 9.1.2] Therefore, it does not matter whether a client invokes an idempotent method once or multiple times, the outcome has to be the same.

Definition *Safety* is a property of methods that do not generate any side-effects the client can be held responsible for. The invocation of a safe method is rather about information retrieval (without modifying anything) than taking action in any form. Therefore, a client is allowed to call a safe method without considering any side effects at all.

In HTTP, the unified interface for all resources is defined by the different methods of HTTP and their returned status codes. The most commonly used HTTP methods are:

- *GET*
  The GET method is used when a client wants to receive a resource’s representation (rather than modifying it in any way). GET is *safe* and *idempotent.*
- *PUT*
  The PUT method is used to update a resource’s representation by sending the whole representation in the request entity. PUT is *idempotent.*
- *PATCH*
  The PATCH method is used to request a set of changes described in the request entity. This method has been specified in 2010, 11 years after HTTP/1.1 [8]. PATCH is *neither safe nor idempotent.*

- *POST*
  The POST method is the least strict HTTP method. It is usually used to invoke a specific resource's state transition. Due to the late specification of the PATCH method, POST is still widely used instead of PATCH for simple resource creations and modifications. POST also is *neither safe nor idempotent.*

- *DELETE*
  The DELETE method is used to delete a resource. DELETE is *idempotent.*

All those methods have to be used completely compliant to their specification. A common mistake is to ignore the properties of idempotence or safety of HTTP methods. This does not only tighten the coupling of the system but can also have unexpected side effects, as a lot of systems rely on that specification. For example, a search engine's crawler could modify or delete an application's data if its GET method does not fulfil the safety property [9].

2.2.3 Self-descriptive messages

Definition All messages have to be self-descriptive. Each message has to include all necessary information describing how to process the message.

This constraint implies that every message has to be ”labelled” so that no out-of-band information or agreement is necessary to ”understand” the received message. Self-descriptiveness does not convey any information about the semantics at all, but only specifies that no other information than the message itself is needed in order to process it.

Application In HTTP, the *Content-Type* header field defines the type of the message body. This is usually a registered Internet media type [10] (for example *text/html* or *application/xml*), but can also be an extended content type only known by a limited number of clients.

2.2.4 Stateless interactions

In order to improve reliability, visibility and scalability, REST requires interactions to be stateless.

Definition ”Communication must be stateless in nature [...] such that each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client.” [1, section 5.1.3]

This constraint is important when it comes to scalability, as it ensures that the scalability of servers only depends on the number of concurrent requests to a server (at a specific point in time). It allows efficient load balancing by just redirecting a request to a different server without keeping track of a history of any kind.

Application HTTP supports this constraint by explicitly not providing any specific features for server-side state. Nevertheless, this leaves the developer responsible not to build stateful applications on top of it. Unfortunately, maintaining a server-side session for every single client (identified by a specific session cookie in the HTTP header), is a common design mistake breaking that constraint. This causes lots of problems when it comes to scaling those applications.

2.2.5 Hypermedia As The Engine of Application State

The last key constraint, and the focus of the evaluation in this work, is that REST applications have to use *hypermedia as the engine of application state* (HATEOAS).

Definition "*HyperText* is a way to link and access information of various kinds as a web of nodes in which the user can browse at will." [11] The term *hypermedia* is used to indicate that hypertext can include other types of media like images or audio data.

Hypermedia therefore describes the concept of providing (hyper-)links which are used to jump between different documents or nodes in a system.

Definition An *application state* is a snapshot of an application's execution at a specific point in time.

As a deduction of those definitions, the HATEOAS constraint introduces the necessity of only using hyperlinks to invoke a state transition within an application. In other words, only the links provided by the server allow the client to traverse, discover and manipulate the resources. The server's response messages are not only the simple response data anymore but an essential part of the network architecture [12]. This allows the server as well as the client to become stateless, because all necessary information is contained in the exchanged messages [2]. This results in a very loose coupling between the server and the client [13]. Figure 2.1 illustrates the general idea of HATEOAS.

Application In HTTP, links can be stored in the *Link* header field [15] or the *Location* header field (which is specified to be used in combination with certain HTTP status codes). But most of the hyperlinks are contained in the message body and their format is depending on the used content type (as described in section 2.2.3). Most of the links have an associated relation type ("rel") defining the semantics of the link. As with the media types of resources, there can be registered relations and extended relations. The registered relations are maintained by the *Internet Assigned Numbers Authority* (IANA) [16].

2.3 Domain Application Protocols

To ease the understanding of the HATEOAS constraint it is helpful to introduce the concept of *domain application protocols*.

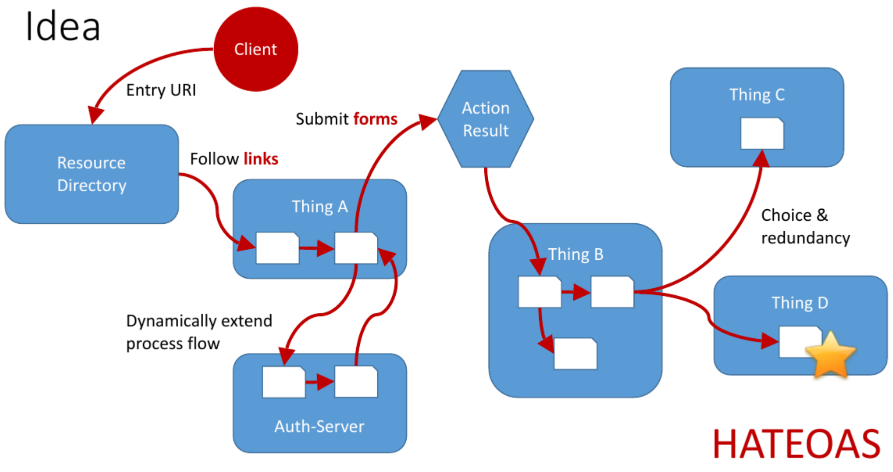

Figure 2.1: Idea of HATEOAS [14]

Definition ”A *domain application protocol* is the set of rules and conventions that guides and constrains the interactions between participants in a distributed application.” [17]

Every service inherently enforces a specific protocol - its domain application protocol. This protocol defines the way how to interact with the service.

The HATEOAS constraint enforces REST services to implement a domain application protocol by adding hypermedia links to its resource’s representations. Those embedded links are the only way a consumer can interact with the different resources and therefore modify the application state.

The client on the other hand is implementing the DAP by invoking application state transitions through the interaction with those embedded hyperlinks. This resembles the execution of a business workflow or, more generally, a finite state machine.

For REST services, all the hyperlinks (representing the state transitions in the domain application protocol) have to be contained in the messages from the server to the client. This allows a very loose coupling between client and server, as the client does not have to know the whole domain application protocol in advance. Instead of understanding a specific URI structure of the resources and how they can interact with each other at a certain point, a consumer only has to understand the semantics of a link embedded in the server’s response messages. This allows a service to evolve and change (with certain limitations) without breaking existing consumers.

2.4 Richardson Maturity Model

The described key constraints of the REST architectural style can be used as criteria to evaluate whether an application can be considered RESTful or not. The most popular model for analysing services is the ”Richardson Maturity Model” (RMM). It is named after Leonard Richardson who presented the model at a conference in 2008 [18].

