DEGREE PROJECT, IN COMPUTER SCIENCE , SECOND LEVEL
STOCKHOLM, *SWEDEN 2015*

RESTful Services in an Enterprise Environment

A COMPARATIVE CASE STUDY OF SPECIFICATION FORMATS AND HATEOAS

ROBERT WIDEBERG

KTH ROYAL INSTITUTE OF TECHNOLOGY

SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION (CSC)

RESTful Services in an Enterprise Environment

A Comparative Case Study of Specification Formats and HATEOAS.

REST-tjänster i en enterprise-miljö

En jämförande fallstudie av specifikationsformat och HATEOAS

ROBERT WIDEBERG
ROBWID@KTH.SE

DD221X, Degree Project in Computer Science, Second Cycle, 30 credits
Master's Programme, Computer Science, 120 credits
Degree Programme in Computer Science and Engineering, 300 credits
Supervisor at CSC was Jeanette Hellgren Kotaleski
Examiner was Anders Lansner
Principal was Scania IT

August 28, 2015

Abstract

RESTful services are becoming increasingly popular. This work, that was carried out at Scania IT, investigates how a RESTful service should be designed and specified so that it meets the demands of an enterprise environment. In particular, this report will focus on documentation and validation of RESTful services, i.e. specifications. These are important aspects in an enterprise environment. The investigation consists of a comparative case study of four specification formats and Hypermedia As The Engine Of Application State (HATEOAS).

The results show that the most commonly used specification formats have flaws. They also suggest that Swagger and RAML (RESTful API Modeling Language) are the two most mature formats. The results also show that HATEOAS, which is a more dynamic approach, can be useful in an enterprise environment but that it requires careful design.

Referat

REST-tjänster i en enterprise-miljö

REST-tjänster blir alltmer populära. Detta arbete, som utfördes på Scania IT, undersöker hur en REST-tjänst bör designas och specifieras för att möta de krav som finns i en enterprise-miljö. Specifikt kommer denna rapport att fokusera på dokumentation och validering av REST-tjänster, med andra ord specifikationer. Dessa är viktiga aspekter i en enterprise-miljö. Undersökningen består av en jämförande fallstudie med fyra specifikationsformat samt Hypermedia As The Engine Of Application State (HATEOAS).

Resultaten visar att de mest vanligt förekommande specifikationsformaten har brister. De antyder också att Swagger och RAML (RESTful API Modeling Language) är de två mest mogna formaten. Resultaten visar också att HATEOAS, som är ett mer dynamiskt angreppssätt, kan vara användbart i en enterprise-miljö men att det kräver omsorgsfull design.

Preface

This report presents a Master Thesis in Computer Science at the Royal Institute of Technology, KTH. The majority of the work was performed at Scania IT in Södertälje during the spring of 2015. The supervisor at Scania IT was Olle Sundblad, Lead Software Architect Java. Professor Jeanette Hellgren was the supervisor at KTH. The examiner of this work was Professor Anders Lansner.

I would like to thank Olle Sundblad for helping me in formulating the problem statement and aim for this thesis. In particular for bringing me to meetings where I was able to understand more of what the situation at Scania IT looks like. I would also like to thank the entire Groundhogs team at Scania IT for sharing offices with me and making the work more enjoyable.

In addition, I would also like to thank Jeanette Hellgren for her feedback and input during this work and Anders Lansner for his valuable feedback on the report.

Contents

Chapter 1

Introduction

1.1 Background and Motivation of the Thesis

RESTful services are becoming increasingly popular in the software industry. They are mostly used to expose system functionality as public APIs or to connect back- and front-ends of systems. Less common is the use of RESTful services for integrating systems in enterprise environments, which is the focus of this report.

The work in this report is of interest for Scania IT since the company has around a thousand applications (there is no exact figure). These need to integrate with each other and one way of doing this is to use REST. The application environment at Scania IT is heterogeneous in nature with Java and .NET solutions being the most common. This leads to a need for standardization of their integrations. Traditionally SOAP, Simple Object Access Protocol, has been used for these types of integrations but new RESTful services are currently being developed at the company. Standardizing the documentation makes it easier for consumers to quickly understand new services. There is currently also a procedure for validating the SOAP WSDL (Web Service Description Language) files at Scania IT. Both to make sure that they follow the standards for SOAP services and to validate incoming messages to the service. There is an interest in finding out how the same procedures can be applied to RESTful services.

Scania IT is in no sense unique as a large IT company in which the organization has adopted a traditional SOA (Service Oriented Architecture) approach with SOAP as the standard message protocol. However, agile development teams can find it difficult to fit their work process into this model. There is a reasonable fear from the organization that the new types of integrations will not hold up to the required standards and an understandable frustration from the developers who want to adopt a RESTful architectural style. The work that is presented in this report can be a first step on the road toward resolving these issues and make it easier to adopt a RESTful architectural style for enterprise integrations.

1.2 Problem Statement

The question that lays the foundation for this work is as follows:

How should a RESTful service be designed and specified so that it meets the demands of an enterprise environment?

1.3 Aim of the Thesis

The specific aim of this thesis is to evaluate and compare four different specification formats for RESTful services and also to investigate HATEOAS which is an abbreviation of Hypermedia As The Engine Of Application State. The evaluation criteria is based on what would be reasonable requirements in an enterprise environment.

1.4 Thesis Structure

This report consists of a background section that gives a general introduction to services and in particular RESTful services. The different specification formats will be presented along with an introduction to HATEOAS. The method section follows and explains how the results in the following result section were derived. At the end there is a discussion section that reflects upon the results and presents some conclusions as well as some speculation about the future in this area.

1.5 Demarcations

No guarantee is given that this report examines all possible tools and approaches when answering the problem definition. For instance, WADL (Web Application Description Language, see Section 2.2.6) is not the only XML-based (Extensible Markup Language) format for describing RESTful services. It is also possible to use WSDL 2.0. However, this format requires more work for no additional benefits. For the purpose of limiting the scope of this report that format was excluded from examination.

The case study in this report is also limited to a single system for practical reasons. Implications of this limitation is discussed in Section 3.3.3.

1.6 List of Abbreviations

- **DDOS** Distributed Denial Of Service
- **DFS** Depth First Search
- **DMZ** De-Militarized Zone
- **ESB** Enterprise Service Bus

- **HAL** Hypertext Application Language
- **HATEOAS** Hypermedia As The Engine Of Application State
- **HTTP** Hypertext Transfer Protocol
- **JAX-RS** Java API for RESTful Web Services
- **JSON** JavaScript Object Notation
- **MEM** Maintenance Expert Management
- **RAML** RESTful API Modeling Language
- **RDF** Resource Description Format
- **REST** REpresentational State Transfer
- **RFC** Request For Comments
- **RPC** Remote Procedure Call
- **SOA** Service Oriented Architecture
- **SOAP** Simple Object Access Protocol
- **URL** Uniform Resource Locator
- **WADL** Web Application Description Language
- **WSDL** Web Service Description Language
- **XML** Extensible Markup Language
- **XSLT** eXtensible Stylesheet Language Transformation
- **YAML** YAML Ain't Markup Language

Chapter 2

Background

2.1 Services

The term service has a broad interpretation in the computer science field. It could be referring to anything that performs business logic, file sharing, authentication, logging etc [6]. However, this report is in particular concerned with services for integrating and sharing logic between systems in an enterprise environment.

2.1.1 The Movement Towards Microservices

A trending phenomenon when it comes to services is microservices. Microservices are small services that have their own single responsibilities. Together several microservices can create a resilient distributed system. This is to be put into contrast to the traditional large monolithic system. The rise of microservices follows the popularity of continuous delivery and the idea of having small development teams that can handle the lifecycle of their application by themselves [14]. The popularity of REST follows from these trends as well. Its simplicity and flexibility allows for swift changes and thus continuous improvement.

2.2 REST

2.2.1 Definition

REST is an abbreviation of REpresentational State Transfer. It is an architectural style and not a protocol. The distinction is important since this means that even though REST is strongly associated with the Hypertext Transfer Protocol (HTTP) it is not necessarily dependent on it. However, the successfulness of the web was clearly the inspiration and motivation for REST when it was presented by Roy T. Fielding in an article published in 2000 [7]. In it he points out three main strengths with REST:

- Separation of concerns between client and server style.

