Which RESTful API Design Rules Are Important and How Do They Improve Software Quality? A Delphi Study with Industry Experts

Sebastian Kotstein[1](✉) and Justus Bogner[2]

[1] Herman Hollerith Zentrum, Reutlingen University, Böblingen, Germany
sebastian.kotstein@reutlingen-university.de

[2] Institute of Software Engineering, University of Stuttgart, Stuttgart, Germany
justus.bogner@iste.uni-stuttgart.de

Abstract. Several studies analyzed existing Web APIs against the constraints of REST to estimate the degree of REST compliance among state-of-the-art APIs. These studies revealed that only a small number of Web APIs are truly RESTful. Moreover, identified mismatches between theoretical REST concepts and practical implementations lead us to believe that practitioners perceive many rules and best practices aligned with these REST concepts differently in terms of their importance and impact on software quality. We therefore conducted a Delphi study in which we confronted eight Web API experts from industry with a catalog of 82 REST API design rules. For each rule, we let them rate its importance and software quality impact. As consensus, our experts rated 28 rules with high, 17 with medium, and 37 with low importance. Moreover, they perceived usability, maintainability, and compatibility as the most impacted quality attributes. The detailed analysis revealed that the experts saw rules for reaching Richardson maturity level 2 as critical, while reaching level 3 was less important. As the acquired consensus data may serve as valuable input for designing a tool-supported approach for the automatic quality evaluation of RESTful APIs, we briefly discuss requirements for such an approach and comment on the applicability of the most important rules.

Keywords: REST APIs · Design rules · Software quality · Delphi study

1 Introduction

Around 2000, Roy T. Fielding formalized the idea of "how a well-designed Web application behaves" by introducing a novel architectural style that he named "Representational State Transfer" (REST) [5]. In detail, he defined REST as a set of constraints that a Web application must fulfill to improve several quality attributes like scalability, efficiency, and reliability. The motivation for this was the concern that the Web as an open platform, i.e. not being under the control of a central entity, would not scale without rules and constraints to govern interactions and information exchanges [5,13].

Although REST mainly relies on well-established Web standards like Uniform Resource Identifiers (URIs) and is aligned with the features of the Hypertext Transfer Protocol (HTTP), REST and its constraints are not an official standard [23]. Moreover, Fielding's initial formulation of REST and its constraints describe the recommended behavior of a Web application from a high-level perspective instead of providing developers with detailed instructions on how to achieve this behavior. As a consequence, REST leaves room for various interpretations [21] and different design decisions when implementing Web applications [23]. With the popularity of RESTful services in industry [1,24] and the rapid growth of publicly available Web applications and APIs over the last two decades (see e.g. the number of newly registered APIs per year at *ProgrammableWeb*[1]), the variety of different "REST" designs were growing as well.

The popularity, diversity, and growing complexity of Web APIs make their quality evaluation and assurance very important. This is often manual work based on design rules or best practices, e.g. derived from the original REST constraints, and several works proposed guidance for developers when designing and implementing Web APIs [13,17,20]. At the same time, multiple studies analyzed the degree of REST compliance among existing real-world APIs by systematically comparing them against interface characteristics deduced from REST best practices and constraints [14,21,23]. Many of these studies concluded that only a small number of existing Web APIs is indeed *RESTful*, which means that the majority of Web APIs do not satisfy all required constraints. Nevertheless, many of the APIs classified as non REST-compliant claim to be RESTful [14], which suggests that the term "REST" may have become a synonym for every type of API that simply offers its functionality through URIs and the HTTP [21,26].

The large body of existing rules and best practices may also make it difficult for practitioners to select the most important ones. Simultaneously, the fact that only a small number of real-world APIs seems to adhere to rules and best practices for REST-compliant APIs leads us to believe that practitioners perceive many rules aligned with Fielding's REST constraints differently in terms of their importance and impact on software quality. We therefore see the need for an investigation from the perspective of industry practitioners. For that purpose, we conducted a Delphi study in which we confronted eight Web API experts from industry with a catalog of REST API design rules and asked them how they rate the importance and software quality impact of each rule. This study should answer the following research questions (RQs):

RQ1: Which design rules for RESTful APIs do practitioners perceive as important?

RQ2: How do practitioners perceive the impact of RESTful API design rules on software quality?

[1] https://www.programmableweb.com.

These results should provide valuable input for creating an approach for the automatic rule-based quality evaluation of real-world Web APIs. More precisely, tool support to automatically evaluate the software quality of APIs based on their conformance to important rules would assist practitioners in designing better RESTful APIs. Based on the results, we briefly discuss requirements for such an approach and comment on the applicability of the most important rules.