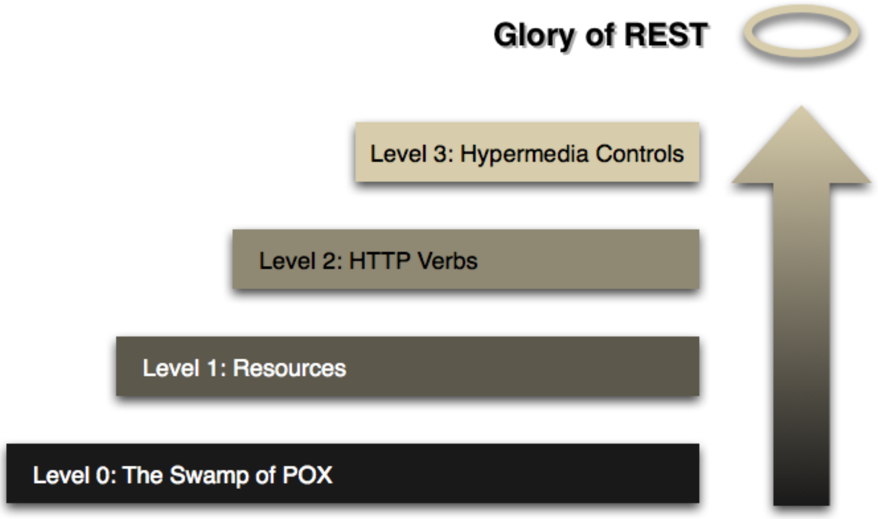

Figure 2.2: Richardson Maturity Model [19]

As shown in figure 2.2, the model consists of four different levels. Level 0 describes an application completely disobeying the REST principles while applications on level 3 can be considered RESTful:

Level 0: RESTless or ”The swamp of plain old XML” The lowest level of the model describes an application which is using HTTP as the transport protocol for remote communication, but it does not use any mechanisms of the Web so far. It basically describes typical XML-RPC (Remote Procedure Call) or SOAP (Simple Object Access Protocol) services, which only use the HTTP protocol as a tunnel for their own protocol. Those services usually only use one HTTP method and only one URI.

Level 1: Resources When it comes to REST the most obvious change is the usage of many URIs (as there has to be one for every resource). On level one the services are exposing multiple resources with unique URIs but they are still only using one HTTP method, clearly violating the uniform interface constraint. Nevertheless there are plenty of widely used services who operate on this level [3][18].

Level 2: HTTP verbs Applications operating on level two are using these HTTP methods and status codes according to their specification (instead of just using one method to tunnel their own protocol). Every single resource, identified by its unique URI, is now accessed and manipulated using an uniform interface.

Level 3: Hypermedia controls The highest level requires hypermedia as the engine of application state. Most of the services settle with level two, because the concept of hypermedia is hard to understand and developers don't see the value of it for their specific web service domain. According to an analysis, less than 20% of the most common Web APIs actually link their related resources [20]. Nevertheless, there are successful examples providing services compliant to the HATEOAS principle.

2.5 Benefits

In his dissertation, Roy Fielding identifies a set of architectural key properties which are clearly influenced by the REST architectural style. The most obvious benefit that comes with a proper implementation of the style is the amplification the following architectural key properties:

Visibility The constraint of stateless interactions increases the visibility by allowing a monitoring system to determine the complete nature of a single request without analysing any other messages.

The uniform interface constraint can be seen as applying the principle of generality to the component interface. This simplifies the overall system architecture and increases the visibility of interactions.

Scalability Stateless interactions in REST applications allow a server to simply free all resources of a request as soon as it has been processed. A server component does not need to maintain and manage a client state of any kind. This drastically increases the scalability of the application and decreases the complexity of the server component. The stateless interactions also allow for a very efficient load balancing [21].

Modifiability The property of modifiability depends heavily on the coupling of the server and the client. As the uniform interface and HATEOAS significantly loosens this coupling [21] [22], REST is maybe the most successful architectural style in supporting runtime application evolution [23] [24]. This property is of increasing interest, especially when it comes to pervasive network environments [25] or the ”Internet of Things” [26].

2.6 Problems

Even though Fielding clearly specifies every single constraint of the REST architectural style and it is defined in the acronym REST itself, there is still a lot of debate about whether specific APIs are RESTful or not. Lots of them claim to be RESTful even though they clearly violate the HATEOAS constraint [3].

This may be caused by a lack of understanding, due to Fielding's PhD thesis being ”far more often referred to than it is read” [27, foreword by Martin Fowler]. In order to address this problem, several projects try to explain the concepts as easy as possible[28].

There is however also a lack of tools and frameworks supporting the developers when it comes to this challenging task. In the past, there were several attempts to design REST frameworks that might help developers fulfil the HATEOAS constraint, unfortunately most of them weren't really successful [29] [30]. This apparent lack of good frameworks leads to the aim of this work.

CHAPTER 3

Methodology

3.1 Framework selection

The aim of this work is to evaluate the most important and popular Java REST frameworks. The Java ecosystem is known for its countless open-source frameworks for every imaginable purpose. This is also the case when it comes to frameworks that aim to support developers in creating RESTful services.

3.1.1 Java Platform

In order to select the appropriate frameworks, we have to select the ones that are most likely picked by a developer who starts a new project. In order to find those, it is necessary to understand the Java platform and its processes and specifications.

Java Community Process (JCP) The JCP is an open mechanism for creating new standard technical specifications for the Java platform. It manages the specification processes for upcoming language features of the Java programming language itself as well as the development of standard libraries. The actual specifications are described in *Java Specification Requests* (JSR). Once the JCP publishes a final release of a JSR it becomes official. Every final JSR has to provide a *reference implementation*, which is an open-source implementation of the new specification.

Java Platform, Enterprise Edition (Java EE or JEE) JEE is a very popular enterprise computing platform. It is developed under the JCP and is basically a collection of different specifications creating a bundled API. This API then allows developers to implement large-scale, multi-tiered webservices. On the other hand, the platform is used as a specification for application servers. A JEE application server has to be compliant to all specifications which are contained in the JEE version it is certified for. This allows applications, which are implemented only against specifications which are contained in a specific JEE version, to be compatible with every application server, which is certified for that JEE version. The current version is Java EE 7.

Java API for RESTful Web Services (JaxRS) With JaxRS the JCP tried to create a standard for implementing services compliant to the REST architectural style. The current version of the standard is 2.0 and is specified in the JSR 339 [31] [1]. The JSR 339 - JaxRS 2.0 - is contained in Java EE 7. Every Java EE 7 certified application server has to provide an implementation of the specification out-of-the-box.

Spring Framework The Spring Framework is a very popular open-source application framework for the Java platform. The core of the Spring framework can be used by any Java application, but its popular features are the extensions for creating web applications on top of the JEE platform. It aims to ease the JEE application development and tries to promote good programming practices, like the usage of specific programming patterns. The downside in turn is, that an application has to commit to the Spring framework instead of the JEE platform, as the two platforms are not compatible.

3.1.2 Selected Frameworks

Considering the Java platform and its specifications, the following frameworks have been selected for the evaluation:

JaxRS 2.0 (JSR 339) [32] If a developer only uses features provided by the official Java API for RESTful Web Services, the application can be used with every framework which officially implements this JSR (like Jersey, RestEasy and RESTlet, just to name a few of them). Most applications can be deployed on every JEE 7 certified application server. Therefore lots of developers commit to use only the official API in order to ensure this portability. For that reason JaxRS 2.0 has been selected to be evaluated.

Jersey 2.22.2 [33] The Jersey framework is the reference implementation for the first three JaxRS versions. For the upcoming version 2.1 of the specification, Jersey will be official open-source implementation as well. The reference implementation is usually the first framework compliant to a specification. It is also a standard component of the Glassfish application server. The development of the framework and the Glassfish application server is sponsored by the Oracle Corporation. These circumstances, as well as the stable Jersey specific feature set, convinces a lot of developers to use the framework for their projects, which makes it necessary for it to be evaluated in this work. The default API is a plain implementation of the JaxRS 2.0 specification, but with version 2.6, a Jersey specific experimental extension for server-side annotation based linking, called declarative linking, has been introduced.

[1] The previous versions of JaxRS, 1.0 and 1.1, are specified in the JSR 311. The upcoming version, JaxRS 2.1, is currently developed under JSR 370.