- Scalability.
- Information hiding.

This is achieved with standardized representations and metadata. A RESTful server and client communicate by sending representations of data using some well defined format that both dynamically agree on that they can support. It could be a HTML-document or a JPEG image, a mediatype such as `application/json` (JavaScript Object Notation), `application/xml` or perhaps a more specific mediatype like `application/hal+json`. Since the representation is of a well defined format no additional information needs to be sent in order for the recipient to interpret incoming data which leads to less overhead in the data transfer, thus improving scalability. Another scalability factor is that the interaction between server and client is stateless. Since representations of data are the only things that are sent between them there is no need for the server to store any particular client state. It also reduces the complexity of processing requests in parallel. This can be compared to RPC (Remote Procedure Call) where the client is invoking methods that are executed on the server. In that case execution state has to be stored. Finally, some responses can also be cached for improved performance, not that it is impossible to cache responses in a non-RESTful service but caching rules are well defined for the HTTP protocol. Information hiding often comes natural with a RESTful architecture since the original business data needs to be converted to some representational format and how the service actually is designed can be hidden behind that format.

2.2.2 Resources and URIs

The data that is communicated in RESTful services is referred to as *resources*. Which particular resource that a request is concerning is determined by a URL (Uniform Resource Locator). A certain type of resource could be located under some specific path and identifiers for specifying a single resource could be put in variable path and/or query parameters.

URL 1: http://example.scania.com/v1/vehicles

URL 2: http://example.scania.com/v1/vehicles/{chassino}

URL 3: http://example.scania.com/v1/vehicles/{chassino}?lang={language}

Consider the above URLs in which path and query parameters are indicated with curly brackets. Let us assume that these belong to a RESTful service that provides information about vehicles. **URL 1** could be used for retrieving a list of all vehicles or adding new ones. **URL 2** utilizes a path parameter to specify a particular chassis, this could be used for retrieving or possibly deleting/modifying that chassis. **URL 3** adds a query parameter that could be used to specify the response language. Query parameters are usually optional, so in this example it could be that there is a default language for all responses unless something else is specified.

Verb	Idempotent	Usage
HEAD	Yes	Requesting that only the response header is returned and not the body.
OPTIONS	Yes	Probing what HTTP verbs that can be used for a particular resource.
GET	Yes	Retrieving a representation of a particular resource.
POST	No	Creating a new resource. (This is the most common use case for POST but it is also the all-round verb that is used when there is no other suitable verb).
PUT	No	Updating a resource. Could also be used for creation when the client is deciding the resource URL itself.
DELETE	No	Deleting a resource.
PATCH	No	Partially updating a resource. (This operation should be used with care since it might not be supported).

Table 2.1. Listing of HTTP verbs and how they should be used in a RESTful service [2]. Idempotent means safe/read-only.

2.2.3 HTTP Verbs and Status Codes

The HTTP protocol offers seven verbs (request methods) that can be utilized when designing a RESTful service, see Table 2.1. The knowledgeable reader might object and say that there are actually nine verbs in the HTTP/1.1 protocol, however TRACE and CONNECT are not interesting in the context of REST. According to the Richardson maturity model (see Section 2.2.8) how verbs and status codes are used matters and is a sign of how mature a service is.

The HTTP status codes can be used to indicate whether a request succeeded or failed and the reason why that is.

2.2.4 Alternatives

REST is often compared with SOAP even though SOAP is a protocol and not an architecture. However, there seems to be a division between proponents of RESTful architectures and services built on SOAP [13]. Some argue that using SOAP leads to better information hiding and better reusability. As described in section 2.2.1, information hiding and loose coupling is possible to achieve with RESTful services as well. In fact, this is more of a design issue rather than a protocol issue.

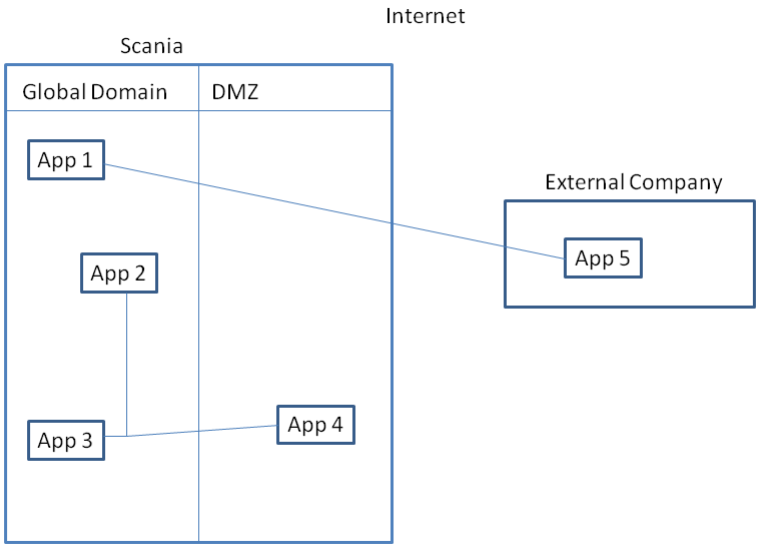

Figure 2.1. Overview example of how services can be integrated at Scania.

2.2.5 Validation on an ESB

Scania uses an Enterprise Service Bus (ESB) for integration of their systems. The primary tasks of an ESB is to monitor, control and validate traffic to and from services. All interactions with services at Scania IT needs to go through their ESB, see Figure 2.1. This policy is especially important for interactions that passes through the De-Militarized Zone (DMZ) to the internal systems. If a Distributed Denial Of Service (DDOS) attack would occur it is possible to deny all accesses from the DMZ since they must all pass through the ESB.

An issue at Scania is that there is currently no support for validating REST calls on their ESB. There is support for validating calls to SOAP services by cross-referencing them with WSDL-files. To summarize, the validation problem has merit [1] and hence it will be investigated in this report. With REST not only the actual data needs to be validated but also the resource identifiers, i.e. the URL-parameters.

2.2.6 Specification for Documentation and Validation

Documentation is important for ease of integrating new systems with each other. However, if the documentation is outdated it can have the opposite effect. Hence, there is often a need for auto-generating it from the source code in order for the documentation to reflect the actual behavior of the service. Alternatively the source code can be auto-generated from a specification from which a documentation can be derived.

Available Solutions

There are tools for generating/writing specifications of RESTful services. Some are more mature than others. Below follow short descriptions of available solutions and examples of how they would be used to describe a simple example service that provides information about a set of vehicles in different languages.

Swagger

Swagger is an open source project previously backed by Reverb until April 2015 when SmartBear announced that they had acquired it. Also, Microsoft announced in April 2015 that its Azure API Apps Tool will make use of Swagger to describe APIs [8]. Swagger can expose the structure of a REST API as JSON via REST. It is also possible to write a Swagger specification in YAML (YAML Ain't Markup Language) in the online Swagger editor [21], see Code 2.1 for an example.

Code 2.1. Swagger example

```
swagger: '2.0'
info:
  title: Scania Example API
  description: This is just an example API
  version: "1.0"
schemes:
  - http
host: example.scania.com
basePath: /v1
produces:
  - application/json
paths:
  /vehicles:
    get:
      summary: Vehicles
      description: This endpoint returns all vehicles
      parameters:
        - name: lang
          in: query
          description: Specifies the response language
          required: false
          type: string
      responses:
        200:
          description: An array of vehicles
          schema:
            type: array
            items:
              $ref: '#/definitions/Vehicle'
definitions:
  Vehicle:
    properties:
      chassino:
        type: string
        description: Unique identifier of chassis.
      vehicletype:
        type: string
        description: The vehicle type.
```

RAML

RAML is an abbreviation of RESTful API Modeling Language and is an open source project created by MuleSoft. RAML is a YAML-based specification format [16]. See

Code 2.2 for an example.

