Best Practices for the Design of RESTful Web Services

Article *in* Proceedings - International Conference on Software Engineering · November 2015

CITATION
1

READS
162

5 authors, including:

Pascal Giessler
Karlsruhe Institute of Technology
4 PUBLICATIONS **1** CITATION
SEE PROFILE

Michael Gebhart
iteratec GmbH
34 PUBLICATIONS **126** CITATIONS
SEE PROFILE

Roland Steinegger
Karlsruhe Institute of Technology
7 PUBLICATIONS **2** CITATIONS
SEE PROFILE

Best Practices for the Design of RESTful Web Services

Pascal Giessler
and Michael Gebhart

iteratec GmbH
Stuttgart, Germany
Email: pascal.giessler@iteratec.de,
Email: michael.gebhart@iteratec.de

Dmitrij Sarancin, Roland Steinegger,
and Sebastian Abeck

Cooperation & Management
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
Email: dmitrij.sarancin@student.kit.edu,
Email: roland.steinegger@kit.edu,
Email: sebastian.abeck@kit.edu

***Abstract*—The trend towards creating web services based on the REpresentational State Transfer (REST) is unbroken. Because of this, several best practices for designing RESTful web services have been created in research and practice to ensure a certain level of quality. But, these best practices are often described differently with the same meaning due to the nature of natural language. In addition, they are not collected and presented in a central place but rather distributed across several pages in the World Wide Web, which impedes their application even further. In this article, we identify, collect, and categorize several best practices for designing RESTful web services and illustrate their application on a real system to show their application. For illustration purpose, we apply the best practices on the CompetenceService, an assistance service of the SmartCampus system developed at the Karlsruhe Institute of Technology (KIT).**

Keywords–REST; RESTful; best practices; collection; catalog; design; quality; research and practice

I. INTRODUCTION

Over the years, more and more web services based on the architectural style REST were developed, which uses existing functionality from the application layer protocol Hypertext Transfer Protocol (HTTP) [1] [2]. This results in an increasing interest compared to traditional web services with Simple Object Access Protocol (SOAP), which can be shown in a Google Trend Analysis with the keywords *REST* and *SOAP* or in the increasing usage of REST- instead of SOAP-based web services [1]. Also big companies, such as Twitter or Amazon, are using REST-like interfaces for their services, which are shown in their Application Programming Interface (API) documentations.

Despite this trend, there are still no standards or guidelines about how to develop a RESTful web service. Instead of this, several best practices in research and practice have been developed and were published in a range of articles, magazines and pages in the World Wide Web (WWW). But, these best practices were often described differently with the same semantics due to the nature of natural language [3]. This results in several obscurities and misconceptions by applying these best practices.

To overcome these issues, we have collected, categorized and formalized several best practices in a way that they can be easily applied during the development of RESTful web services , as well as for analyzing existing RESTful web services. More precisely, we have defined eight different categories and found an amount of 23 best practices that will be described in this paper. These best practices provide guidelines for the design of RESTful web services to support certain quality goals such as the usability of the Web API. Furthermore, their usage also results in an increasing consistency of web services.

For illustration, we have used this set of best practices for the development of the CompetenceService as part of the SmartCampus system at the KIT. The SmartCampus is a system which provides functionality for students, guests and members of an university to support their daily life. Today, the SmartCampus already offers some services, such as the ParticipationService to support the decision-making process between students, professors and members of the KIT with a new approach called system-consenting [4]. The developed services at the SmartCampus are based on REST, so that they can be used for several different devices as a lightweight alternative to SOAP.

The current paper is structured as follows: In Section II, the architectural style REST will be described in detail to lay the foundation for this paper. Afterwards, existing papers and articles will be discussed in Section III to show the necessity of identification, collection, and categorization of existing best practices for RESTful web services. The CompetenceService is used to illustrate the best practices will be presented in Section IV. In Section V, the best practices for RESTful web services will be presented in detail so that they can be easily applied during the design phase of such web services. Finally, a summary of this paper and an outlook on further work will be given in Section VI.