RestEasy 3.0.17 [34] Besides Jersey, RestEasy is probably the most successful REST framework for the Java platform. It is not only fully implementing JaxRS 2.0, but also provides a very rich set of features which are not part of the standard. The development of the open-source framework is managed by JBoss Inc., which is part of RedHat. It is hence the standard component in the JBoss Enterprise Application Server and the WildFly Application Server. This framework is known to be very innovative and was created by Bill Burke, who is a member of the expert group for the JSR 339 (JaxRS 2.0) and the upcoming JSR 370 (JaxRS 2.1). Therefore, this framework is also part of the evaluation.

Spring HATEOAS 0.19 [35] Spring HATEOAS is aiming to provide an API to ease the creation of REST resources which actually are compliant to the HATEOAS constraint when using the Spring application framework. Due to the major popularity of the Spring framework for enterprise application development, Spring HATEOAS is part of this work's evaluation.

3.2 Analysis method

In order to analytically evaluate the different frameworks, it is necessary to develop a clear scheme of how to assess them. For that purpose a new maturity model is developed in section 3.3. It defines the several levels a framework can reach in terms of adherence to the HATEOAS principle. The maturity model will define a set of characteristics for each level and the frameworks will then be analysed by assessing which characteristics of the different levels they fulfil.

The evaluation is based on a test application which is described in section 3.4. This test application will be implemented using each of the selected frameworks.

These test application implementations are then used to execute the tests for the characteristics of each level. These tests are specified in section 3.5. For each test level, the initial test application implementation will be used in order to avoid side effects of the necessary changes for one test on any of the other tests. Therefore, a new branch in the source code management system, created from the same base, is created for every level.

As a final step the results for every framework are displayed and compared with each other.

3.3 REST Framework Maturity Model

The Richardson Maturity Model (as described in section 2.4) is used to analyse how RESTful an application actually is. The "REST Framework Maturity Model" (RFMM) introduced in this section, on the other hand, is built for the purpose of the evaluation of frameworks. Nevertheless, the levels of evaluations and their indicators could also be used for applications in order to determine how well the HATEOAS constraint is fulfilled.

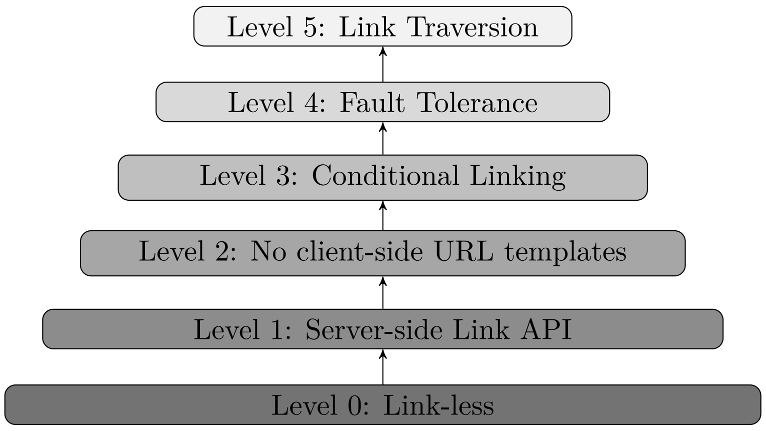
Level 5: Link Traversion
Level 4: Fault Tolerance
Level 3: Conditional Linking
Level 2: No client-side URL templates
Level 1: Server-side Link API
Level 0: Link-less

Figure 3.1: Framework Maturity Model

A problem here is that, at least until fulfilling level two, an application is definitely not compliant to the HATEOAS constraint at all (as it cannot be done partly).

The maturity model basically starts at the last level of the Richardson Maturity Model, the correct usage of resources and the HTTP protocol is a premise. It consists of 6 different levels. Each of these levels can be seen as a ”checkpoint” a framework can reach in assisting a developer to create HATEOAS compliant applications. The purpose of those levels is to simulate the evolution of a REST application when it comes to fulfilling the HATEOAS constraint. Therefore, each level depends on its predecessor (like in the RMM). In order to allow for a more fine-granular distinction of the frameworks, the model defines a set of characteristics for every single level. The structure of the model can be seen in figure 3.1. Due to the fact that the proper client-side implementation is as important as the server-side when it comes to the HATEOAS compliance, the levels consider both sides. Therefore, it can either be used to analyse a framework containing both, a server- and a client-side API, or to analyse a specific technology stack containing one server- and a different client-side framework.

Level 0: Link-less The lowest level describes a framework which does not provide any linking-API. There is nothing provided that allows adding links to the message at all (neither to the metadata nor to the body of the message). If a developer decides to use this framework, all linking mechanisms have to be built from scratch. These frameworks therefore are not supporting the HATEOAS principle at all.

Characteristics

- *No linking* - The only characteristic of this level is the complete absence of any linking mechanisms provided by a framework.

Level 1: Server-side Link API The first step when it comes to HATEOAS is to actually include hyperlinks to the server’s response message. Therefore the level one of the maturity model applies to frameworks which server-side API do actually provide mechanisms for the creation and adding of links to the response. The characteristics of this level are meant to describe a reliable, readable API which is aware of common standards.

Characteristics

- *Links* - The response of the server actually contains links, like the "self" link referencing itself.

- *Type safety* - Is the provided API typesafe? This supports server-side refactoring and evolution.

- *Annotation-based linking* - Is the provided API based on Java annotations? This increases readability of the code. Only the business logic invocations have to be contained in the service method body.

- *IANA link relation usage* - Does the framework support the usage of the registered IANA link relations as specified in the Web Linking specification [15] [16]?

Level 2: No client-side URI templates One of the most important rules when it comes to HATEOAS is that a client may not use any static URIs besides the entry-point URI. Every other URI used during the execution must be provided by the server. Frameworks on level two of the maturity model actually encourage the developer to only follow ad-hoc links provided by the server's response messages instead of relying on prior knowledge of a URI structure.

Characteristics

- *Link usage* - Is it easily possible to use the links, contained in the metadata and the body of the message, for the next request to the server?

- *Misguidance* - Is the framework providing mechanisms which use static URIs (even though there may also be a dynamic mechanism)? This would probably entice developers not to build HATEOAS compliant clients even though the server's messages would provide all the necessary links.

Level 3: Conditional Linking Once the server is providing all the links necessary for the client to invoke the state transitions and once the client actually uses them, an application can be considered compliant to the HATEOAS constraint. But the server-side code can still be improved. As the server has to model the domain application protocol (as described in section 2.3), deciding which link should be added to the response - if at all - can be challenging. Such a condition usually leads to a decrease in the readability of the server-side code. Level three therefore defines that a framework has to provide an API which allows for defining conditions for specific links to be added to the response.

Characteristics

- *Conditions* - Does the framework provide any mechanisms to define conditions for links to be added?

Level 4: Fault Tolerance One of the main benefits that come with HATEOAS is modifiability due to the loose coupling. To allow the server to evolve, it is necessary that the client allows the server's responses to change as much as possible. Frameworks on level 4 of the maturity model help the developer in implementing fault tolerant clients.

Characteristics

- *Tolerant to added entity properties* - Does the unmarshalling allow for unexpected entity properties?

- *Tolerant to removed entity properties* - Does the unmarshalling allow for the absence of expected entity properties?

- *Default tolerance* - If the framework can be tolerant to changes in the response entities, is this the default behaviour?

- *Status code handling* - How well is the framework handling HTTP status codes other than 200 ("successful"), e.g. 307 ("temporary redirect")?

Level 5: Link Traversion The last level of the maturity model covers the client side. Frameworks compliant to this level are actively supporting the traversion of the provided links. These frameworks provide an API which allows the developer to easily implement clients executing the state transitions.

Characteristics

- *Traversion* - Does the framework provide a client-side API to easily traverse the links provided by the server?

3.4 Test application

In addition to the maturity model, a test application is necessary to practically analyse the selected frameworks. This application is meant to cover all typical scenarios. Therefore, a simple online shop scenario has been selected. The online shop provides a set of articles which can be added to baskets and these baskets can finally be paid. When implementing the different tests, the test application will be extended and modified to analyse the modifiability of the solution. Those modifications are specified for every test, which is described in section 3.5.