Code 2.2. RAML example

```
#%RAML 0.8
title: Scania Example API
baseUri: http://example.scania.com/{version}
version: v1

/vehicles:
  get:
    queryParameters:
      lang:
        type: string
        description: Specifies the response language
        required: false
    responses:
      200:
        body:
          application/json:
            schema: |
              {
                 "$schema": "http://json-schema.org/schema",
                 "type": "array",
                 "description": "An array containing all vehicles",
                 "items":{
                    "properties": {
                        "chassino": { "type": "string" },
                        "vehicletype": { "type": "string" }
                     },
                     "required": [ "chassino", "vehicletype" ]
                 }
              }
```

API Blueprint

API Blueprint is also an open source project. It is a Markdown-based specification format [3]. See Code 2.3 for an example.

Code 2.3. API Blueprint example specification.

```
FORMAT: 1A
HOST: http://example.scania.com/v1

# Scania Example API

# Resource [/vehicles{?lang}]
## Retrieve all vehicles [GET]
+ Response 200 (application/json)
    This operation will retrieve a list of all vehicles

    + Parameters
        + lang(optional, string) ... Specifies the response language

    + Schema

            {
              "$schema": "http://json-schema.org/schema",
              "type": "array",
              "description": "An array containing all vehicles",
              "items":{
                "properties": {
                   "chassino": { "type": "string" },
                   "vehicletype": { "type": "string" }
                    },
                "required": [ "chassino", "vehicletype" ]
                }
            }
```

WADL

WADL or Web Application Description Language is an XML-based specification of HTTP-based web applications. It was designed by Sun Microsystems and the latest version was submitted to W3C in 2009 [10]. See Code 2.4 and 2.5.

Code 2.4. Vehicle.xsd file. The XML-schema for a vehicle.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="VehiclesList">
    <xs:sequence>
      <xs:element name="Vehicle" maxOccurs="unbounded" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="chassino" type="xs:string"/>
            <xs:element name="vehicletype" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Code 2.5. The WADL specification example.

```
<wadl:application xmlns:wadl="http://wadl.dev.java.net/2009/02"
    xmlns:v="http://example.scania.com/schema/vehicle Vehicle.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://wadl.dev.java.net/2009/02 wadl.xsd ">

   <wadl:doc title="Scania Example API">
     This is just an example API.
   </wadl:doc>
   <wadl:grammars>
     <wadl:include href="Vehicle.xsd"/>
   </wadl:grammars>

   <wadl:resources base="http://example.scania.com/v1">
     <wadl:resource path="vehicles">
       <wadl:doc title="Description">This operation will retrieve a list of all vehicles</wadl:doc>
      <wadl:method name="GET" id="getListOfVechicles">
         <wadl:request>
           <wadl:param name="lang" type="xsd:string" style="query"
           required="false">
           <wadl:doc>Specifies the response language</wadl:doc>
           </wadl:param>
         </wadl:request>
         <wadl:response status="200">
           <wadl:representation mediaType="application/xml"
           element="v:VehiclesList">
           <wadl:doc>An array of vehicles</wadl:doc>
           </wadl:representation>
         </wadl:response>
       </wadl:method>
     </wadl:resource>
   </wadl:resources>
</wadl:application>
```

2.2.7 Auto-generation contra Contract-first

All of the tools above support a contract-first approach. Meaning that it is possible to type up the specification before any code is written and possibly also auto-generate code from it. However, the contract-first approach often leads to problems

with keeping the specification up do date, as mentioned in Section 2.2.6. There is, on the other hand, often support for going the other way, i.e. auto-generating the specification from the source code.

2.2.8 Richardson's Maturity Model and HATEOAS

In his 2008 Qton talk Leonard Richardson defined the following maturity levels of RESTfulness [17]:

- **Level 0:** The service merely mimics traditional RPC. Each invocation is performed with HTTP POST operations and XML-encoded messages to the same URL.
- **Level 1:** The service utilizes multiple resources and encodes method names and parameters in the URL. HTTP GET is normally used for operations.
- **Level 2:** In addition to level 1 the service is now fully utilizing the HTTP verbs and status codes.
- **Level 3:** In addition to level 2 the service is presenting the next possible actions to the client in its responses (HATEOAS).

HATEOAS is an abbreviation for Hypermedia As The Engine Of Application State and is also known as the hypermedia constraint. In essence it means that the service is supplying links in its responses that represent the next possible actions that the consumer can take. The links can be supplied in the message header, normally in the location header field, or in the body [12], see Code 2.6 for an example. The purpose of these links is to provide the consumer of the service with possible new interactions. For instance, a GET operation on the base URL can return a response body that lists all resources that can be of interest to start interacting with. The idea is that this will make the service self-explanatory. Given a single base URL to start with a consumer of the service should be able to navigate through it and perform its desired operations. In other words, there is no need for any additional documentation. The question is just if this is possible to achieve in complex services and, if it is, how long it takes to reach that level of maturity.

Code 2.6. An example of how links can be included in a response. Here an example vehicle resource in JSON has links to itself and to another supposedly related vehicle resource.

```
{
  "chassino":"EX1234",
  "vehicletype":"truck",
  "links":[
    {
      "href":"http://example.scania.com/v1/vehicles/EX1234",
      "rel":"self"
    },
    {
      "href":"http://example.scania.com/v1/vehicles/EX1234567",
      "rel":"related"
    }
  ]
}
```

Definition of Links

Unfortunately, the idea of just providing a base URL and then guide the client with links is not as trivial as it sounds. Somehow the consumer of the REST service needs to know what a link looks like. Two formats for describing links, Atom and HAL, are presented in the following sections.

Atom Link Format

Atom is an XML-based format that is used for describing web feeds and was developed as an alternative to RSS. However, in the Atom format there is a standardized way of defining links as elements. See Table 2.2 for a list of attributes [15] and Code 2.7 for an example. It is easy to define the same format in JSON, in fact Code 2.6 is an example of that.

Attribute	Description
href	The URL.
rel	The relation type of the link. This describes the meaning of the link.
type	The expected media type to be returned from the linked resource.
hreflang	The language of the of the linked resource.
title	Human readable description of the linked resource.
length	An advisory length of the linked resource.

Table 2.2. Listing of attributes that are defined for the atom:link element. Note that only **href** and **rel** are supported in Spring HATEOAS.

Code 2.7. Example of a link on Atom format.

```
<atom:link rel="self" href="http://example.scania.com/vehicles/EX1234"/>
```

HAL

Hypertext Application Language (HAL) was designed to create a uniform practice of how to express e.g. links in a Web API. So in contrast to Atom it was designed to solve exactly this standardized notation problem. When researching the format closer it is clear that there is no finalized RFC (Request for Comments) for HAL and that there is in fact only an Internet Draft describing the JSON format, which expired in April 2014, available [11]. As an example, the response with links on atom format in Code 2.6 will look like in Code 2.8 when the links are on the HAL format.

Code 2.8. Example of a vehicle represented in JSON with links on HAL format.

```
{
   "chassino":"EX1234",
   "vehicletype":"truck",
   "_links":{
      "self":{
         "href":"http://example.scania.com/vehicles/EX1234"
      },
      "related":{
         "href":"http://example.scania.com/vehicles/EX1234567"
      }
   }
}
```

The Semantic Gap

A common idea with HATEOAS is that it should make a service possible to be automatically discovered by an autonomous agent. The difficulty with this is what is commonly referred to as "The Semantic Gap". Web pages guide users through its content with links just the same way as Richardson proposes that a RESTful service should guide its consumers. However, humans can understand the title of a link and make an informed decision about whether it is worth clicking on or not. Such decisions are difficult for a machine to make without extensive hard-coding [18].

2.2.9 Previous Work With RDF-based Formats

There is previous research on the area of HATEOAS and automatic discovery performed by Steiner et al. [20] in which HTTP OPTIONS and Atom links were utilized together with RDF (Resource Description Format). RDF is the core language of the semantic web. A similar study presented an RDF based description format called RESTdesc [22] [23]. RESTdesc is based on Notation3, which is a RDF extension that makes it possible to express modus ponens inferences. This means that a full result chain of an operation on a resource can be expressed. In other words, what implications a particular operation will have. This enables context-based service discovery, see Code 2.9 for an example.