II. FOUNDATION

REST is an architectural style, which was developed and first introduced by Fielding [5] in his dissertation. According to Garlan and Shaw [6], an architectural style can be described as follows: “an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined.” [6, p. 6].

 ISBN: 978-1-61208-438-1

For the design of REST, Fielding [5] has identified four key characteristics, which were important for the success of the current WWW [7]. To ensure these characteristics, the following constraints were derived from existing network architectural styles together with another constraint for the uniform interface [5]: 1) Client and Server, 2) Statelessness, 3) Layered Architecture, 4) Caching, 5) Code on Demand and 6) Uniform Interface. The latter one represents the Web API of RESTful web services and can be seen as an umbrella term, since it can be decomposed into four sub-constraints [7]: 6.1) Identification of resources, 6.2) Manipulation of resources through representations, 6.3) Self-descriptive messages and 6.4) Hypermedia.

If all of these constraints are fulfilled by a web service, it can be called RESTful. The only exception is "Code on Demand", since it is an optional constraint and has not to be implemented by a web service.

III. State of the Art

This section discusses different articles, magazines and approaches in the context of RESTful best practices, which respect the architectural style REST and its underlying concepts.

In Fielding [5], Fielding presents the structured approach for designing the architectural style REST, while it remains unclear how a REST-based web service can be developed in a systematic and comprehensible manner. Furthermore, there is also a lack of concrete examples of how hypermedia can be used as the engine of the application state, which can be one reason why REST is understood and implemented differently.

Mulloy [8] presents different design principles and best practices for Web APIs, while he puts the focus on "pragmatic REST". By "pragmatic REST" the author means that the usability of the resulting Web API is more important than any design principle or guideline. But, this decision can lead to neglecting the basic concepts behind REST such as hypermedia.

Jauker [9] summarizes ten best practices for a RESTful API, which represent, in essence, a subset of the described best practices by Mulloy [8] and a complement of new best practices. Comparable with [8], the main emphasis is placed on the usability of the web interface and not so much on the architectural style REST, which can lead to the previously mentioned issue.

Papapetrou [10] classifies best practices for RESTful APIs in three different categories. However, there is a lack of concrete examples of how to apply these best practices on a real system compared to the two previous articles.

In Vinoski [11], a checklist of best practices for developing RESTful web services is presented, while the author explicitly clarifies that REST is not the only answer in the area of distributed computing. He structures the best practices in four sections, which addressing different areas of a RESTful web service such as the representation of resources. Despite all of his explanations, the article lacks in concrete examples to reduce the ambiguousness.

Richardson et. al [7] cover in their book as a successor of [12], among other topics, the concepts behind REST and a procedure to develop a RESTful web service. Furthermore, they place a great value on hypermedia , as well as Hypermedia As The Engine Of Application State (HATEOAS), which is not taken into account by all of the prior articles. But, the focus of this work is the comprehensive understanding of REST rather than providing best practices for a concrete implementation to reduce the complexity of development decisions.

In Burke [13], Burke presents a technical guide of how to develop web services based on the Java API for RESTful Web Services (JAX-RS) specification. But, this work focuses on the implementation phase rather than the design phase of a web service, where the necessary development decisions have to be made.

IV. Scenario

The SmartCampus is a modern web application, which simplifies the daily life of students, guests, and members at the university. Today, it offers several services, such as the ParticipationService for decision-making [4], the SmartMeetings for discussions or the CampusGuide for navigation and orientation on the campus. By using non-client specific technologies, the services can be offered to a wide range of different client platforms, such as Android or iOS.

The CompetenceService is a new service as part of the SmartCampus to capture and semantically search competences in the area of information technology. For easier acquisition of knowledge information, the CompetenceService offers the import of competence and profile information from various social networks such as LinkedIn or Facebook. The resulting knowledge will be represented by an ontology, while the profile information will be saved in a relational database. SPARQL Protocol And RDF Query Language (SPARQL) is used as the query language for capturing and searching knowledge information in the ontology.