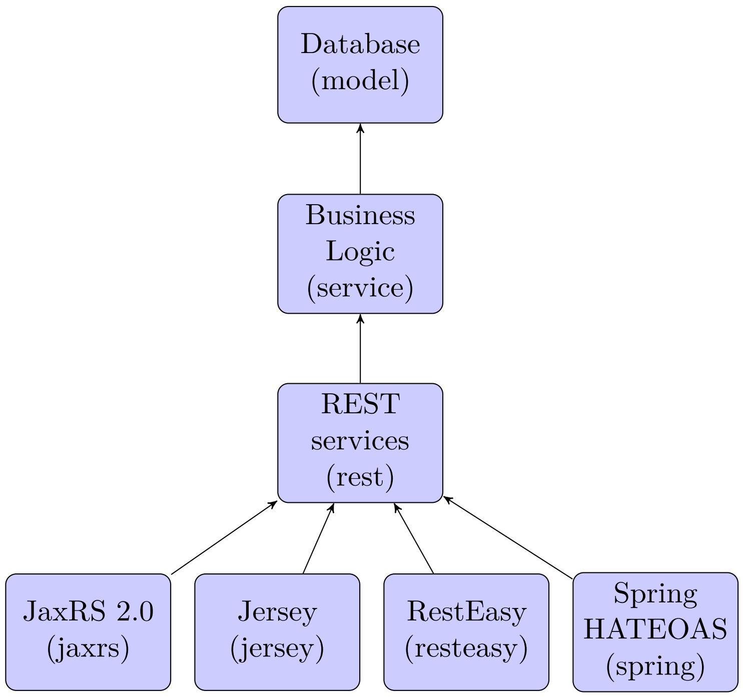

Figure 3.2: Test application architecture

Architecture The test application will be implemented for every single selected framework. In order to keep the REST service implementations as simple as possible, the architecture of the implementation is based on the classical three-tier architecture. This leads to a project structure and dependencies as shown in figure 3.2.

Resources The application consists of different resources. Every resource has its properties (fields), its own unique identifier (the URI path), supports a subset of the HTTP methods and provides a specific set of links depending on the current application state. All link relations are implemented according to the *Web Linking* specification [15]. Therefore, all officially registered link relation types are used according to their documentation in the registry [16]. All extension relation types are URIs (as defined in the *Web Linking* specification) with the custom scheme "tos" ("test online shop"). The only supported media type of all the resources is either *application/json* or *application/xml*.

Root The root resource is the initial resource a client starts with.

- *Properties:* –
- *Path:* /
- *Supported methods:* GET
- *Links:* tos:articles, tos:baskets, tos:bills

Articles The articles resource is the resource containing all article resources.

- *Properties:* list of articles
- *Path:* /articles
- *Supported methods:* GET, POST (create a new article)
- *Links:* self, next (if available), prev (if available), self (for every article in the list)

Article An article represents a single element which can be bought in the shop.
- *Properties:* name, description, price
- *Path:* /articles/{id}
- *Supported methods:* GET, DELETE, POST (add article to basket)
- *Links:* self, tos:addToBasket (one for every unpaid basket, adds article to a basket)

Baskets The baskets resource is the resource containing all basket resources.
- *Properties:* list of baskets
- *Path:* /baskets
- *Supported methods:* GET, POST (create a new basket)
- *Links:* self, next (if available), prev (if available), self (for every basket in the list)

Basket A basket represents a set of articles which can be bought/paid by a client.
- *Properties:* list of articles with their amount
- *Path:* /baskets/{id}
- *Supported methods:* GET, PUT, DELETE, POST (pay the basket)
- *Links:* self, payment (if not paid yet), tos:bill (if already paid)

Bills The bills resource is the resource containing all bill resources.
- *Properties:* collection of bills
- *Path:* /bills
- *Supported methods:* GET
- *Links:* self, next (if available), prev (if available), self (for every bill in the list)

Bill A bill resource represents a bill for a paid basket.
- *Properties:* –
- *Path:* /bills/{id}
- *Supported methods:* GET, DELETE
- *Links:* self, tos:basket

Typical workflow The typical workflow of a client when using the application can be seen in figure 3.3. It expects that the master data has already been created, which means that the system already contains some articles and at least one basket. The client starts with the usual root URI path "/". Afterwards, it follows the relation "tos:articles" to receive the first page of the collection of articles. The client selects an article in the list and follows its "self" link to get the detailed data of the article. In order to add the article to the basket, the client performs a POST to the link marked with the relation "tos:addToBasket". The result message will contain a link to the modified basket in its

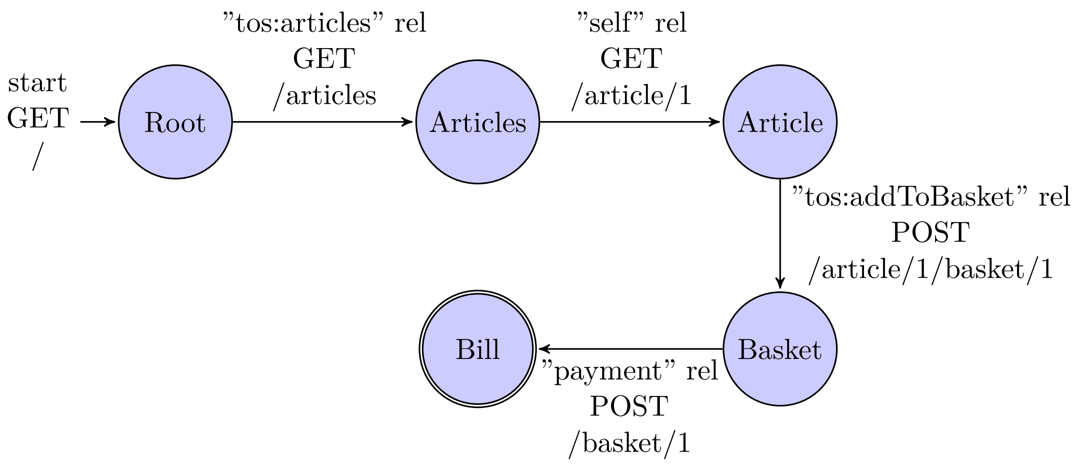

Figure 3.3: Typical test application workflow

"Location" header field. The client follows that link and receives the content of the basket. The client then performs the payment of the basket by performing a POST to the link with the relation "payment". The response finally links to the newly created bill.

3.5 Tests

The framework classification with regards to the different levels of the maturity model is done by testing their compliance with the different characteristics of each level. If possible, the test is implemented as a JUnit integration test. This is not possible for every characteristic, as some of them can only be tested manually by identifying specific elements in the API.

Level 0: Link-less For the initial level, there are no specific tests. The complete absence of any link API is trivially determined.

Level 1: Server-side Link API Level one describes the server-side linking API. The most important part is, that it is actually available.

Tests

- *Links* - *JUnit* - An integration test checks the response message of the server if it actually contains links.

- *Type safety* - *JUnit* - The type safety is checked by using an integration test implementing the default workflow described in section 3.4. On the server-side the class- and method-names are changed. If the API is actually type-safe the integration test continues to pass.

- *Annotation based linking - Manually* - Are there any features in the API which allows adding links by annotating classes, methods or fields?
- *IANA link relation usage - Manually* - Are there any features in the API which support using link relations according to the specifications?

Level 2: No client-side URI templates Level two defines that a link actually has to use the provided links instead of using static URI templates.

Tests

- *Link usage - JUnit* - An integration test uses the provided client API to implement the default workflow described in section 3.4. On the server-side, the paths of the different resources are changed. If the client can actually use the provided links, the integration test continues to pass.
- *Misguidance - Manually* - Are there any features in the API that allow the usage of client-side URI templates?

Level 3: Conditional Linking In order to support the developer in implementing a domain application protocol, level three inspects the mechanisms of the API to apply conditions to links.