Code 2.9. Example of the RESTdesc description format. Schemas can be imported with the prefix annotation. **If** your image has a smallThumbnail link to a resource **then** there exists a GET request to that resource resulting in a response **that** will be an 80 pixels high thumbnail of your image. This example is fetched from http://restdesc.org/about/descriptions.

```
@prefix : <http://example.org/image#>.
@prefix http: <http://www.w3.org/2011/http#>.
@prefix dbpedia: <http://dbpedia.org/resource/>.
@prefix dbpedia-owl: <http://dbpedia.org/ontology/>.
{
  ?image :smallThumbnail ?thumbnail.
}
=>
{
  _:request http:methodName "GET";
            http:requestURI ?thumbnail;
            http:resp [ http:body ?thumbnail ].
  ?image dbpedia-owl:thumbnail ?thumbnail.
  ?thumbnail a dbpedia:Image;
             dbpedia-owl:height 80.0.
}.
```

It is clear that this format has not been widely adopted, there are only a few inactive GitHub projects that use it. These formats were not used in the case study performed in this report since they are trying to achieve goals that are out of scope for what is investigated. They would be more relevant to investigate in a thesis about the semantic web. However, they are worth mentioning and are also discussed with respect to the results of this study in Section 5.3.

Chapter 3

Method

3.1 Methodology of the Work

The results in this report are supported by two methods. One is a literature study that gives support to the content of this report and the conclusions of this work. The other is a case study in which the different solutions are evaluated on a specific system at Scania IT which provides the results of this work.

3.1.1 Motivation of Methods

The literature study was critical in order to support the content of this report. It was necessary to give the author an understanding of what has been done previously in the area and what would be relevant to explore further. A case study on a particular system was a necessary limitation of the study. Implementing the solutions in more than one system would make the scope too large. The case study was however a good complement to the literature study since it put the solutions to test with real business data and logic.

3.2 Evaluation

3.2.1 Criteria for the Specification Formats

The implementation case study will shine light upon the aspects listed below that are used to evaluate the different specification formats. The format versions that are investigated in this study are Swagger 2.0, RAML 0.8, API Bluprint 1A and WADL 2009 W3C Submission.

- **Complexity.** How easy is it to integrate the solution into a working system without introducing complexity?
- **Support for testing.** Does the specification format offer any support for testing?

- **Validation capabilities.** Is there support for specifying input length, type and other common validation parameters? Is it possible to reference JSON- or XML-Schemas for this validation.
- **Generation of a UI for documentation.** What is the support like for generating a GUI with HTML/CSS/JavaScript to document and possibly test the service?
- **Future proof.** How active is the community/company that is backing the format? This can be an indicator about possible future development and support.

3.2.2 Criteria for HATEOAS

HATEOAS is evaluated with the same complexity aspect that is mentioned in Section 3.2.1. This study is also attempting to investigate how HATEOAS can be useful in an enterprise environment.

3.3 Maintenance Expert Management

The different specification formats and HATEOAS were tested in a system called MEM (Maintenance Expert Management). MEM is currently exposing SOAP-services to external systems. REST is only used for communication between its front- and back-end. MEM is a part of a part of a larger project at Scania IT that focuses on software for providing dynamical and adaptable plans for maintenance of various Scania products.

3.3.1 Implementation Details

MEM has a Java back-end and implements its RESTful services with the Spring framework. This should be sufficient information about the MEM technical stack in order to replicate the case study results presented in this report.

3.3.2 Motivation for Selecting the System

Testing on a real enterprise system increases the credibility of the results. MEM was chosen since its architecture is closer to the vision that the development team at Scania IT where this study was carried out has. There is also an ambition that the services that is currently exposed with SOAP to external systems should be available as RESTful services in the future.

3.3.3 Scope and the Possibility to Replicate Results

Since the system is built in Java the results are dependent on how well the different solutions are adapted for a Java environment and specifically with the Spring

framework. It should be noted that the system and its services are not publicly available so replication of the results are strictly limited to other systems with an equivalent technical stack.

3.4 How the Specification Formats were Evaluated

The formats were first tested for the code to specification auto-generation approach. If this was not possible a contract-first approach was used. However, in this case just a subset of all resources was specified since it is too time consuming to specify all of them by hand. Different approaches for testing are then investigated, e.g. testing via the HTML/CSS/JavaScript documentation or via SoapUI. If the specification could not be auto-generated it would be interesting to be able to write unit tests in the code that validate that the actual code matches the specification. Validation is not strictly tested for how well it would work on an ESB bus but rather if it is possible to specify validation parameters at all. In any case it would not be wise to have a validation scheme that is so tightly coupled to the interpreting technology. Even if the bus technology is replaced it should still be possible to perform the validation.

The complexity aspect that is listed in Section 3.2.1 receives more of a collective judgement based on how much complexity that was necessary to introduce in order to test out the tools. The other aspects that are listed in that section are binary in nature, either there is support for them or not. The results will later be discussed in Chapter 5 and conclusions will be drawn regarding how mature the different solutions are.

3.5 How HATEOAS was Investigated

HATEOAS was investigated by implementing it for a subset of the resources in MEM, exactly like with the specification formats that did not support auto-generation.

The author created a simple application that parses and presents all links, defined on Atom or HAL format, in service responses[1]. It also presents the result of a HTTP OPTIONS request for the same resource and presents buttons for each of the allowed verbs, see Figure 3.1. This application was developed in order to make it easier to observe and resolve design flaws in the HATEOAS implementation.

The application can also auto-discover and present the service as a graph with resources as nodes and link relations as edges, see Figure 3.2. This functionality was built as a proof of concept for how service interaction can be automated.

[1] https://github.com/robbanw/hateoas-explorer

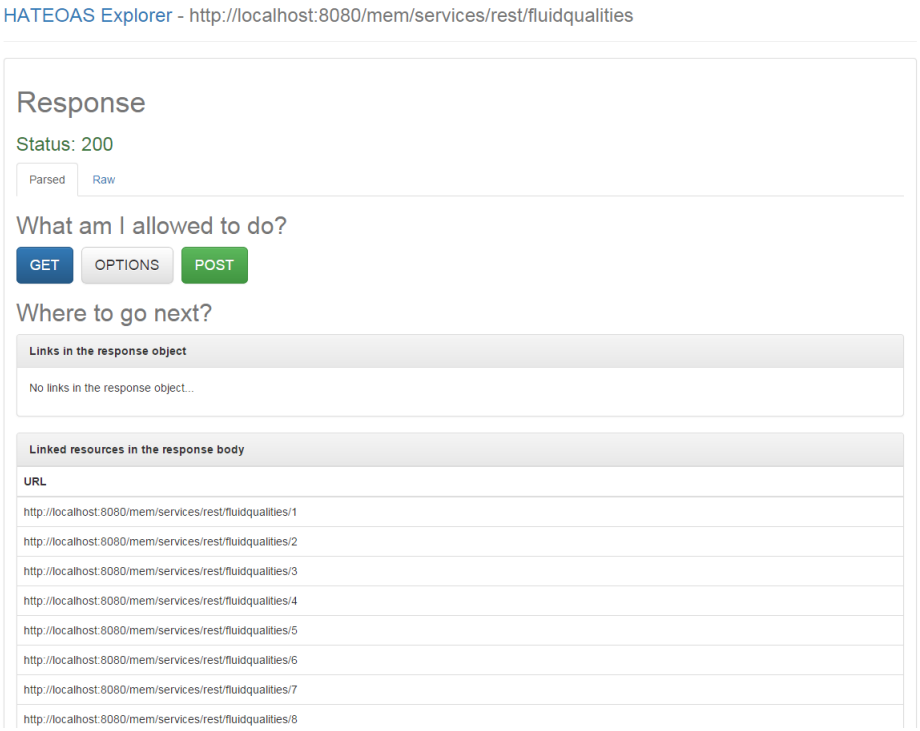

Figure 3.1. The explorer tool used on a MEM resource.