The remaining paper is organized as follows: as a basis for the selection of rules, we discuss the architectural style REST as well as existing literature about rules and best practices in Sect. 2. Furthermore, we present existing Web API analysis approaches, which might serve as a foundation for an automatic rule-based quality evaluation approach. Section 3 explains the details of our study design. In Sect. 4, we present the results of our study and discuss their interpretation and implications in Sect. 5. Finally, we describe threats to the validity of our study (Sect. 6) and close with a conclusion (Sect. 7).

2 Background and Related Work

In this section, we discuss the principles of the architectural style REST and the reasons why this style may lead to different interpretations and design decisions. To make a selection of rules and best practices for our study, we briefly discuss existing literature presenting rules and best practices for RESTful API design. Finally, we mention existing approaches for systematic Web API analysis that might serve as a foundation for automatic rule-based quality evaluation.

2.1 REST as an Architectural Style

In 2000, as part of his PhD dissertation [5], Fielding introduced REST as an architectural style that describes how distributed hypermedia systems like the World Wide Web (WWW) are designed and operated [26]. On the one hand, REST is a blueprint formalizing the application of existing Web techniques and standards like URIs and the HTTP [26]. On the other hand, it serves as a guideline for designing new applications such that they adopt these well-established Web concepts successfully and meet several quality attributes like scalability, efficiency, and reliability [4].

Fielding presented REST as a framework of architectural constraints that describe the proper use of these existing Web techniques and standards and, therefore, express several design decisions for building Web applications (e.g. Web APIs). In detail, he named six core constraints that a Web application must fulfill to be RESTful, namely *Client-Server*, *Layered System*, *Stateless Communication*, *Cache*, *Code-On-Demand*, and *Uniform Interface* [4]. Moreover, Fielding complemented the *Uniform Interface* by four additional constraints: *Identification of Resources*, *Manipulation of Resources Through Representations*, *Self-Descriptive Messages*, and *Hypermedia as the Engine of Application State* (HATEOAS) [4]. For detailed descriptions of these constraints, we refer to [4,6], and [13].

While some constraints like *Client-Server* and *Layered System* are automatically fulfilled because they are fundamental concepts of the WWW architecture, the concepts behind URIs and the HTTP promote other constraints like *Cache* and *Uniform Interface*, but cannot enforce their compliance [6]. In fact, a developer can ignore or misuse given mechanisms of URIs and the HTTP, which might result in a violation of these constraints. Especially in Web API design, adhering to *Uniform Interface* and its four additional constraints is crucial, since this enables developers to reuse existing client code for different APIs. Individual interface styles, however, force client-side developers to implement individual clients [14]. In our point of view, especially the *Uniform Interface* constraint and its four additional constraints leave developers too much room for different interpretations, thereby leading to various interface designs in practice.

2.2 Best Practices for REST

While Fielding's formal description of REST and its constraints mainly focus on the recommended behavior of a Web application and the naming of quality attributes that can be achieved by satisfying these constraints, several works examined the principles of REST. They translated them into rules and best practices to guide developers towards good RESTful design with instructions on how to satisfy them. The existing literature in this field includes scientific articles, e.g. by Pautasso [19], Petrillo et al. [20], and Palma et al. [16,17], as well as textbooks, e.g. by Richardson and Ruby [22], Massé [13], and Webber et al. [26]. While both [22] and [26] proposed several best practices embedded into continuous guides for designing and implementing REST APIs, Massé [13] presented a catalog of 82 rules, where each rules is described concisely and is often complemented by short examples and suggestions for its implementation. Petrillo et al. [20] compiled and presented a similar catalog, with rules inspired by the work of [13], but also incorporating rules and best practices from [22] and other sources. Unsurprisingly, most of the rules and best practices in the mentioned literature focus on *Uniform Interface* and its four additional constraints.

In addition to rules and best practices for REST API design, Leonard Richardson developed a maturity model [12] that allows one to estimate the degree of REST compliance of a Web API. More precisely, the model consists of four levels of maturity that incorporate the principles of REST:

– Web APIs complying with level 0 provide their functionality over a single endpoint (URI). Moreover, they use the HTTP solely as a transport protocol for tunneling requests through this endpoint by using `HTTP POST` and without leveraging any other Web mechanisms. SOAP and XML-RPC services typically rely on level 0.
– At level 1, Web APIs use the concept of resources, which means that they expose different endpoints (URIs) for different resources. However, at level 1, operations are still identified via URIs or specified in the request payload rather than using different HTTP methods for different operations.

– Web APIs at level 2 use HTTP mechanisms and semantics including different verbs for different operations as well as the interpretation of status codes. Level 2 is partially aligned with the *Uniform Interface* constraint of REST.
– Level 3 Web APIs embed hypermedia controls into responses to advertise semantic relations between resources and to guide a client through the application. This final level of the maturity model addresses the HATEOAS constraint of REST.