Test

- *Conditions - Manually* - Are there any features in the API that allow adding of links only when a specific condition is fulfilled?

Level 4: Fault Tolerance Level four describes the fault tolerance of the framework.

Tests

- *Tolerant to added entity properties - JUnit* - An integration test uses the provided client API to implement the default workflow described in section 3.4. On the server-side a new property "name" is added to the basket resource representation. If the client API is tolerant to this change, the test continues to pass.
- *Tolerant to removed entity properties - JUnit* - An integration test uses the provided client API to implement the default workflow described in section 3.4. On the server-side the property "description" is removed from the article resource representation. If the client API is tolerant to this change, the test continues to pass.
- *Status Code Handling - JUnit* - An integration test uses the provided client API to implement the default workflow described in section 3.4. On the server-side, the server's response for the bill's collection resource URI path returns the HTTP status code 307 ("temporary redirect"), redirecting the client to another URI. If the client API automatically follows the given link, the integration test will continue to pass.

- *Default tolerance - JUnit* - If the framework is tolerant to those changes, is it the default behaviour? The test cases for the tolerance to adding and removing properties will be repeated, but with a newly created, non-configured marshaller.

Level 5: Link Traversion Passing the last level requires that the framework has to provide an API to traverse through the resources.

Tests

- *Traversion - Manually* - Are there any mechanisms provided by the framework to easily traverse through the links on the client-side?

3.6 Implementation

In order to allow a reliable classification of the selected frameworks, the tests of the different levels are implemented based on the test application described in section 3.4. Due to the outstanding branching model, the source code management system Git is used to support the development process. The whole Git repository is available online [36]. In order to preserve the code in a sustainable and identifiable way, every single branch has been archived [37] [38] [39] [40] [41] [42] [43].

At first, the test application is implemented in the *master* branch using the different frameworks. In addition to the application, a set of integration tests ensures the correct functionality of the test application. This branch acts as the base for the *test_prep* branch which prepares the code base for the different level-specific tests. Therefore the build process configuration is slightly adapted and the initial integration tests are deleted. On top of the *test_prep* branch, the tests for the different levels are finally implemented in separate independent branches (*rfmm_lvl1-5*).

The repository structure with the different commit hashes and branches can be seen in figure 3.4.

3.7 Evaluation Results

After executing the tests for every framework by implementing the JUnit tests and analysing the features they provide, the evaluation shows the following results [2]:

3.7.1 JaxRS 2.0

The Java API for RESTful Web Services can be considered as the base of all Java REST framework for the Java Enterprise Edition platform.

[2]Level 0 is not explicitly listed as it is only defined for frameworks which do not even fulfil level 1.

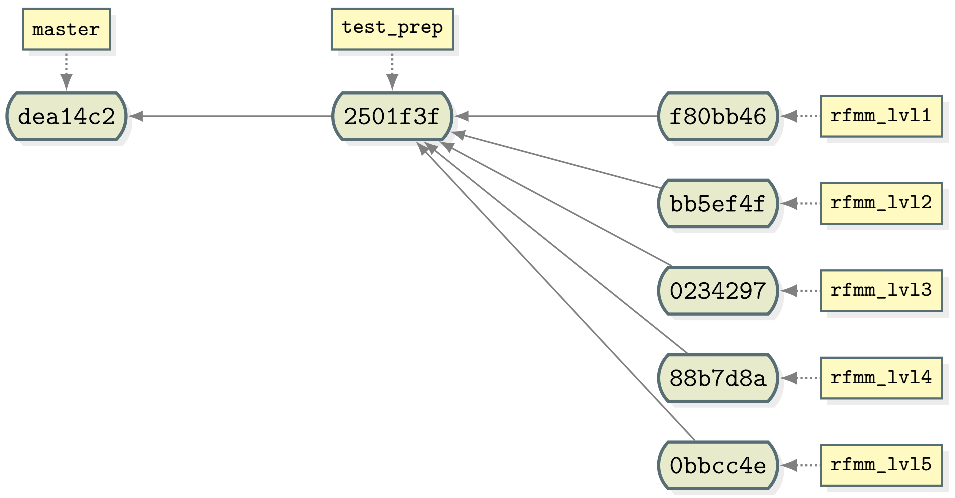

Figure 3.4: Git branch structure of the evaluation sourcecode

Level 1: Server-side Link API

With version 2.0 of the official Java API for RESTful Web Services the *Link* type as well as mechanisms for building and marshalling those links have been introduced. Unfortunately, the link building mechanism is not completely typesafe as linked methods are defined by String- or Method-objects (as seen in listing 3.1 - the creation of the link which allows adding an article to a basket). The specification API neither provides any IANA link relation types nor annotations which would allow adding links.

```
Link basketLink = Link.fromUriBuilder(UriBuilder.fromResource(ArticleResource.class)
   .path(ArticleResource.class, "addArticleToBasket"))
   .rel("tos:addToBasket")
   .build(article.getId(), basket.getId());
```

Listing 3.1: JaxRS 2.0 link creation

This shows that JaxRS 2.0 only fulfils the minimum requirements for level one of the maturity model.

Level 2: No client-side URI templates

JaxRS 2.0 finally introduced a client-side API and therefore frameworks which fulfil the specification have to provide those features as well. This part of the specification is a clear statement to HATEOAS as it completely dismisses the usage of any static URI templates. It enforces the usage of provided links and does not support reusing server-side interfaces as a static URI template definition.

The specification fulfils all criteria for level two of the maturity model.

Level 3: Conditional Linking

Unfortunately, JaxRS 2.0 does not provide any possibility to assign conditions to links in order to control when links are added to the response message. Therefore, possibly complex if/else or switch/case statements have to be used in the REST service methods to determine which links have to be included according to the current resource and application state.

JaxRS 2.0 therefore does not fulfil level 3 of the maturity model.

Level 4: Fault Tolerance

The marshalling and unmarshalling of the transferred messages is not part of the JaxRS standard. It provides, however, a plug-in-facility for frameworks supporting the parsing of the different media types. The specification also does not mention any default behaviour for specific HTTP status codes. This requires testing different implementations. Two frameworks, which are implementing the standard, are part of this evaluation. The explanations concerning level 4 of the Jersey framework can be found in section 3.7.2, the results of the RestEasy framework in section 3.7.3.

Level 5: Traversion

JaxRS 2.0 provides no features designed to ease the client-side link traversion.

It is not compliant to the last level of the maturity model.

Result

From a platform perspective JaxRS 2.0 provides a major step forward as it is finally providing a linking- and client-API. Unfortunately, it takes time for a JSR and the community to find a uniform solution. Therefore, the JSRs are usually very conservative when it comes to new advanced features, as this requires each implementing framework has to completely stick to the specification.

When it comes to portability, JaxRS 2.0 has great benefits, but a developer has to deal with a lot of boilerplate code and will miss a lot of features from other frameworks.

JaxRS 2.0 is conform to level two of the REST framework maturity model.

3.7.2 Jersey 2.22.2

As mentioned in section 3.1.2, the Jersey specific features for server-side linking are still experimental and have been introduced with version 2.6 of the framework. This is the result of a long design process, as Jersey developers have been trying to design a proper linking API for years now [30].

Level 1: Server-side Link API

Jersey's declarative linking approach is a straight-forward API which allows annotating JaxB beans and their fields to declare which links should be injected and when. It allows adding links to the HTTP header as well as to the body itself. An example of those annotations, which illustrates the declaration of the links of a basket resource, can be seen in listing 3.2.

```
@InjectLinks({
    @InjectLink(resource = BasketResource.class, method = "getBasket", style =
        Style.ABSOLUTE, bindings = @Binding(name = "id", value = "${instance.id}"),
        rel = "self"),
    @InjectLink(resource = BasketResource.class, method = "payBasket", style =
        Style.ABSOLUTE, bindings = @Binding(name = "id", value = "${instance.id}"),
        rel = "payment", title = "Pay the basket", condition = "${instance.bill ==
        null}"),
    @InjectLink(resource = BillResource.class, method = "getBill", style =
        Style.ABSOLUTE, bindings = @Binding(name = "id", value =
        "${instance.bill.id}"), rel = "tos:bill", title = "Bill", condition =
        "${instance.bill != null}") })
private List<Link> links;
```

Listing 3.2: Jersey declarative linking

Listing 3.2 also shows a flaw when it comes to type safety: Just like JaxRS 2.0, Jersey uses a String-based method definition in the annotation which is clearly not typesafe. Unfortunately, the Java language does not provide any access to the methods of a class in a static way, therefore, no annotation based linking approach which enforces a method definition can be typesafe. Jersey does not provide any IANA link relation definitions.