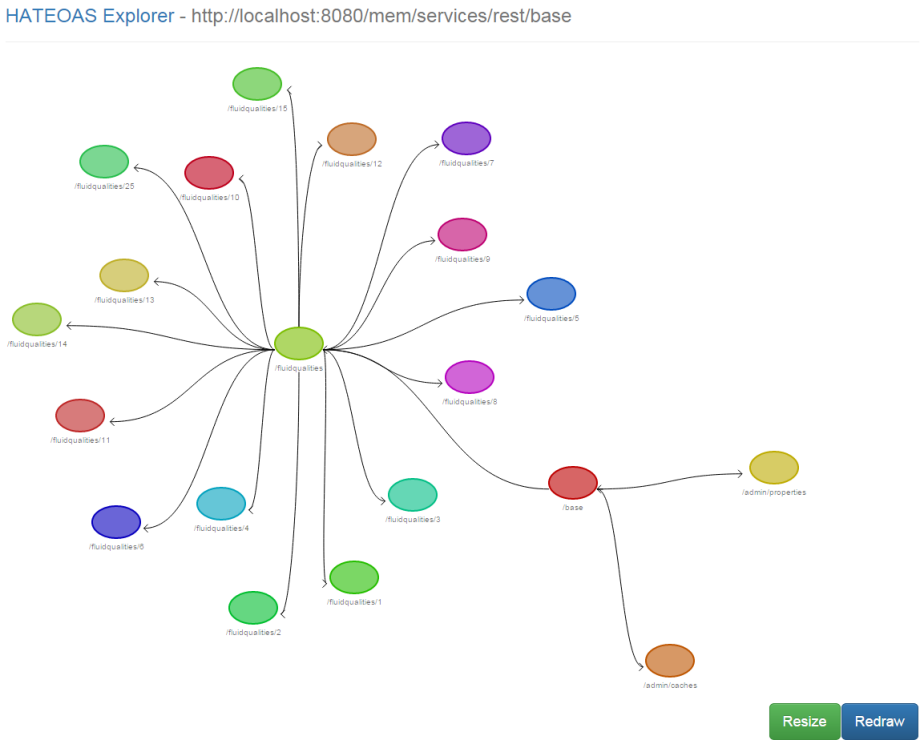

Figure 3.2. The explorer auto-discovery tool used on the MEM REST service.

Chapter 4

Results

4.1 General Capabilities of the Formats

In this section general results and findings about the formats will be presented. These are important in order to understand how the particular choice of system affected the results of the case study. As can be seen in Table 4.1, Swagger has the best support for Scania ITs main development languages Java and .NET when it comes to auto-generation. In fact, Swagger has support for more languages than the other formats. Also, SoapUI, which is one of the main tools used for testing services at Scania IT has support for all formats in the Pro version but only WADL in the free version. The GitHub search results, though not a fool proof measurement, indicate that it is the most popular format in the open source community.

	Swagger	**RAML**	**API Blueprint**	**WADL**
Java Auto-Generation	Yes	Yes*	No	Yes*
.NET Auto-Generation	Yes	No	Yes	No
Nr of languages supported	>10	>5	3	1
Support in SoapUI	Yes**	Yes**	Yes**	Yes
Search hits on GitHub* **	1143	355	189	162

Table 4.1. Supported languages refers to auto-generation of the specification or vice versa. * = Only supporting JAX-RS annotations (not Spring). ** = Only supported via plugin downloads in the Pro version. *** = Retrieved from https://github.com/ in May 2015.

4.2 Auto-Generation

It turns out that for the particular technical stack that MEM has only Swagger has support for auto-generation. RAML and WADL has support for auto-generation from Java code but only for JAX-RS (Java API for RESTful Web Services) annotations and not for Spring REST annotations. API Blueprint is currently not

supporting Java at all. HATEOAS is not evaluated in regards to this aspect but as mentioned in Section 4.7.1 there exists a support tool for generating HATEOAS links in a Java Spring environment.

4.2.1 Swagger Spring MVC Plugin

The Swagger Spring MVC Plugin (springfox)[1] is a piece of open-source software that can parse Spring RestControllers and expose their functionality as a Swagger specification. In the best case scenario it is only necessary to add an @EnableSwagger annotation in a Spring @Configuration annotated class to get it working. However, in order to set an appropriate title and version information in the generated specification there is need for some additional configuration. Also, since the Spring DispatcherServlet is configured, in a web.xml, to handle requests under the endpoint "/rest" in MEM it is necessary to explicitly define that in the configuration in order to get correct endpoints in the generated Swagger specification, see Code 4.1.

Code 4.1. This code snippet shows how Swagger was configured in MEM

```
@Autowired
private SpringSwaggerConfig swaggerConfig;

private ServletContext context;

@Bean
public SwaggerSpringMvcPlugin swaggerConfig(){
   ApiInfo apiInfo = new ApiInfo("MEM", "This is a documentation of the MEM REST API","","","","");
   RelativeSwaggerPathProvider pathProvider = new RelativeSwaggerPathProvider(context);
   pathProvider.setApiResourcePrefix("rest");
   return new SwaggerSpringMvcPlugin(this.swaggerConfig).apiInfo(apiInfo).pathProvider(pathProvider);
}
```

The swagger Spring MVC plugin will group the service operations according to the RestController class that they belong to. Additional documentation and configuration can be added with annotations, see Table 4.2.

Annotation	Description
@Api	Can be used to add additional documentation to RestControllers.
@ApiIgnore	RestControllers with this annotation will not be parsed and exposed. Can be used to hide resources.
@ApiOperation	Can be used to add additional documentation for an operation on a resource.
@ApiResponse	Can be used to specify and document the different response codes that an operation on a resource can return.

Table 4.2. Annotations that can be used in conjunction with the Swagger Spring MVC plugin.

[1] https://github.com/springfox/springfox

4.3 Contract First

Since it was possible to auto-generate Swagger definitions that format will not be investigated in depth in this section even though it can be said that it is possible to have a contract-first workflow with Swagger as well, as explained in Section 2.2.6. The following section will go into detail about using a contract-first approach for each of the other specification formats. These will suggest that RAML and API Blueprint specifications are more readable and less verbose than their WADL counterpart, which is supported by the results displayed in Table 4.3.

Format	Number of Characters
API Blueprint	885
RAML	752
WADL	2769

Table 4.3. This table shows how many characters it takes to define a specific subset of the MEM RESTful services for each of the specification formats.

4.3.1 RAML Contracts

The format of RAML Specifications is introduced in Section 2.2.6. Specifications can be written with syntax highlighting and syntax checking in the API Designer that is available in the Anypoint Platform. There is also support for writing the specification in the Sublime text editor[2] with the RAML plugin, even though it will not perform any complete syntax checking. RAML is written in YAML which is designed as a human-friendly format [25].

The project raml-tester[3] can be used to verify that the specification is actually matching the source code. The author was unable to make this work properly in MEM. However, it should be noted that there at least are possibilities to perform those types of tests in Java.

4.3.2 API Blueprint Contracts

The format of API Blueprint Specifications is introduced in Section 2.2.6. An online editor with syntax checking is available in Apiary, similar to the Anypoint Platform for RAML. There is also a Sublime plugin but to get it or any other offline validation working an installation of Microsoft Visual Studio is needed. API Blueprint is based on Markdown which is a human-friendly format that is designed with conversion to HTML in mind [9].

There is no support for testing the specification against the code in a Java environment with API Blueprint. This is natural since Java is not one of the three languages that API Blueprint has tooling support for.

[2] http://www.sublimetext.com/
[3] https://github.com/nidi3/raml-tester

4.3.3 WADL Contracts

The format for WADL-files is defined in Section 2.2.6. It is complicated to validate a WADL-file. There is no editor that can perform syntax checking. The only validation that is possible to perform is checking whether the file is valid XML. A possible workaround would be to check whether the WADL-file can be imported in a tool like e.g. SoapUI without errors. WADL is based on XML which is not considered to be a human-friendly format [5].

There is currently no tool for automatically testing that the Java code is following the WADL-specification.

4.4 GUI

4.4.1 Swagger UI

Swagger UI is an open source project that is currently hosted on GitHub[4]. There is some need for modification in order for the UI to have an acceptable functionality in MEM. The pre-built UI always initially points the user to the standard Swagger sample API "Petstore". Even though the user can navigate to the MEM REST services via the input box at the top of the UI it is still confusing. However, this behavior can be modified easily in the index.html file in the UI. It is also possible to remove the URL input box at the top completely in order to not confuse the user, which is easily done by removing the form tag in the same HTML file, see the final result in Figure 4.1. Swagger UI supports Basic and OAuth 2.0 authentication schemes if the service is secured but this will also require slight modifications of the GUI since the authentication scheme is not part of the Swagger specification.