2.3 Studies for the Evaluation of Web APIs

Several publications analyzed the quality or conformity of existing RESTful APIs or proposed automatic approaches for their evaluation.

In [21], Renzel et al. analyzed the 20 most popular RESTful Web services listed in *ProgrammableWeb* in May 2011. For that purpose, they created a catalog of 17 metrics, which are based on the REST architecture style, relevant features of the HTTP, as well as characteristics and best practices of REST as described by [22]. This catalog contains metrics like the number of different HTTP methods used in an API or whether links are embedded into representations or not. For each of these 20 APIs, they manually read the API service description to capture the values of these metrics. The analysis revealed that nearly none of the 20 Web services were indeed RESTful (only four Web services were HATEOAS-compliant, for instance). Moreover, the study identified differences in terms of adaption and interpretation of REST principles among the analyzed Web services.

Palma et al. introduced an approach for the automatic detection of REST patterns and antipatterns called *SODA-R* [16]. They identified five patterns and eight antipatterns for REST and defined heuristics for detecting them. Palma et al. translated these heuristics into algorithms that automatically detect these (anti-)patterns on a set of APIs by invoking their interfaces. The tests against 12 real-world REST APIs revealed that developers tend to use customized headers negatively affecting the *Uniform Interface*, in particular, the *Self-Descriptive Messages* constraint. Moreover, eight out of 12 APIs did not embed hyperlinks into response payloads, thereby violating HATEOAS.

Rodríguez et al. [23] analyzed more than 78 GB of HTTP network traffic to gain insights into the use of best practices when designing Web APIs. Since the analyzed data contained any kind of Web-based traffic, they used API-specific heuristics to extract REST API related requests and responses and identified the relevant APIs based on this data. Then, they validated the extracted requests and responses against 18 heuristics, which they deduced from design principles and best practices introduced in [13,18,19]. Moreover, they mapped the heuristics to the levels of the Richardson maturity model [12] to estimate the level of compliance of the identified APIs. According to the paper, only a few APIs reached level 3 of the maturity model (embedded hyperlinks into resource representations), while the majority of APIs complied with level 2.

Petrillo et al. of [20] compiled a catalog of 73 best practices, which we already mentioned in Sect. 2.2. Based on this catalog, they manually analyzed the documentations of three cloud provider APIs and determined whether the respective cloud provider fulfilled these best practices or not. Although all three APIs fulfilled only between 56% and 66% of the best practices, Petrillo et al. assessed their maturity level as acceptable.

In [17], Palma et al. presented an approach for the detection of linguistic patterns and antipatterns addressing the URI structure of REST APIs and their compliance with best practices of REST. They tested the approach with 18 real-world APIs by invoking their endpoints and analyzing their documentations. Most of the analyzed APIs used appropriate resource names and did not use verbs within URI paths, which confirms the acceptance of this best practice. However, URI paths often did not convey hierarchical structures as it should be according to another best practice.

An extensive study for investigating the quality of real-world Web APIs and estimating the level of REST compliance was presented by Neumann et al. in [14]. The authors compared the documentations of 500 APIs claiming to be RESTful against 26 features reflecting REST principles and best practices deduced from [13,17,20,21,23] and other works. Similar like Renzel et al. [21], the authors identified a very heterogeneous adoption of REST. Especially the *Uniform Interface* constraint was barely adopted. The study came to the result that only four APIs (0.8%) fully complied with REST.

The aforementioned studies primarily focus on the analysis of Web API compliance with rules, features, and best practices aligned with the principles and constraints of REST. However, there are also two approaches by Haupt et al. [7] and Bogner et al. [2]. Rather than using best practices or rules, they analyzed Web APIs based on a set of interface characteristics and calculated metrics that provide insights about the general structure and quality of Web APIs. Both approaches run completely automatically based on machine-readable API documentation, which might be interesting for our rule-based quality evaluation approach. In detail, Haupt et al. [7] introduced a framework to provide insights about the structure of a REST API by automatically analyzing its service description. Bogner et al. [2] proposed a modular framework called *RESTful API Metric Analyzer (RAMA)* that calculates maintainability metrics from service descriptions and enables the automatic evaluation of REST APIs. Moreover, it converts the content of a service description into a canonical meta-model, which is used for the subsequent evaluation against a set of metrics.