Therefore the framework fulfils the most important characteristics of level one of the maturity model.

Level 2: No client-side URI templates

Jersey's client API does not provide any additional features to the JaxRS 2.0 specification. As explained in section 3.7.1 though, the official specification only provides mechanisms to follow links without the definition of static URI templates.

Therefore, Jersey also fulfils all criteria for level two of the maturity model.

Level 3: Conditional Linking

As shown in listing 3.2, the declarative linking API provides the possibility to apply a condition to every link declaration. These constraints define whether a link should be added based on a boolean Java Expression Language 3.0 expression. This expression language is specified in JSR-341[44] and also part of Java EE 7.

Jersey fulfils the criteria for level three of the maturity model.

Level 4: Fault Tolerance

Just as JaxRS, Jersey can use different parser frameworks for the marshalling and unmarshalling of the messages. The most commonly used JSON parser in combination with Jersey is the Jackson framework. It allows configuring the tolerance to unknown properties. The default configuration however throws an exception if it faces an unknown property during the unmarshalling process. Jersey automatically follows redirects when the server answers with the particular HTTP status codes.

Jersey fulfils most of the criteria for level four, fault tolerance can at least be configured, even if it is not enabled by default.

Level 5: Traversion

Jersey does not add any additional features to the JaxRS 2.0 client API, accordingly it also does not provide any features designed to ease the client-side link traversion.

Therefore, is not compliant to the last level of the maturity model.

Result

Even though the declarative linking feature is still experimental, it is a very powerful API that clearly has its benefits when it comes to the readability of the REST service methods. Nearly no code for the creation and addition of links is contained in these methods anymore. The client API however does not provide any additional features in comparison to the JaxRS specification.

Jersey 2.22.2 is compliant to level four of the REST framework maturity model.

3.7.3 RestEasy 3.0.17

Just like Jersey, RestEasy provides the advanced linking features as an extension. This extension, called RestEasy Linking, is considered to be final and promises simple, standard conform linking.

Level 1: Server-side Link API

RestEasy's linking API clearly tries to promote a modern convention-over-configuration approach. Just by adding an annotation to a resource's CRUD-methods, the framework tries to correctly identify and add them as Atom links [45, section 4.2.7] to the resource representations. The annotations provide a high amount of configuration possibilities that allows changing the relation types, defining conditions or adding custom links (in addition to the CRUD links).

```
@AddLinks
@LinkResources({
  @LinkResource(Basket.class),
  @LinkResource(value = Bill.class, rel = "tos:basket", pathParameters =
      "${this.basketId}", constraint = "${this.basketId != null}") })
public Basket getBasket(@PathParam("id") Long id) { ... }
```

Listing 3.3: RestEasy linking

This interesting approach cleverly bypasses the type safety problem Jersey is facing, as not the resources are annotated, but the targeted methods are. This results in a completely typesafe annotation-based API. RestEasy does not define any IANA link relations.

RestEasy therefore fulfils the important characteristics of level one.

Level 2: No client-side URI templates

The most popular client API RestEasy is providing is the Proxy Framework. It allows re-using the server-side interface definitions with their annotations to provide a completely static URI template. This API is very easy to use, but it completely bypasses all possible efforts in providing links to support HATEOAS. As RestEasy implements JaxRS 2.0 though, it also provides another, standard conform, client API.

Therefore RestEasy fulfils the important characteristics of level two, even though it misguides developers to use static URI templates.

Level 3: Conditional Linking

The framework allows adding constraints to all linking annotations. Just as Jersey, RestEasy uses boolean Java Expression Language 3.0 expressions.

RestEasy fulfils the criteria for level three of the maturity model.

Level 4: Fault Tolerance

The RestEasy linking plug-in is only compatible to the Jettison parser. On the one hand, this parser is very tolerant by default, it simply ignores unknown or removed properties. On the other hand, RestEasy does not handle any redirection status codes provided by the server.

The framework fulfils the important characteristics of level four of the maturity model.

Level 5: Traversion

RestEasy does not provide any link traversion mechanisms on the client side.

It is not compliant to the last level of the maturity model.

Results

RestEasy's linking approach is clearly the most interesting one, as it follows the popular convention-over-configuration approach, is typesafe and annotation-based. The feature is very stable, but the current activity on the plug-in and some developer discussions create the impression that there is currently no real priority to drive the API even further [46]. However, RestEasy produces the best results with a minimum amount of code.

RestEasy 3.0.17 is conform to level four of the REST framework maturity model.

3.7.4 Spring HATEOAS 0.19

If a developer commits to the Spring application framework and REST, there is no way around Spring HATEOAS. As usually the case in the Spring community, there is only one specific project for a specific approach. Spring HATEOAS is, as the name implies, aiming to provide features to implement HATEOAS compliant REST services on top of the Spring Web framework.

Level 1: Server-side Link API

Spring HATEOAS provides a fluent, type-safe API to easily create links to specific Spring controllers and methods. It is the only framework that comes with native support for the *Hypertext Application Language* (HAL) [47]. HAL is currently a RFC-draft which aims to specify a media type for resources and their links. This addresses the downside of JSON or XML in comparison to HTML, as these media types do not specify any link format. The framework also handles IANA link relations. A developer can provide a default URI scheme, all non-registered links are then automatically converted to URIs using this defined scheme. As shown in listing 3.4, the link building is based on static builder methods and not on annotations.

```java
if (entity.getBill() != null) {
   basket.add(linkTo(methodOn(BillController.class)
      .get(entity.getBill().getId())).withRel("bill"));
} else {
   basket.add(linkTo(methodOn(BasketController.class)
      .payBasket(entity.getId())).withRel("payment"));
}
```

Listing 3.4: Spring HATEOAS conditional link

Spring HATEOAS fulfils the most important criteria of level one of the maturity model.

Level 2: No client-side URI templates

On the client-side, Spring HATEOAS provides very useful features when it comes to paging through collections and traversing to the links. It enhances the usage of the client framework provided by Spring Web which eases the dynamic link usage. There also is no other client-side API promoting static URI templates.

Spring HATEOAS therefore fulfils all criteria of level two of the maturity model.

Level 3: Conditional Linking

As illustrated in listing 3.4, the framework does not provide a mechanism to assign conditions to a link. The decision of adding a link to a resource's representation has to be made in the service method.

Spring HATEOAS does not fulfil the criteria of level three of the maturity model.

Level 4: Fault Tolerance

The Spring Web's RestTemplate automatically follows server-side redirection commands. Also the marshalling can be configured to ignore unknown or missing parameters, even though it is not fault tolerant by default.

Therefore Spring HATEOAS fulfils the important criteria of level four of the maturity model.

Level 5: Traversion

Spring HATEOAS is the only framework which provides a client-side mechanism to easily traverse the links through the resource representations. An example of this API is shown in listing 3.5.

```
ResponseEntity<Basket> actual = new Traverson(basketURI,
  MediaTypes.HAL_JSON).follow("tos:bill").follow("self").follow("tos:basket")
  .follow("self").toEntity(Basket.class);
Basket basket = actual.getBody();
```

Listing 3.5: Spring HATEOAS link traversion

Spring HATEOAS is the only framework evaluated that fulfils the last level of the maturity model.