4.4.2 Anypoint Platform

The standard way of exposing RAML definitions graphically is to use MuleSofts Anypoint Platform[5]. Unlike the Swagger UI, this is not a GUI that must be hosted by the user, instead the platform is provided by MuleSoft and an account will need to be created in order to start using it. Authentication schemes can be defined directly in a RAML specification and then selected in a dropdown in the GUI.

4.4.3 Apiary

Apiary[6] is a commercial product that can display APIs defined with API Blueprint. It is a similar solution to the Anypoint Platform. The user logs in to the Apiary service in order to edit and publish its APIs.

[4]https://github.com/swagger-api/swagger-ui
[5]https://anypoint.mulesoft.com/apiplatform
[6]http://apiary.io

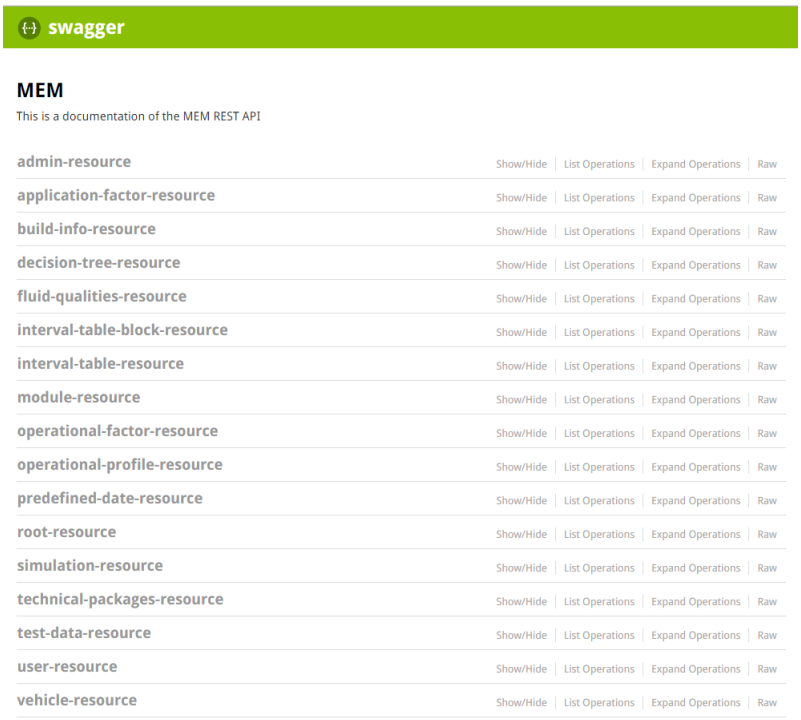

Figure 4.1. Swagger UI displaying the REST services in MEM.

4.4.4 WADL Stylesheets

There exists a project that can expose a graphical representation of WADL-files called WADL Stylesheets[7]. It uses XSLTs (eXtensible Stylesheet Language Transformation) to render the graphical view. This can be rather limiting when testing since modern browsers like Google Chrome disables CORS access to XSLT-files. Include the XSLT at the top of the WADL-file in order to make it work, see Code 4.2.

Code 4.2. XSL-file inclusion to make WADL Stylesheets work.

```
<?xml-stylesheet type="text/xsl" href="wadl.xsl"?>
```

4.5 Validation

Defining validation parameters for URLs is supported in all the specification formats investigated in this report. All valid URLs are those that can be found in the specification. Path and query parameters can make these URLs variable but these parameters can be constrained in all formats.

[7]https://github.com/ipcsystems/wadl-stylesheet

This section will investigate solutions from a more general perspective rather than from the specific MEM perspective. This is since, as explained in Section 3.3, MEM is currently not exposing its RESTful services externally and hence there has been no need to define any schemas yet.

4.5.1 Validating by Referencing Schemas

To achieve loose coupling between the specification format and validation of request bodys there needs to be possibilities for referencing external schemas. For XML requests XML-schemas is the natural choice, for JSON requests there are several options, where JSON-schema is one of those. RAML and API Blueprint support references to these types of schemas but Swagger does not. WADL has support for referencing external XML-schemas but does not support JSON-schemas.

4.5.2 Validating in Swagger

Swagger does not support schema references but validation parameters for response and request bodies can be defined in the Swagger specification. However, this limits the validation parameters to those that Swagger support and thus decreases flexibility and re-usability.

4.6 Testing

4.6.1 Testing via GUI

All GUI formats presented in Section 4.4 except WADL Stylesheets gives the user the ability to try out the different operations on the resources that are listed.

4.7 HATEOAS

In this section results from investigating and partially implementing HATEOAS in MEM is presented. Links are provided from the service in the response body or in the header.

4.7.1 Spring HATEOAS

HATEOAS was tested in MEM with the help of the Spring HATEOAS project[8]. Spring HATEOAS provides link builders that simplifies the process of linking resources. Fortunately, the way it works fits right into the design pattern in MEM where all core objects are transformed into REST resource objects before they are passed out to the consumer. This is in general a desirable design pattern since it enables information hiding. However, the Spring HATEOAS project has a different philosophy about how the REST resource objects should be designed. In MEM these

[8] http://projects.spring.io/spring-hateoas

are immutable and all their fields are set in the constructor. Spring HATEOAS does not support instantiation of such objects and thus forces introduction of mutability (setters). There is also an unfortunate method naming in the Spring HATEOAS ResourceSupport class. A method named "getId" returns its self referring link but such a getter is also highly likely to be present in a REST resource class, thus possibly causing conflicts upon extension, which was indeed the case in MEM. The introduction of Spring HATEOAS also breaks tests in MEM. The transformation classes, that translate core to REST objects, have not been mocked in test cases which they now need to be since the link builder will attempt to fetch information about request contexts which are not initialized when methods in RestControllers are called directly in the test cases. See Code 4.3 for an example of how Spring HATEOAS can be used to add links to resources.

Code 4.3. Extending the resource class with ResourceSupport and then extending the util that transforms the core objects into REST objects with ResourceAssemblerSupport. By calling the createResourceWithId method a self referring link is added but it is also possible to link other resources by calling the add method defined in the ResourceSupport class.

```
public class FluidQualityRest extends ResourceSupport {

   ...

}
```

```
public class FluidQualityUtils extends ResourceAssemblerSupport<CoreFluidQuality, FluidQualityRest> {

    public FluidQualityUtils() {
        super(FluidQualitiesResource.class, FluidQualityRest.class);
    }

    public FluidQualityRest toResource(@NotNull final CoreFluidQuality fq) {
        FluidQualityRest resource = createResourceWithId(fq.getId(), fq);
        ...
        return resource;
    }

    ...

}
```

Spring HATEOAS can generate links on the Atom and HAL format. Experimenting with the latest version of Spring HATEOAS showed that it produces links on HAL format for JSON responses by default if Spring Plugin Core[9] is added as a dependency. If it is not added Spring HATEOAS will fall back on the Atom format instead. So unfortunately this choice will be made silently dependent on whether Spring Plugin is included as a dependency in the build or not. Atom is always used for XML responses.

[9]https://github.com/spring-projects/spring-plugin

4.7.2 Observations from Testing of the Design

The Base URL

A central concept when talking about HATEOAS is the base URL. A starting point for all interactions with a RESTful service that is to supposed to return links to all resources that are relevant to start interacting with. MEM had no such URL. In fact, what could be assumed to be the base URL, normally it is assumed that the base is under the the empty URL "/", will direct the consumer to a JSP (JavaServer Pages) page. In order to show the concept of a base URL the author added a new resource "/base" to act as the actual base URL.