By analyzing real-world Web APIs, the mentioned studies revealed that only a small number of Web APIs adheres to the rules and best practices of REST and, therefore, are truly RESTful. While these studies identified mismatches between theoretical concepts of REST and implementations by examining the work of developers, the reasons for these mismatches between theory and practice remain unclear. To the best of our knowledge, the opinions of developers about theoretical REST principles as well as rules and best practices for REST API design have not been investigated so far. We therefore see the need for a study from the perspective of industry practitioners: instead of analyzing their work as previous studies did, we directly want to confront them with rules and best practices for REST API design and query their perceived importance as well as the perceived impact on software quality. Furthermore, these results should enable rule-based quality evaluations of Web APIs relying not only on rules and best practices originating from theory but also incorporating opinions of practitioners to evaluate a Web API from a more practical point of view.

3 Research Design

In this section, we describe our followed methodology. We first present a general overview of our research process and then describe more design details of the most important phase, the conducted Delphi study with industry experts. For transparency and replicability, we make the most important study artifacts available online.[2]

3.1 General Process

This research took place in several distinct stages (see Fig. 1). In an initial step, we thoroughly analyzed existing rule catalogs and best practices for REST API design as well as approaches for systematic Web API analysis and related work in this area (see Sect. 2). Based on the results, we selected a comprehensive list of rules (namely the catalog from Massé [13]) and created a detailed study protocol with the necessary materials. We also recruited industry experts as participants and scheduled the study execution with them. The Delphi study took place over a period of roughly nine weeks between June and August 2020. As results, this study produced – for each rule – a consensus on its perceived importance (RQ1) as well as the quality attributes that experts perceived as positively impacted by adhering to it (RQ2). Finally, we analyzed the automation potential of this consensus data, i.e. we investigated the important rules from the perspective of using them in a tool-supported approach for the rule-based quality evaluation of RESTful APIs.

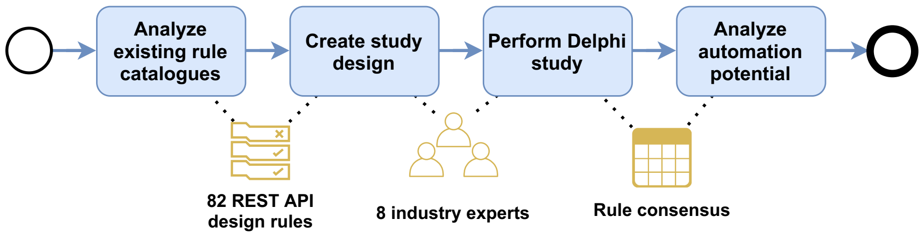

Fig. 1. Overview of the research process

[2] https://doi.org/10.5281/zenodo.4643906.

3.2 Delphi Study

While one-time interviews or a survey could have been used to gather relevant data for RQ1 and RQ2, we instead decided to employ the Delphi method [3], a structured communication approach originally designed as a forecasting technique by the RAND corporation in the 1950s. Within several iterations, individual expert opinions and rationales for the same set of questions are collected without experts getting in direct contact with each other, thereby avoiding issues with group dynamics. As input for the next round, participants get the anonymous results from their peers, often in aggregated form, and are encouraged to rethink their previous answers. Iterations are stopped once sufficient consensus has been reached. Dalkey and Helmer, two of the original authors, describe the advantages of this approach as follows [3]: "The method [...] appears to be more conducive to independent thought on the part of the experts and to aid them in the gradual formation of a considered opinion. Direct confrontation, on the other hand, all too often induces the hasty formulation of preconceived notions, an inclination to close one's mind to novel ideas, a tendency to defend a stand once taken, or, alternatively and sometimes alternately, a predisposition to be swayed by persuasively stated opinions of others." Delphi studies can be conducted with different objectives and several methodological variations exist. The method has been successfully used as a consensus technique in both information systems [15] and software engineering [11] research.

In our study, the objectives were to reach consensus on the importance of RESTful API design rules (RQ1) and their impact on software quality (RQ2). For each RQ, we conducted a separate Delphi process with the same experts.

Study Objects. The objects of this study were the 82 design rules for RESTful APIs compiled by Massé [13]. While several catalogs of design rules or best practices have been proposed in this area (see Sect. 2), Massé's collection comprises the vast majority of frequently mentioned rules in diverse areas. It is a comprehensive collection of industrial REST API design knowledge, which has also been cited over 440 times in scientific literature[3]. Massé organized the 82 rules nicely into five categories, namely *Interaction Design with HTTP* (29), *Identifier Design with URIs* (16), *Metadata Design* (15), *Representation Design* (13), and *Client Concerns* (9), and illustrated them with helpful examples and implementation suggestions. This pragmatic presentation makes the rules very actionable for practitioners, but also well suited for our Delphi study. Massé relied on RFC 2119[4] to incorporate a level of rule importance, in particular the three modifiers `may`, `should`, and `must`, optionally combined with `not`. For example, a rule from *Interaction Design with HTTP* prescribes that "22: `POST` must be used to create a new resource in a collection". Another example from *Representation Design* states that "73: A consistent form should be used to represent error responses".