Results

Spring HATEOAS is, like most Spring projects, doing a good job in providing a straightforward and safe implementation that satisfies the needs of application developers. It integrates seamlessly into the Spring Web framework and provides a rich feature set with focus on practical usage and standard conformity. The downside is, that the link creation consumes a relatively big amount of code in the service implementations, as it cannot be transferred to annotations.

Spring HATEOAS 0.19 is conform to level two of the REST framework maturity model.

3.8 Comparison

When it comes to the results of the evaluation, it turns out that each selected framework has its up- and downsides. A comparison of the results for the individual tests of every level of the REST framework maturity model for each framework is described in table 3.1.

Each evaluated framework reaches at least level two of the maturity model. This means that they all provide the minimal feature-set needed in order to allow for the implementation of an application compliant to the REST architectural style.

JaxRS 2.0 seems to have the least amount of features. On the one hand this is clearly explainable by the typically conservative decisions during the specification process as every decision has to be accepted by every implementing framework. On the other hand, the specification ensures portability within the Java EE platform.

The Jersey framework comes with a well thought-out annotation-based extension, which clearly increases the readability of the REST service methods due to the outsourcing of all link related code to annotations in the bean classes. The downside, however, is that the API is still experimental and lacks type safety.

RFMM Level & Test	JaxRS	Jersey	RestEasy	Spring HATEOAS
0: Link-less	✓	✓	✓	✓
1: Links	✓	✓	✓	✓
1: Type safety	✗	✗	✓	✓
1: Annotations	✗	✓	✓	✗
1: IANA link relations	✗	✗	✗	✗
2: Link usage	✓	✓	✓	✓
2: Misguidance	✓	✓	✗	✓
3: Conditions	✗	✓	✓	✗
4: Added properties	✻	✓	✓	✓
4: Removed properties	✻	✓	✓	✓
4: Status handling	✻	✓	✗	✓
4: Default handling	✻	✗	✓	✗
5: Traversion	✗	✗	✗	✓

Table 3.1: Result comparison
✓ Test passed
✗ Test failed
✻ Test skipped

RestEasy clearly provides the most sophisticated and innovative API with its linking plug-in. It implements a powerful, annotation-based, typesafe convention-over-configuration approach. The currently low priority of the plug-in can be considered a disadvantage though.

Spring HATEOAS fits pretty well in the Spring eco-system. It delivers a seamlessly integrated typesafe server- and client-API focused on the practical usage and conformity to well established standards.

CHAPTER 4

Critical reflection

4.1 Comparison with related work

This work aims to evaluate the HATEOAS support of the most popular Java REST frameworks. It seems that there is not a lot of activity on this issue on a practical level. This may be caused by the frequent changes of the APIs and their supported features. Nevertheless, Fischer et al compared nine different Java REST frameworks in 2013 [48]. Their aim is to evaluate all different aspects when it comes to REST application development. This is why they only cover HATEOAS on the surface with a small set of criteria in their relatively comprehensive criteria catalogue. This work, however, focusses on that single aspect, which leads to a more accurate evaluation. In addition, their comparison was published in 2013, at a time where there wasn't even a reference implementation for JaxRS 1.1. As the JaxRS specification improvements and the ongoing developments of the frameworks changed a lot since then, their work may be considered outdated.

4.2 Discussion of open issues

Even though the implementation of the test application and the JUnit tests brought up some minor bugs in the different frameworks, their API and feature-set can be considered stable.

A bigger issue when it comes to generic clients, which has not been covered in detail in this work, is the lack of specifications when it comes to link-aware media types. As opposed to HTML, the most popular media types for REST service implementations like XML or JSON do not contain any specifications for links. This creates a major disadvantage when it comes to the implementation of generic clients, as the response messages of the different server implementations are using different and often proprietary linking formats. This means that the client has to have more knowledge about the different implementations and formats, which obviously leads to an unnecessary tight coupling.

And last but not least, the framework-developer- and specification-community seem to have a low priority on the linking frameworks due to their low popularity. This allows the conclusion that the further development of the JaxRS specification and the linking frameworks will slow down at least in the near future [46]. The question here is, whether the developers are not using the frameworks because they do not want to commit to the HATEOAS constraint, or whether the frameworks available just don't provide the required feature-set and simplicity yet.

CHAPTER 5

Summary and future work

Representational State Transfer is an architectural style, introduced by Roy Fielding as a post-hoc conceptualization of the World Wide Web. It is defined as a set of constraints whose implementation promises an application architecture which is highly scalable and evolvable. In order to support developers in creating distributed applications matching this architectural style, several frameworks have been created for the Java platform.

In order to evaluate the different frameworks, a maturity model was created. The most popular frameworks have been selected and classified in this maturity model. The classification is based on the implementation of a specified test application and special tests, which correspond to the different levels of the maturity model.

Unfortunately, none of the frameworks evaluated in this work actually fulfils all of a REST service developer's needs. They follow different approaches in their provided APIs, but at least all of them are fulfilling level two of the maturity model. Therefore they all allow for the creation of HATEOAS compliant client- and server-implementations, even though some frameworks are providing more advanced features than others.

The future work on this topic has to concentrate on the further development of the frameworks to continuously improve the simplicity and usability from a developer's perspective. Only when the developers are supported with proper tools, the architectural style and HATEOAS as a principle will find wide acceptance.

Another topic for further developments is the specification of media types that properly support linking. This problem is already being addressed, leading to new media type specification recommendations, like *JSON-LinkedData* (JSON-LD) [49] or *Hypermedia Application Language* (HAL) [47]. These media types are going to be used for even more advanced and evolvable REST applications [50].

The popularity of REST as an architectural style as well as the needs for highly scalable and modifiable distributed systems are increasing now more than ever. This will certainly lead to a change of the specification and framework developer's focus back to HATEOAS and linking in general.

Bibliography

[1] R. T. Fielding, “Architectural styles and the design of network-based software architectures,” Ph.D. dissertation, University of California, Irvine, 2000.

[2] P. Adamczyk *et al.*, “REST and Web Services: In Theory and in Practice,” in *REST: From Research to Practice.* Springer, 2011, pp. 35–57.

[3] R. T. Fielding. (2008, October) REST APIs must be hypertext-driven. [Online]. Available: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven [Accessed: 08-Jun-2016]

[4] R. T. Fielding *et al.*, “Hypertext transfer protocol – HTTP/1.1,” Internet Engineering Task Force, RFC 2616, June 1999. [Online]. Available: https://www.ietf.org/rfc/rfc2616 [Accessed: 08-Jun-2016]

[5] R. T. Fielding. (2009, November) rest-discuss entry about the development of REST. [Online]. Available: https://web.archive.org/web/20091111012314/http://tech.groups.yahoo.com/group/rest-discuss/message/6757 [Accessed: 08-Jun-2016]

[6] R. T. Fielding and R. N. Taylor, “Principled design of the modern web architecture,” *ACM Transactions on Internet Technology (TOIT)*, vol. 2, no. 2, pp. 115–150, 2002.

[7] T. Berners-Lee, R. T. Fielding, and L. Masinter, “Uniform resource identifiers (uri): Generic syntax,” Internet Engineering Task Force, RFC 2396, August 1998. [Online]. Available: https://www.ietf.org/rfc/rfc2396 [Accessed: 08-Jun-2016]

[8] L. Dusseault and J. Snell, “PATCH Method for HTTP,” Internet Engineering Task Force, RFC 5789, March 2010. [Online]. Available: https://tools.ietf.org/html/rfc5789 [Accessed: 08-Jun-2016]

[9] S. Vinoski, “RESTful web services development checklist,” *Internet Computing, IEEE*, vol. 12, no. 6, pp. 96–95, 2008.

[10] IANA. Media Types. [Online]. Available: https://www.iana.org/assignments/media-types/media-types.xhtml [Accessed: 08-Jun-2016]

[11] T. Berners-Lee and R. Cailliau. (1990) Worldwideweb: Proposal for a hypertext project. [Online]. Available: http://www.w3.org/Proposal.html [Accessed: 08-Jun-2016]

[12] M. Amundsen, “Hypermedia types,” in *REST: From Research to Practice.* Springer, 2011, pp. 93–116.