Using HTTP OPTIONS

It is relevant for the consumer that follows a link to know what HTTP operations that can be performed on it. As mentioned in Section 2.2.3, HTTP has a verb specifically designed for this purpose, OPTIONS. Currently, MEM returns all HTTP verbs in the "Allow" header when an OPTIONS operation is performed on a resource but this behavior was possible to change by modifying the return value of the options method in the RestController. The alternative to using OPTIONS is to define one link for each operation and e.g. set the rel attribute to the HTTP operation that is possible to perform on it. This however breaks the RESTfulness of the service in the sense that a resource is no longer defined by one unique URL but rather the different types of interactions with a resource are defined by unique URLs. This design misses the point and strength of using the HTTP verbs to define the type of interaction with a resource that is desired. The OPTIONS approach does require all resources to allow HTTP OPTIONS requests otherwise the design will be broken.

The Points of No Return

A commonly encountered issue when testing the added links to the MEM REST service responses were points of no return. One could view a RESTful service as a tree structure where the consumer traverses down along branches in order to reach a leaf (a single resource) to interact with. Using this analogy, for a designer it is easy to see how a parent node should link to its children but less trivial to remember to link back to parents in child resources. The tree structure is not a fitting analogy. It is a natural one since URLs are pointing to directory trees on a server but it is not a useful one in the context of HATEOAS. A better way to view a RESTful service is as a state machine. Each operation on a resource is a transition from one state to another, or possibly to the same state again. This is elegantly explained in a 2008 article on InfoQ by Webber et. al. [24]. With this mindset the points of no return could have been avoided.

Often times the design breaks upon delete requests. The standard response to a delete request is an empty body with the status code 204 (No Content). This leaves the consumer with no links to interact with at all and is therefore unacceptable

from a HATEOAS standpoint. It could be solved by linking to another resource in the location header but this field is usually ignored if the response is something other than a redirect or 201 (Created). One could instead use the link header field, even though it is not commonly used, or send back a body containing links.

Another interesting scenario where the consumer can be left out of options is when an error occurs. Links must therefore be added in error responses as well [4]. This guidance is especially useful when the consumer hits an invalid URL. A 404 (Not Found) can then be returned with a body containing a link pointing to the base URL. In MEM this means adding a custom error page to override the JBoss standard error page and instead redirecting to a controller that can return links to the base URL, see Code 4.4.

Code 4.4. Modifying the web.xml so that it overrides the JBoss default error page. Note that it is now pointing to the same URL as a controller defined below. 404:s that occur under the endpoint "/rest" will now be handled there.

```
    <error-page>
        <error-code>404</error-code>
        <location>/rest/error/404.json</location>
    </error-page>
```

```
@RestController
@RequestMapping(value = "/error", produces = MediaType.APPLICATION_JSON_VALUE)
public class ErrorHandlingController {

    \@RequestMapping(value = "/404.json")
    public ResponseEntity handle404() {
        return new ResponseEntity(new NotFoundRest(), HttpStatus.NOT_FOUND);
    }

}
```

```
public class NotFoundRest {

    private final Link[] links = new Link[]{linkTo(BaseResource.class).withRel("base")};

    public Link[] getLinks() {
        return links;
    }
}
```

Auto-Discovery

The explorer application was able to auto-discover all linked resources in the MEM REST service. This was the case even with points of no return, given that all resources can eventually be retrieved by following the links from the base resource, since it effectively performs a DFS to extract all linked resources. It would be possible to program a similar application that performs specific operations for specific link relations.

Absolute URLs

Presenting links as guidance can be an issue if the service is exposed via some type of load balancer or bus. The service might be possible to reach with different endpoints depending on from where it is called. This introduces a problem where the called service needs to be aware of which endpoint the caller is using to consume it. One possible solution is to only present relative links but this problem could also be solved by letting the service know this by setting some header in the request or with a more complex approach in which all middleware parses responses and corrects the links.

4.7.3 HATEOAS as a Complement

Naturally, HATEOAS can co-exist with the specification formats mentioned in this report. For instance any of the GUIs presented in Section 4.4 can be linked to in the response at the base URL.

4.7.4 HATEOAS for GUIs, Load Balancing and Failover

The small HATEOAS explorer application shows the potential of HATEOAS when designing GUIs. Instead of hard-coding ULRs for fetching data from the service in the GUI-client links can be extracted from the service. Thus, making it easier to keep URLs up to date, especially when GUI clients are developed externally and the developers have no insight into when the service infrastructure changes.

HATEOAS does not necessarily mean linking to resources that are hosted on the same origin. The links can point to other services as well. This would require a custom built solution for providing the links since Spring HATEOAS makes the assumption that all linked resources are served from the same origin. This could be utilized in a microservice environment where small services run together in a distributed system. A new version of a microservice can be deployed alongside an old version and then gradually integrated by letting a part of the other instances change their link to point to the new version instead of the old. Thus preventing an expensive complete rollback if the new version is not functioning properly.

Load balancing can easily be achieved by letting the services link to different machines depending on load. Typically this would be most useful in a cloud-based environment in which starting up and closing down server instances can be done fast, cheap and with minor effort.

4.8 Outcome at Scania IT

At the time that this report is written the internal RESTful services in MEM are documented with the help of Swagger as a result of this case study.

As a de facto Scania IT standard both Swagger and RAML are discussed as potential candidates. The choice of these two formats was partly a result of this

comparison of the alternatives. This work was not finished by the time that this report was finalized in May 2015.

Chapter 5

Discussion

5.1 Reliability of the Results

It must be noted that the evaluation criteria used for evaluating the specification formats were extracted with the IT environment at Scania IT in mind. The results obtained from the implementations in MEM should also be seen as case specific.

5.2 Specification Format Maturities

The question is if the specification formats are mature enough to be used in an enterprise environment. There are strengths and weaknesses in all of the formats that are presented in this report. Whether the tools can be considered mature depends on how high standards they are evaluated with. The following subsections will discuss each of the evaluation criteria presented in Section 3.2.1 with regards to the results.

5.2.1 Complexity

Complexity is an aspect that is difficult to address in an objective manner with the support of just one case study. The specification formats claim support for a set of languages but without evaluating them one by one it is difficult to assess the complexity that they will introduce in general. It is clear that choosing a format that does not allow for auto-generation or tests that run against the actual code will introduce complexity. In the specific case study of this report Swagger was the only specification format that was possible to auto-generate. RAML is at least possible to test in a Java environment even though the author failed to do so in MEM. Swagger has the best language support overall and it is supporting both of the main development languages at Scania IT. Both API Blueprint and WADL has limited language support in comparison.

Maintaining specifications that are not auto-generated or possible to test against code so that they are always up-to-date will require more work and discipline.

How much will depend on the attitude towards the specification format. Having a contract-first mindset will be helpful in this case.

Not much code and configuration was necessary to set up Swagger auto-generation in the case study. Writing the RAML and API Blueprint specifications was straight forward with the tooling that is available. WADL did not have the same tooling support. Another advantage that those two formats have over WADL is that they are based on more human-friendly formats such as YAML and Markdown rather than XML.

5.2.2 GUI

All specification formats can be displayed graphically and thus meet the GUI requirement. The WADL GUI project does not support testing. Since no other quality aspects of the GUIs were part of the evaluation requirements all other formats can be considered as mature with respect to GUI. It should however be clear to from the results that the GUI solutions for the specification formats differentiate in a number of ways.

5.2.3 Testing

Apart from the GUI testing capabilities all formats are supported in SoapUI but only WADL is available in the free version. One might argue that this is not a particularly important aspect. However, having the whole structure of the service defined in a test tool gives the tester a better overview.

5.2.4 Validation

Swagger is not holding up to standards when it comes to validation. It is possible to specify validation parameters in the format but the lack of support for referencing external schemas is a clear issue since it introduces a hard coupling between the specification and the validation. WADL is also limited in this regard since it does not support references to JSON-based schemas. The rest of the formats have support for both XML- and JSON-based schemas and are thus meeting the validation requirement.

5.2.5 Future Proof

The results indicate that Swagger currently has the largest and most active developer community. If this continues to be the case it will have a great advantage in the future. How it will evolve will also depend on how the new owner SmartBear will handle the project. The inclusion of the format by Microsoft in its products is also increasing the likelihood that it will survive in the future.

RAML has a strong backing in its owner MuleSoft and it has a moderate popularity in the open source community.