In Figure 1, the previously described CompetenceService is illustrated in the form of a component diagram. For implementation of the CompetenceService, the Java framework Spring was used.

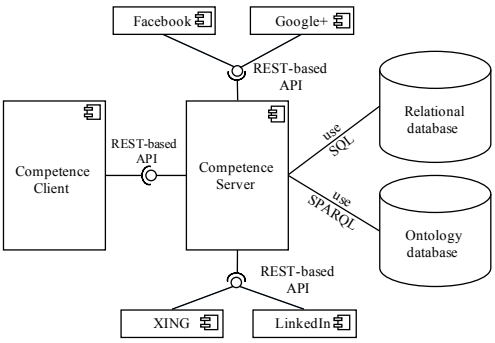

Figure 1. Component model of the CompetenceService.

To demonstrate the benefits of this service, a simple use case will be described in the following. A young startup company is looking for a new employee, who has competences in "AngularJS" and "Bootstrap". For that purpose, the startup company uses the semantics search engine of the CompetenceService to search for people with the desired skills. The resulting list of people will be ordered by relevance so that the startup company can easily contact the best match.

 ISBN: 978-1-61208-438-1

V. BEST PRACTICES FOR RESTFUL WEB SERVICES

This section presents eight different categories of best practices for designing RESTful web services, whereby each one is represented by a subsection. The contained best practices have to be seen as recommendations to design and improve such services rather than as strict guidelines. So, it is fine, if not all of the given best practices are fulfilled by a RESTful web service so long as an understandable reason for not considering one can be given. Furthermore, it is important to point out here that the fulfillment of the following best practices does not guarantee the compliance of the mentioned constraints in Section II. For this, the Richardson Maturity Modell (RMM) can be used to analyze the preconditions of a RESTful web service [14].

A. No Versioning

Versioning of a Web API is one of the most important considerations during the design of web services since the API represents the central access point of a web service and hides the service implementation. This is why a web interface should never be deployed without any versioning identifier according to Mulloy [8]. For versioning, many different approaches exist such as embedding it into the base Uniform Resource Identifier (URI) of the web service or using the HTTP-Header for selecting the appropriate version [8]. But, web services based on REST do not need to be versioned due to hypermedia.

That is why, RESTful web services can be compared with traditional websites that are still accessible on all web browsers when modifying the content of the websites. So, no additional adjustment is necessary on the client side. Furthermore, versioning also has a negative impact on deployed web services, which Fielding states as follows: “Versioning an interface is just a polite way to kill deployed applications” [15] since it increases the effort for maintaining the web service.

B. Description of resources

The description of resources correlates with the usability of the web service since the resources represent or abstract the underlying domain model. For this category, five best practices could be identified:

1) According to Vinoski [11], Papapetrou [10] and Mulloy [8], nouns should be used for resource names.
2) The name of a resource should be concrete and domain specific, so that the semantics can be inferred by a user without any additional knowledge [8] [10].
3) The amount of resources should be bounded to limit the complexity of the system, whereby this recommendation depends on the degree of abstraction of the underlying domain model [8].
4) The mixture of plural and singular by naming resources should be prevented to ensure consistency [8] [9].
5) The naming convention of JavaScript should be considered since the media type JavaScript Object Notation (JSON) is the most used data format for the client and server communication by this time [2] [8] [16].

Figure 2 illustrates the first, second and third best practice of this category.

```
1  /* ProfileController */
2  @RestController
3  @RequestMapping(value = "/profiles")
4  public class ProfilesController {
5    ...
6    @RequestMapping(method = RequestMethod.GET)
7    public List<Profile> getProfiles() {...}
8    ...
9  }
10
11 /* CompetenceController */
12 @RestController
13 @RequestMapping(value = "/competences")
14 public class CompetenceController {
15   ...
16   @RequestMapping(method = RequestMethod.GET)
17   public List<Competence> getCompetences() {...}
18   ...
19 }
```

Figure 2. Example for description of resources.