[13] C. Pautasso and E. Wilde, “Why is the web loosely coupled?: a multi-faceted metric for service design,” in *Proceedings of the 18th international conference on World wide web.* ACM, 2009, pp. 911–920.

[14] M. Kovatsch. (2016) Overview of HATEOAS approaches. [Online]. Available: https://www.ietf.org/proceedings/interim/2016/01/25/t2trg/slides/slides-interim-2016-t2trg-1-1.pdf [Accessed: 08-Jun-2016]

[15] M. Nottingham, “Web linking,” Internet Engineering Task Force, RFC 5988, March 2010. [Online]. Available: https://tools.ietf.org/html/rfc5988 [Accessed: 08-Jun-2016]

[16] IANA. Link Relations. [Online]. Available: http://www.iana.org/assignments/link-relations/ [Accessed: 08-Jun-2016]

[17] I. Robinson, “RESTful domain application protocols,” in *REST: From Research to Practice.* Springer, 2011, pp. 61–91.

[18] L. Richardson. (2008) Justice will take us millions of intricate moves. QCon talk. [Online]. Available: https://www.crummy.com/writing/speaking/2008-QCon/act3.html [Accessed: 08-Jun-2016]

[19] M. Fowler. (2010) Richardson maturity model: Steps toward the glory of rest. [Online]. Available: http://martinfowler.com/articles/richardsonMaturityModel.html [Accessed: 08-Jun-2016]

[20] F. Bülthoff and M. Maleshkova, “RESTful or RESTless–Current State of TodayâĂŹs Top Web APIs,” in *The semantic web: ESWC 2014 satellite events.* Springer, 2014, pp. 64–74.

[21] S. Vinoski, “REST Eye for the SOA Guy,” *Internet Computing, IEEE*, vol. 11, no. 1, p. 82, 2007.

[22] C. Pautasso, O. Zimmermann, and F. Leymann, “Restful web services vs. "big" web services: making the right architectural decision,” in *Proceedings of the 17th international conference on World Wide Web.* ACM, 2008, pp. 805–814.

[23] P. Oreizy, N. Medvidovic, and R. N. Taylor, “Runtime software adaptation: framework, approaches, and styles,” in *Companion of the 30th international conference on Software engineering.* ACM, 2008, pp. 899–910.

[24] C. McClanahan. (2006, April) Why HATEOAS. [Online]. Available: https://blogs.oracle.com/craigmcc/entry/why_hateoas [Accessed: 08-Jun-2016]

[25] M. Caporuscio, M. Funaro, and C. Ghezzi, “RESTful service architectures for pervasive networking environments,” in *REST: From Research to Practice.* Springer, 2011, pp. 401–422.

[26] D. Guinard, I. Ion, and S. Mayer, “In search of an internet of things service architecture: REST or WS-*? A developers’ perspective,” in *Mobile and Ubiquitous Systems: Computing, Networking, and Services.* Springer, 2011, pp. 326–337.

[27] J. Webber, S. Parastatidis, and I. Robinson, *REST in practice: Hypermedia and systems architecture.* O’Reilly Media, Inc., 2010.

[28] J. Thijssen *et al.* (2016) REST CookBook. [Online]. Available: http://restcookbook.com/ [Accessed: 08-Jun-2016]

[29] S. Parastatidis *et al.*, “The role of hypermedia in distributed system development,” in *Proceedings of the First International Workshop on RESTful Design.* ACM, 2010, pp. 16–22.

[30] M. Hadley, S. Pericas-Geertsen, and P. Sandoz, “Exploring hypermedia support in Jersey,” in *Proceedings of the First International Workshop on RESTful Design.* ACM, 2010, pp. 10–14.

[31] S. Pericas-Geertsen and M. Potociar, *JAX-RS: The Java API for RESTful Web Services*, 2013, JSR 339. [Online]. Available: https://jcp.org/en/jsr/detail?id=339 [Accessed: 08-Jun-2016]

[32] JaxRS project website. [Online]. Available: https://jax-rs-spec.java.net/ [Accessed: 08-Jun-2016]

[33] Jersey project website. [Online]. Available: https://jersey.java.net/ [Accessed: 08-Jun-2016]

[34] RestEasy project website. [Online]. Available: https://resteasy.jboss.org/ [Accessed: 08-Jun-2016]

[35] Spring HATEOAS project website. [Online]. Available: http://projects.spring.io/spring-hateoas/ [Accessed: 08-Jun-2016]

[36] A. Rashed. (2016, Jun.) Evaluation of HATEOAS support by Java frameworks. GitHub repository. [Online]. Available: https://github.com/alexrashed/eval_rest [Accessed: 24-Jun-2016]

[37] A. Rashed, “Evaluation sourcecode - master branch,” Jun. 2016. [Online]. Available: http://dx.doi.org/10.5281/zenodo.56344 [Accessed: 24-Jun-2016]

[38] A. Rashed, “Evaluation sourcecode - test base branch,” Jun. 2016. [Online]. Available: http://dx.doi.org/10.5281/zenodo.56345 [Accessed: 24-Jun-2016]

[39] A. Rashed, “Evaluation sourcecode for RFMM level 1,” Jun. 2016. [Online]. Available: http://dx.doi.org/10.5281/zenodo.56346 [Accessed: 24-Jun-2016]

[40] A. Rashed, “Evaluation sourcecode for RFMM level 2,” Jun. 2016. [Online]. Available: http://dx.doi.org/10.5281/zenodo.56347 [Accessed: 24-Jun-2016]

[41] A. Rashed, “Evaluation sourcecode for RFMM level 3,” Jun. 2016. [Online]. Available: http://dx.doi.org/10.5281/zenodo.56348 [Accessed: 24-Jun-2016]

[42] A. Rashed, “Evaluation sourcecode for RFMM level 4,” Jun. 2016. [Online]. Available: http://dx.doi.org/10.5281/zenodo.56349 [Accessed: 24-Jun-2016]

[43] A. Rashed, “Evaluation sourcecode for RFMM level 5,” Jun. 2016. [Online]. Available: http://dx.doi.org/10.5281/zenodo.56350 [Accessed: 24-Jun-2016]

[44] K.-m. Chung, *Expression Language Specification*, 2013, JSR 341. [Online]. Available: https://jcp.org/en/jsr/detail?id=341 [Accessed: 08-Jun-2016]

[45] M. Nottingham and R. Sayre, “The Atom Syndication Format,” Internet Engineering Task Force, RFC 4287, December 2005. [Online]. Available: https://www.ietf.org/rfc/rfc4287 [Accessed: 08-Jun-2016]

[46] M. Karg *et al.* (2014) JSR 370 expert group discussion on hypermedia APIs. [Online]. Available: http://java.net/projects/jax-rs-spec/lists/jsr370-experts/archive/2014-12/message/2 [Accessed: 08-Jun-2016]

[47] M. Kelly. (2016) JSON Hypertext Application Language. [Online]. Available: https://tools.ietf.org/html/draft-kelly-json-hal [Accessed: 08-Jun-2016]

[48] M. Fischer, K. Képes, and A. Wassiljew, “Vergleich von Frameworks zur Implementierung von REST-basierten Anwendungen,” Studienarbeit, Universitẗ Stuttgart, 2013, German.

[49] M. Lanthaler, M. Sporny, and G. Kellogg, “JSON-LD 1.0,” W3C, W3C Recommendation, Jan. 2014. [Online]. Available: http://www.w3.org/TR/2014/REC-json-ld-20140116/ [Accessed: 08-Jun-2016]

[50] M. Lanthaler and C. Gütl, “On using JSON-LD to create evolvable RESTful services,” in *Proceedings of the Third International Workshop on RESTful Design.* ACM, 2012, pp. 25–32.