[3] According to Google Scholar (https://scholar.google.com) in March, 2021.
[4] https://tools.ietf.org/html/rfc2119.

Study Participants. For our study participants, we set out the following requirements: (i) at least three years of experience with RESTful APIs, (ii) industry practitioners with a technical background, and (iii) from several different companies or at least divisions. This ensured that participants had sufficient practical experience and came from different backgrounds to avoid a single company-specific perspective. We used convenience sampling to recruit participants via our personal industry network and further referrals. While this may introduce some bias, it also ensured that we could rely on the participants' expertise and time commitment to the study. In the end, we were able to recruit eight participants (Table 1) with REST-related professional experience between six and 15 years (median: 11.5 years). They were from four different global IT enterprises, where they worked in seven different divisions. All participants were located in Germany.

Table 1. Delphi study participant demographics (PID: participant ID, CID: company ID, DID: company division ID)

PID	CID	Domain	# of employees	DID	REST exp. (years)
1	C1	Software & IT Services	100k–150k	D1	6
2				D2	15
3				D2	7
4				D3	13
5	C2	Software & IT Services	50k–100k	D4	6
6				D5	11
7	C3	Telecommunications	200k–250k	D6	15
8	C4	Technology & Manufact.	250k–300k	D7	12

Study Materials and Measurements. Several artifacts were used in this study. To get familiar with the rules and as a source for more details on specific rules, all participants received a digital copy of Massé's *REST API Design Rulebook*. Additionally, we extracted the rules and their description. We used this content to create Excel spreadsheets to gather participants' input. The spreadsheets also contained concrete instructions for participants and a form to gather their demographics. For RQ1, the spreadsheet contained a list of all 82 rules with a field to rate the perceived importance of each rule. Participants had to use a 3-point ordinal scale with the labels `low`, `medium`, and `high`. For RQ2, a similar spreadsheet was used, but this time with fields to associate a rule with positive impact on several different quality attributes (QAs). We used the eight top-level attributes from ISO 25010 [9], i.e. functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability. To ensure a common understanding, participants received a link to a website with descriptions about the standard[5]. For each rule, participants had to select which

[5] https://iso25000.com/index.php/en/iso-25000-standards/iso-25010.

QAs where positively impacted by adhering to it. The association was binary, i.e. a rule could either affect a specific QA (1) or not (0). Multi-selection was possible, i.e. a rule could affect several QAs. Both spreadsheets also contained tabs to provide the aggregated responses of the previous iteration for each rule, i.e. the number of experts who voted for the different levels of importance or the different associated QAs respectively.

Lastly, we created an analysis spreadsheet to summarize the responses and calculate *consensus* and *stability*. Since required percentage agreements of 70% and greater are common in Delphi studies [25], we defined 87.5% as consensus condition for a rule, i.e. only one of the eight experts could rate it differently than the rest. For the impact on software quality, agreement therefore meant that the exact same QAs had to be selected by seven experts. Additionally, we considered a stopping condition based on rating stability to avoid being deadlocked in iterations without significant change of votes [8]. As it is commonly used in Delphi studies, we chose Cohen's kappa coefficient (κ) to calculate the group stability between two iterations [10]. Since the ratings for rule importance were on an ordinal scale, we used a weighted kappa variant [8], i.e. the weight for changing your vote from `low` to `high` or vice versa was 2, while e.g. changing from `medium` to `high` only had a weight of 1. We used the standard variant for the impact on software quality. Landis and Koch [10] label κ-values between 0.81 and 1.00 as "almost perfect" agreement. While we did not define a strict stopping threshold and the stability has to be interpreted in conjunction with the percentage agreement, κ-values of 0.90 and greater for at least two iterations indicate that it may be advisable to stop the process.

Study Execution. Before starting the regular Delphi iterations, we scheduled personal meetings with participants to explain study procedure and provided materials. Furthermore, we addressed open questions and communicated the response timelines. We also encouraged participants to contact the study lead in case of uncertainties about the procedure or material during the iterations. After these meetings, the two consecutive Delphi processes were conducted (see Fig. 2).

The goal of the first Delphi process was to reach consensus on the perceived importance of the 82 REST API design rules (RQ1). After the first iteration, participants were in agreement about 42 of the 82 rules (51.22%). To start the second iteration, we provided them with the aggregated results from their peers, i.e. the number of times each level of importance had been chosen for a rule. Based on this feedback, participants were already able to reach full consensus on the 82 rules after two iterations, i.e. κ-based stopping was not necessary.