C. Identification of Resources

According to Fielding [5], URIs should be used for unique identification of resources. For this constraint, we have found four best practices:

1) An URI should be self-explanatory according to the affordance [8]. The term affordance refers to a design characteristic by which an object can be used without any guidance.
2) A resource should only be addressed by two URIs. The first URI address represents a set of states of the specific resource and the other one a specific state of the previously mentioned set of states [8].
3) The identifier of a specific state should be difficult to predict [10] and not references objects directly according to the Open Web Application Security Project (OWASP) [17], if there is no security layer available.
4) There should be no verbs within the URI since this implies a method-oriented approach such as SOAP [8] [9].

Figure 3 illustrates the second best practice of this category. Note that there are no verbs within the URIs, hence the fourth best practice is also fulfilled.

```
1 /* Set of profiles */
2 competence-service/profiles
3
4 /* Specific profile with identifier {id} */
5 competence-service/profiles/{id}
```

Figure 3. Example for identification of resources.

D. Error Handling

As already mentioned, the Web API represents the central access point of a RESTful web service, which is comparable with a provided interface of a software component [18]. Each information about the implementation of the service is hidden by the interface. Therefore, only the outer behavior can be

 ISBN: 978-1-61208-438-1

observed through responses by the web service, which is why well-known software debugging techniques such as setting exception breakpoints can not be applied.

For this reason, the corresponding error message has to be clear and understandable so that the cause of the error can be easily identified. With this in mind, we could identify three best practices:

1) The amount of used HTTP status codes should be limited to reduce the feasible effort for looking up in the specification [8] [9].
2) Specific HTTP status codes should be used according to the official HTTP specification [19] and the extension [20] [9] [11] [10].
3) A detailed error message should be given as a hint for the error cause on client side [8] [9]. That is why, an error message should consist of four ingredients: 3.1) a message for developers, which describes the cause of the error and possibly some hints how to solve the problem, 3.2) a message that can be shown to the user, 3.3) an application specific error code and 3.4) a hyperlink for further information about the problem.

Figure 4 illustrates the mentioned ingredients of an error message according to the third best practice of error handling.

```
1  HTTP/1.1 404 NOT FOUND
2  /* More header information */
3  {
4    "error" : {
5      "responseCode" : 404,
6          "errorCode" : 107,
7          "messages" : {
8            "developer" : "The resource 'profile'
                 could not be found.",
9            "user" : "An error occurred while
                 requesting the information. Please
                  contact our technical support."
10         },
11         "additionalInfo": ".../competence-
               service/errors/107"}
12 }
```

Figure 4. Example for detailed error message.

E. Documentation of the Web API

A documentation for Web APIs is a debatable topic in the context of RESTful web services since it represents an out-of-band information, which should be prevented according to Fielding: "Any effort spent describing what method to use on what URIs of interest should be entirely defined within the scope of the processing rules for a media type" [21]. This statement can be explained with the fact that documentation is often used as a reference book in traditional development scenarios. As a result of this, it can lead to hardcoded hyperlinks in the source code instead of interpreting hyperlinks of the current representation following the HATEOAS principle. Also business workflows will be often implemented according to the documentation. In this case, we call it Documentation As The Engine Of Application State (DATEOAS). As a result of this, we have developed a new kind of documentation in consideration of HATEOAS to give developers a guidance for developing a client component.

The new documentation consists of three ingredients: 1) Some examples which show how to interact with different systems according to the principle of HATEOAS due to the fact that some developers are not familiar with this concept [21], 2) an abstract resource model in form of a state diagram, which defines the relationship and the state transitions between resources. Also a semantics description of the resource and its attributes should be given in form of a profile such as Application-Level Profile Semantics (ALPS) [22], which can be interpreted by machines and humans and 3) a reference book of all error codes should be provided so that developers can get more information about an error that has occurred.

Figure 5 illustrates an abstract resource model of the CompetenceService. Based on this model, it can be derived which request must be executed to get the desired information. For example to get all competences of a specific profile, we have to first request the resource *profiles*. This results in a set of available profiles, whereby each profile contains one hyperlink for further information. After following the hyperlink by selecting the desired profile, the whole information about the profile will be provided , as well as further hyperlinks to related resources such as *competences*.