API Blueprint does not have the same community popularity as Swagger and RAML and neither does WADL. WADL was the first real description format for RESTful services and is thus naturally still supported in some tools. Looking at the growth of the other formats and the lack of interest for WADL it seems like WADL does not have much of a future.

5.3 HATEOAS

Initially the angle that this report took was that HATEOAS was an alternative in contrast to the specification formats. However, as it has been showed in the results it can instead be considered to be a complement. They differ so much in nature, the formats being static and HATEOAS being dynamic, that they both can bring value even when co-existing.

The results show that HATEOAS is technically trivial. In essence it is about linking resources. However, there are deeper design questions hidden under this trivial surface. How is a link defined in a standardized way? How is a service designed so that a consumer will not end up in a state with no options? How should absolute URLs be treated when the same service is reached with different URLs depending on from where it is called? The result section shows that these problems can be solved with careful design with the vision of building a state machine in mind.

In Section 2.2.9 two other investigations in this area were presented. Both opted to use a more verbose RDF description in order to handle the hypermedia constraint. Using RDF requires more design and there is no tooling like Spring HATEOAS to simplify the process. The question is whether the RDF formats bring any additional value. The results show that it is possible to make an entire service automatically explorable with links only. However, the RDF-formats allow for descriptions of implications of each resource operation, thus clarifying for an autonomous client what will happen if that operation is performed. It is clear that these formats will create overhead both in terms of transmitted data and workload for developers that need to maintain these descriptors and writing clients that can parse them. Relying on such solutions introduces a hard coupling to the format in a way that just supplying links does not.

Not introducing hard coupling is one of the advantages of HATEOAS since no binding to any specific format or tool is necessary. However, the results show that there will still be need for standardization when it comes to defining links.

5.4 Conclusions

None of the specification formats meet all demands that were formulated for evaluation in the method section perfectly. Swagger is the format that meets all of the requirements well except for the validation. The results also indicate that RAML is a format that meet many requirements, such as descent tooling support, good

validation capabilities with schema references, testing and GUI. API Blueprint lacks tooling support and popularity and the same goes for the old WADL format which is also suffering from being an XML-based format. The conclusion is that there exists specification formats that are mature enough to be used in an enterprise environment. Those two formats being Swagger and RAML. If a choice has to be made between them it will come down to what aspects that are most important. Swagger is lacking when it comes to validation but the tooling for RAML supports less languages.

A HATEOAS design of a RESTful service can be beneficial in an enterprise environment as was explained in the results. It can be used for load balancing, failover and for facilitating the design of GUIs. All of which can be useful in an enterprise environment. The results show that designing a service with HATEOAS in mind needs thought and will thus add an extra workload for developers.

5.5 Future

What is expected in the end is that some format will become more or less a de facto standard for specifying RESTful services. This expectation stems from the trend that competing protocols and formats usually do evolve into one standard. The fact that a specification format is not necessary at all to build RESTful services, in a way that e.g. the WSDL-file is for SOAP, is a factor in why this has not happened yet and perhaps might never occur.

Since HATEOAS is a concept rather than a specific tool or format it is less critical with future tooling support. Even though it might be interesting to have a tool like Spring HATEOAS to save some time in the development process. The use cases for microservices, with load balancing and failover, in the cloud can lead to more interest in the future. This can make HATEOAS go from being a vague theoretical concept for most developers to a practical design approach.

Possible future research is to perform an even more comprehensive comparison of the specification formats. Such as testing them out in multiple heterogeneous systems and for a longer period of time. In this way it would be possible to draw conclusions about how much value they bring, and how much complexity they introduce, in the long run. It is also possible to further investigate the benefits of using HATEOAS in an enterprise environment and in particular contrasting it to how much extra workload in terms of development and maintenance that it introduces.

Bibliography

[1] Adamczyk, P et al. 2011. *REST: From Research to Practice*. Ch. 2 (REST and Web Services: In Theory and in Practice). Springer. ISBN: 978-1-4419-8303-9.

[2] Allamaraju, S. 2010. *RESTful Web Services Cookbook*. O'Reilly Media, Inc. ISBN: 9780596801687.

[3] *API Blueprint*. Format 1A Revision 7. Retrieved from: https://github.com/apiaryio/api-blueprint/blob/master/API%20Blueprint%20Specification.md

[4] Bloomberg, J. 2013. *The Agile Architecture Revolution: How Cloud Computing, REST-Based SOA, and Mobile Computing Are Changing Enterprise IT*. John Wiley & Sons. ISBN: 9781118409770.

[5] Bos, Bert. 2003. *What is a good standard? An essay on W3C's design principles.* Section Readability. Retrieved from: http://www.w3.org/People/Bos/DesignGuide/readability.html

[6] Daigneau, R. 2011. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Addison-Wesley Professional. ISBN-13: 978-0-321-54420-9.

[7] Fielding, R. Taylor, R. 2000. *Principled Design of the Modern Web Architecture*. ICSE '00 Proceedings of the 22nd international conference on Software engineering, pages 407-416.

[8] Fritz, J. 2015. *Introducing Azure API Apps.* .NET Web Development and Tools Blog. Retreived from: http://blogs.msdn.com/b/webdev/archive/2015/03/24/introducing-azure-api-apps.aspx

[9] Gruber, J. 2004. *Markdown.* Retrieved from: http://daringfireball.net/projects/markdown/

[10] Hadley, M. 2009. *Web Application Description Language*. W3C Member Submission. Retrieved from: http://www.w3.org/Submission/wadl/

[11] Kelly, M. 2013. *JSON Hypertext Application Language.* Internet Draft. Available at: https://tools.ietf.org/html/draft-kelly-json-hal-06

[12] Liskin, O. Singer, L. Schneider, K. 2011. *Teaching old services new tricks: adding HATEOAS support as an afterthought.* Proceeding: WS-REST '11 Proceedings of the Second International Workshop on RESTful Design, pages 3-10. ACM. ISBN 978-1-4503-0623-2

[13] Muehlen, M. Nickerson, J. Swenson, K. 2004. *Developing web services choreography standards - the case of REST vs. SOAP.* Decision Support Systems Vol. 40, Issue 1, Pages 9-29.

[14] Newman, Sam. 2015. *Building Microservices.* O'Reilly Media Inc. ISBN: 978-1-4919-5035-7.

[15] Nottingham, M. Sayre, R. 2005. *The Atom Syndication Format.* RFC 4287. Section 4.2.7. Available at: https://tools.ietf.org/html/rfc4287#section-4.2.7

[16] *RAML™Version 0.8: RESTful API Modeling Language.* Retrieved from: http://raml.org/spec.html

[17] Richardson, L. 2008. *The Maturity Heuristic.* Justice Will take us Millions of Intricate Moves. Act 3. Speach notes retreived from: http://www.crummy.com/writing/speaking/2008-QCon/act3.html.

[18] Richardson, L. Amundsen M. Ruby, S. 2013. *RESTful Web APIs.* O'Reilly Media, Inc. ISBN 978-1-4493-5806-8. Ch 9.

[19] Sporny, M. et al. 2014. *JSON-LD 1.0. A JSON based Serialization for Linked Data.* Available at: http://www.w3.org/TR/json-ld

[20] Steiner, T. Algermissen, J. 2011. *Fulfilling the Hypermedia Constraint Via HTTP OPTIONS, The HTTP Vocabulary In RDF, And Link Headers.* WS-REST '11 Proceedings of the Second International Workshop on RESTful Design, Pages 11-14. ISBN 978-1-4503-0623-2.

[21] *The Swagger Specification.* Retrieved from: https://github.com/swagger-api/swagger-spec

[22] Verborgh, R. 2011. *Description and Interaction of RESTful Services for Automatic Discovery and Execution.* FTRA. ISBN: 9788996509790.

[23] Verborgh, R. 2011. *Efficient runtime service discovery and consumption with hyperlinked RESTdesc.* NWeSP-11. Pages 373-379. ISBN: 978-1-4577-1125-1.

[24] Webber, J. Parastatidis, S. Robinson, I. *How to GET a Cup of Coffee.* Article retrieved from: http://www.infoq.com/articles/webber-rest-workflow

[25] *YAML 1.2* Available at: http://yaml.org/