The goal of the second Delphi process was to reach consensus on the perceived positive rule impact on different software QAs (RQ2). To reduce the cognitive load and required effort, we only advanced the 45 rules for which the experts had rated the importance as `medium` or `high` to the final stage. Despite the substantially greater rating complexity per rule (choosing 0 or 1 for eight different QAs instead of three different importance levels), experts reached a similar

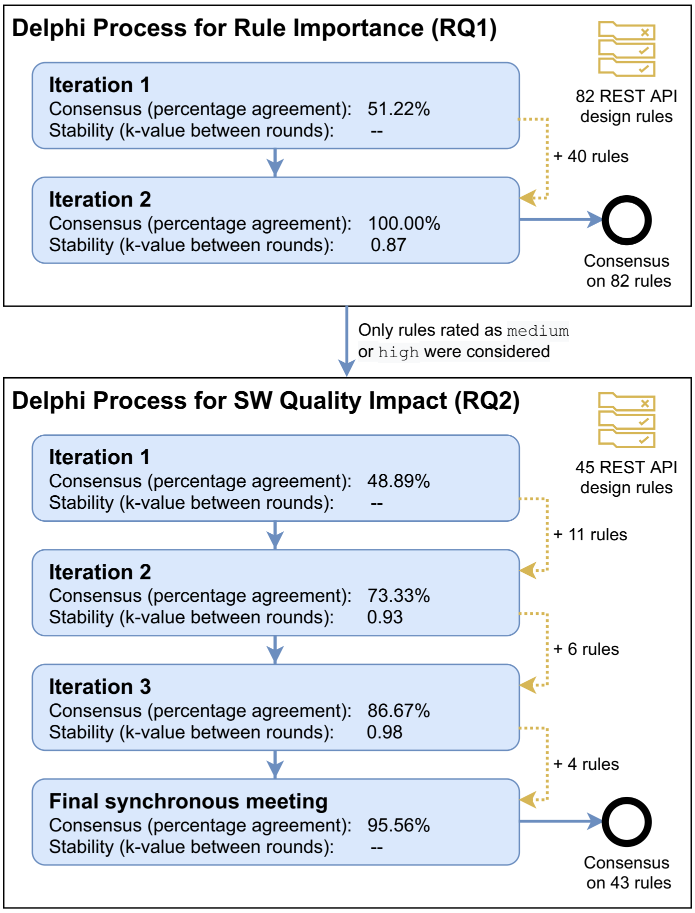
Delphi Process for Rule Importance (RQ1)
Iteration 1
Consensus (percentage agreement): 51.22%
Stability (k-value between rounds): --
Iteration 2
Consensus (percentage agreement): 100.00%
Stability (k-value between rounds): 0.87
82 REST API design rules
+ 40 rules
Consensus on 82 rules
Only rules rated as medium or high were considered
Delphi Process for SW Quality Impact (RQ2)
Iteration 1
Consensus (percentage agreement): 48.89%
Stability (k-value between rounds): --
Iteration 2
Consensus (percentage agreement): 73.33%
Stability (k-value between rounds): 0.93
Iteration 3
Consensus (percentage agreement): 86.67%
Stability (k-value between rounds): 0.98
Final synchronous meeting
Consensus (percentage agreement): 95.56%
Stability (k-value between rounds): --
45 REST API design rules
+ 11 rules
+ 6 rules
+ 4 rules
Consensus on 43 rules

Fig. 2. Overview of the Delphi study

percentage agreement in the first iteration, with consensus on 48.89% of the rules (22 of 45). Based on the provided peer responses, consensus on an additional 11 rules could be reached in iteration #2 (73.33% agreement with a κ-value of 0.93). However, progress slowed down in the third iteration, where agreement on only six more rules could be reached (86.67% agreement, $\kappa = 0.98$). Since κ-values had been greater than 0.90 for two consecutive rounds and only six rules without consensus were left, we decided to end the iterations and instead hold a final synchronous meeting to discuss the remaining rules. While this breaks participant anonymity and might reintroduce group dynamics bias, such meetings can lead to efficient consensus finalization. They are often suggested after the third or even later iterations. All participants agreed to this virtual meeting, which took place over one hour via Microsoft Teams. During the meeting, the study lead first introduced the remaining open topics and the current state of ratings for them. Afterwards, he initiated an open discussion. As a result, consensus could be reached for four of the six rules. The two rules without consensus are "54: Caching should be encouraged" and "61: JSON should be supported for resource representation". In both cases, five experts perceived a positive impact on functional suitability, while three experts could not be convinced by this.

4 Study Results

In this section, we present the aggregated results of the conducted Delphi study, i.e. how important our eight experts judged the rules and how they perceived the rule impact on software quality.