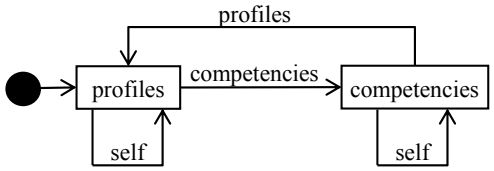

Figure 5. Example for documentation of the Web API.

F. Usage of Parameters

Each URI of a resource can be extended with parameters to forward optional information to the service. In the following, we are focusing on four different use cases since they will be supported by several web services offered by Facebook or Twitter.

1) Filtering: For information filtering of a resource either its attributes or a special query language can be used. The election for one of these two variants depends on the necessary expression power of the information filtering. Figure 6 illustrates how a special user group can be fetched by using a query language [9].

```
1 GET /profiles?filter=(competencies=java%20and%20
      certificates=MCSE_Solutions_Expert)
```

Figure 6. Filtering information by a using query language.

2) Sorting: For information sorting, Jauker [9] recommends a comma separated list of attributes with "sort" as the URI parameter followed by a plus sign as a prefix for an ascending order or a minus sign for a descending order. Finally, the order of the attributes represents the sort sequence. Figure 7 illustrates how information can be sorted by using the attributes *education* and *experience*.

 ISBN: 978-1-61208-438-1

```
1 GET /profiles?sort=-education,+experience
```

Figure 7. Sorting a resource by using attributes.

3) Selection: The selection of information in form of attributes reduces the transmission size over the network by responding only with the requested information. For this purpose, Mulloy [8] and Jauker [9] recommend a comma separated list of attributes and the term *fields* as the URI parameter. Figure 8 represents an example how the desired information can be selected before transmitting over the network.

```
1 GET /profiles?fields=id,name,experience
```

Figure 8. A selection of resource information.

4) Pagination: Pagination enables the splitting of information on several virtual pages, while references for the next (*next*) and previous page (*prev*) exist , as well as for the first and last page (*first* and *last*). As URI parameter, *offset* and *limit* were recommended, whereby the first one identifies the virtual page and the last one defines the amount of information on the virtual page [8] [9]. A default value for *offset* and *limit* can not be given since it depends on the information to be transmitted to the client, which Mulloy stated [8] as follows: "If your resources are large, probably want to limit it to fewer than 10; if resources are small, it can make sense to choose a larger limit" [8, p. 12]. Figure 1 illustrates a request using pagination on the resource *profiles*.

```
1 GET /profiles?offset=0&limit=10
```

Listing 1. Requesting 10 profiles by using pagination.

G. Interaction with Resources

By using REST as the underlying architectural style of a system, a client interacts with the representations of a resource instead of using it directly. The interaction between client and server is built on the application layer protocol HTTP, which already provides some functionality for the communication. For the interaction with a resource, we could identify three different best practices:

1) According to Jauker [9] and Mulloy [8], the used HTTP methods should be conform to the method's semantics defined in the official HTTP specification. So, the HTTP-GET method should only be used by idempotent operations without any side effects. For a better overview, Table I sums up the most frequently used HTTP methods and their characteristics. These characteristics can be used to associate the HTTP methods with the correct Create Read Update Delete (CRUD)-operation [11].
2) The support of HTTP-OPTIONS is recommended if a large amount of data has to be transmitted since it allows a client to request the supported methods of the current representation before transmitting information over the shared medium. But, this additional HTTP-OPTIONS request is only necessary, if the supported operations were not written explicitly in the representation.
3) The support of conditional GET should be considered during the development of a service based on HTTP since it prevents the server from transmitting previously sent information. Only if there are modifications of the requested information since the last request, the server responds with the latest representation. For the implementation of conditional GET, there are two different approaches that are already described by Vinoski [11].

TABLE I. CHARACTERISTICS OF THE MOST USED HTTP METHODS.

HTTP method	safe	idempotent
POST	No	No
GET	Yes	Yes
PUT	No	Yes
DELETE	No	Yes

H. Support of MIME Types

Multipurpose Internet Mail Extensions (MIME) types are used for the identification of data formats, which will be registered and published by the Internet Assigned Numbers Authority (IANA). These types can be seen as representation formats of a resource. For this category, we could identify the following four best practices:

1) At least two representation formats should be supported by the web service, such as JSON or Extensible Markup Language (XML) [8].
2) JSON should be the default representation format since its increasing distribution [8].
3) Existing MIME types should be used, which already support hypermedia such as JSON-LD (JSON for Linking Data), Collection+JSON and Siren [11].
4) Content negotiation should be offered by the web service, which allows the client to choose the representation format by using the HTTP header field "ACCEPT" in his request. Furthermore, there is the opportunity to weight the preference of the client with a quality parameter [11].

VI. SUMMARY AND OUTLOOK

In this article, we identified, collected, and categorized best practices for a quality-oriented design of RESTful web services. More precisely, based on existing work 23 best practices could be identified and classified into eight different categories. The intention of this article was not to reinvent the wheel. For this reason, the best practices of this article were reused from existing work. Focus of the work presented in this article was their collection, categorization, and thus unification. We illustrated the best practices by means of the CompetenceService developed at the KIT. By applying the best practices, the CompetenceService could be designed in a quality-oriented manner. Any time during the design or afterwards, the quality of the design could be systematically evaluated. The clear set of best practices enabled to

 ISBN: 978-1-61208-438-1

perform the evaluation by different developers. Furthermore, the repeatability of the analysis and the comparability of the results were guaranteed. As result, design weaknesses of the CompetenceService could be identified and rapidly corrected and the time spent making design decisions could be reduced.

The best practices and their categorization and unification help software architects and developers to design RESTful web services in a quality-oriented manner. As best practices are distributed across several existing work, until now, a systematic analysis of RESTful web services regarding their design quality has been a complex task. In most cases, software architects and developers have a basic understanding about how to create well-designed web services. However, a common understanding about how to evaluate web services is missing. The unification of best practices introduced in this article reduces the necessity to lookup best practices in literature.

In the future, we plan to investigate the impact of such best practices on the development speed. To evaluate the usefulness of the best practices for RESTful web services, we consider setting up two teams of students, Team A and Team B, with the requirement to develop two services as part of the SmartCampus at the KIT of similar complexity. Both teams are expected to have similar experiences in developing software systems and both teams should not have knowledge about the quality-oriented design of RESTful web services. However, Team A will be equipped with our catalog of best practices for RESTful web services. We expect that Team A will spend much less time searching appropriate design rules and design agreements. The best practices will provide Team A with guidelines about how to design the services. Furthermore, the design of the resulting service supports certain quality goals. However, we expect that the more sophisticated design will result in a more complex implementation phase. Figure 9 shows the expected results.

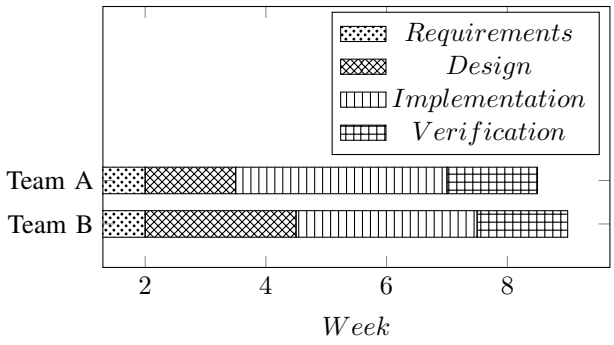

Figure 9. Duration of the development phases in weeks.

In addition, we plan to describe the best practices by means of technology-independent metrics. In a next step, we plan to map these technology-independent metrics onto concrete technologies, such as Java and JAX-RS. This mapping constitutes the basis for an automated application of the metrics on concrete design or implementation artifacts. We are currently working on a software tool, the QA82 Analyzer [23] [24]. This tool enables the automatic evaluation of software artifacts regarding best practices. This tool is available as open source to support the quality-oriented design of RESTful web services in practice, teaching, and research.