4.1 Perceived Importance of API Design Rules (RQ1)

Of the 82 REST API design rules studied in the first Delphi process, participants rated 37 with `low` importance (45%), 17 with `medium` importance (21%), and 28 rules with `high` importance (34%). In other words, our experts perceived nearly half of the rules as not really important and only one third as very important. Especially rules from the smaller categories *Client Concerns* and *Metadata Design* were rated as less relevant. Most important categories were *Identifier Design with URIs* (50% of rules rated as `high`) and *Interaction Design with HTTP* (45% of rules rated as `high`), which together also make up 55% of the total rules. These findings are summarized in Table 2. Table 3 lists all rules with `medium` importance and Table 4 the ones with `high` importance.

4.2 Perceived Rule Impact on Software Quality (RQ2)

During the second Delphi process, our experts rated the impact of the 45 rules with `medium` or `high` importance on the eight software QAs of ISO 25010. As a result, 10 rules were assigned to one QA, 12 rules to two QAs, 18 rules to three QAs, and the remaining five rules to four different QAs, i.e. a bit more than half of the rules were perceived with a positive impact on three or more QAs. Most associated attributes were *usability* (35), *maintainability* (35), and *compatibility* (26). In many cases, these QAs were impacted by the same rules. Of the 35 rules linked to *usability*, 21 were also selected for *maintainability* and *compatibility.* At the same time, *functional suitability* was only perceived as impacted by seven rules. The remaining attributes were very rarely selected, with *performance efficiency* being associated with three rules, *reliability* and *portability* with one rule each, and *security* with zero rules.

Table 2. Perceived rule importance by category (RQ1), percentages represent the fractions of total rules and fractions within the category respectively

Category	Total # of rules	Importance	# of rules
Identifier design with URIs	16 (20%)	High	8 (50%)
		Medium	5 (31%)
		Low	3 (19%)
Interaction design with HTTP	29 (35%)	High	13 (45%)
		Medium	5 (17%)
		Low	11 (38%)
Representation design	13 (16%)	High	4 (31%)
		Medium	2 (15%)
		Low	7 (54%)
Metadata design	15 (18%)	High	2 (13%)
		Medium	3 (20%)
		Low	10 (67%)
Client concerns	9 (11%)	High	1 (11%)
		Medium	2 (22%)
		Low	6 (67%)

Figure 3 summarizes these findings and also includes the associated rule categories. Our experts especially perceived rules from *Interaction Design with HTTP* with a broad impact on software quality: all 18 rules from this category were associated with three or more QAs, with only five rules from other categories having a similar spread. As the second largest category, *Identifier Design with URIs* (13) was primarily associated with *maintainability* (12) and *usability* (7). Furthermore, *Representation Design* (6) spread fairly evenly among the three most impacted QAs *usability* (6), *compatibility* (5), and *maintainability* (4). Lastly, the rules from *Metadata Design* (5) were the only ones not associated with *maintainability* at all.

5 Discussion

The interpretation of the study results leads to several findings with interesting potential implications for research and practice.

Concerning the importance of rules, we identified substantial differences between Massé's provided categorization and our participants' verdict. In total, Massé classified 10 of the 82 rules with the lowest level `may`, 50 with `should`, and 22 with `must`. Conversely, our participants perceived 45% of proposed rules (37) to be of `low` importance, among them seven classified as `must` and 24 as `should` by Massé. Additionally, they only selected `medium` for 17 rules, but `high` importance for 28 rules. Among the latter, Massé classified 15 with `should`

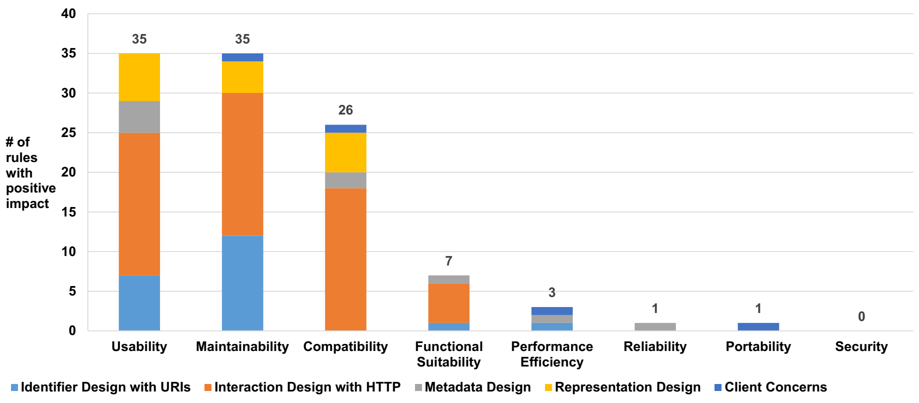

Fig. 3. Perceived rule impact on software quality (RQ2), stacked by rule category