REFERENCES

[1] R. Mason, "How rest replaced soap on the web: What it means to you," October 2011, URL: http://www.infoq.com/articles/rest-soap [accessed: 2015-02-20].

[2] A. Newton, "Using json in ietf protocols," the IETF Journal, vol. 8, no. 2, October 2012, pp. 18 – 20.

[3] IEEE, "Std 830-1998: Recommended Practice for Software Requirements Specifications," 1998.

[4] M. Gebhart, P. Giessler, P. Burkhardt, and S. Abeck, "Quality-oriented requirements engineering for agile development of restful participation service," Ninth International Conference on Software Engineering Advances (ICSEA 2014), October 2014, pp. 69 – 74.

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

[6] D. Garlan and M. Shaw, "An introduction to software architecture," Pittsburgh, PA, USA, Tech. Rep., 1994.

[7] L. Richardson, M. Amundsen, and S. Ruby, RESTful Web APIs. O'Reilly Media, 2013.

[8] B. Mulloy, "Web API Design - Crafting Interfaces that Developers Love," March 2012, URL: http://pages.apigee.com/rs/apigee/images/api-design-ebook-2012-03.pdf [accessed: 2015-04-09].

[9] S. Jauker, "10 Best Practices for better RESTful API," Mai 2014, URL: http://blog.mwaysolutions.com/2014/06/05/10-best-practices-for-better-restful-api/ [accessed: 2015-02-19].

[10] P. Papapetrou, "Rest API Best(?) Practices Reloaded," URL: http://java.dzone.com/articles/rest-api-best-practices [accessed: 2015-02-26].

[11] S. Vinoski, "RESTful Web Services Development Checklist," Internet Computing, IEEE, vol. 12, no. 6, 2008, pp. 94–96. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4670126

[12] L. Richardson and S. Ruby, Restful Web Services. O'Reilly Media, 2007.

[13] B. Burke, RESTful Java with JAX-RS 2.0. O'Reilly Media, 2013.

[14] J. Webber, S. Parastatidis, and I. Robinson, REST in Practice: Hypermedia and Systems Architecture. O'Reilly Media, 2010.

[15] R. T. Fielding, "Evolve'13 - The Adobe CQ Community Technical Conference - Scrambled Eggs," 2013, URL: http://de.slideshare.net/royfielding/evolve13-keynote-scrambled-eggs [accessed: 2015-09-23].

[16] A. DuVander, "1 in 5 APIs Say "Bye XML"," 2011, URL: http://www.programmableweb.com/news/1-5-apis-say-bye-xml/2011/05/25 [accessed: 2015-02-20].

[17] OWASP, "Testing for insecure direct object references (otg-authz-004)," 2014, URL: https://www.owasp.org/index.php/Testing_for_Insecure_Direct_Object_References_(OTG-AUTHZ-004) [accessed: 2015-05-12].

[18] J. Bosch, Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach, ser. ACM Press Series. Addison-Wesley, 2000.

[19] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Rfc 2616, hypertext transfer protocol – http/1.1," http://tools.ietf.org/html/rfc2616, 1999.

[20] M. Nottingham and R. Fielding, "Rfc 6585, additional http status codes," 2012, URL: http://tools.ietf.org/html/rfc6585 [accessed: 2015-02-18].

[21] R. T. Fielding, "REST APIs must be hypertext-driven," October 2008, URL: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven [accessed: 2015-02-20].

[22] M. Amundsen, L. Richardson, and M. W. Foster, "Application-Level Profile Semantics (ALPS) ," Tech. Rep., August 2014, URL: http://alps.io/spec/ [accessed: 2015-04-09].

[23] M. Gebhart, "Query-based static analysis of web services in service-oriented architectures," International Journal on Advances in Software, 2014, pp. 136 – 147.

[24] QA82, "QA82 Analyzer," 2015, URL: http://www.qa82.org [accessed: 2015-02-27].

 ISBN: 978-1-61208-438-1