Table 3. All 17 REST API design rules perceived with `medium` importance (RQ1)

ID	Category	Rule
2	URIs	A trailing forward slash (/) should not be included in URIs
6	URIs	File extensions should not be included in URIs
11	URIs	A plural noun should be used for store names
12	URIs	A verb or verb phrase should be used for controller names
15	URIs	The query component of a URI may be used to filter collections or stores
20	HTTP	`PUT` must be used to both insert and update a stored resource
26	HTTP	`200` ("OK") should be used to indicate nonspecific success
31	HTTP	`301` ("Moved Permanently") should be used to relocate resources
41	HTTP	`406` ("Not Acceptable") must be used when the requested media type cannot be served
43	HTTP	`412` ("Precondition Failed") should be used to support conditional operations
47	Meta	Content-Length should be used
51	Meta	Location must be used to specify the URI of a newly created resource
54	Meta	Caching should be encouraged
63	Repr.	XML and other formats may optionally be used for resource representation
69	Repr.	Minimize the number of advertised "entry point" API URIs
79	Client	The query component of a URI should be used to support partial response
82	Client	CORS should be supported to provide multi-origin read/write access from JavaScript

and even one with `may` ("13: Variable path segments may be substituted with identity-based values"). All in all, the verdict of our industry experts was more "extreme". Despite the intuitive expectation that such a consensus technique may lead to a more balanced compromise, participants perceived many more rules as not really important and a few more as very important in comparison

Table 4. All 28 REST API design rules perceived with `high` importance (RQ1)

ID	Category	Rule
1	URIs	Forward slash separator (/) must be used to indicate a hierarchical relationship
3	URIs	Hyphens (-) should be used to improve the readability of URIs
4	URIs	Underscores (_) should not be used in URI
5	URIs	Lowercase letters should be preferred in URI paths
9	URIs	A singular noun should be used for document names
10	URIs	A plural noun should be used for collection names
13	URIs	Variable path segments may be substituted with identity-based values
14	URIs	CRUD function names should not be used in URIs
17	HTTP	`GET` and `POST` must not be used to tunnel other request methods
18	HTTP	`GET` must be used to retrieve a representation of a resource
22	HTTP	`POST` must be used to create a new resource in a collection
23	HTTP	`POST` must be used to execute controllers
24	HTTP	`DELETE` must be used to remove a resource from its parent
27	HTTP	`200` ("OK") must not be used to communicate errors in the response body
28	HTTP	`201` ("Created") must be used to indicate successful resource creation
30	HTTP	`204` ("No Content") should be used when the response body is intentionally empty
34	HTTP	`304` ("Not Modified") should be used to preserve bandwidth
37	HTTP	`401` ("Unauthorized") must be used when there is a problem with the client's credentials
38	HTTP	`403` ("Forbidden") should be used to forbid access regardless of authorization state
44	HTTP	`415` ("Unsupported Media Type") must be used when the media type of a request's payload cannot be processed
45	HTTP	`500` ("Internal Server Error") should be used to indicate API malfunction
46	Meta	Content-Type must be used
57	Meta	Custom HTTP headers must not be used to change the behavior of HTTP methods
61	Repr.	JSON should be supported for resource representation
71	Repr.	A consistent form should be used to represent media type formats
72	Repr.	A consistent form should be used to represent media type schemas
73	Repr.	A consistent form should be used to represent error responses
74	Client	New URIs should be used to introduce new concepts

to Massé. This may suggest that practitioners prefer a small set of important rules instead of a large collection in which many rules are situational.

When analyzing rules related to the Richardson maturity model [12], participants rated the vast majority of rules necessary to reach *Level 2: HTTP Verbs* with `medium` or `high` importance. This includes the rules 17–25 about the correct usage of HTTP verbs (with the exception of `HEAD`, `OPTIONS`, and interestingly `PUT`), but also the rules 26–45 about correct status code usage. Conversely, nearly all of the rules associated with *Level 3: Hypermedia Controls* were categorized with `low` importance, including the *Representation Design* rules 65–68 and also the HATEOAS rule "70: Links should be used to advertise a resource's available actions in a state-sensitive manner". This indicates that our experts saw maturity level 2 as strongly desirable but also as sufficient, which is consistent with the results of Rodriguez et al. [23] and Neumann et al. [14]. Reaching level 3 may not be a priority for industry, even though it is one of the original RESTful constraints postulated by Fielding [5].

Concerning the rule impact on software quality, the vast majority of rules focus on usability (from an API user perspective), maintainability, or compatibility, with many rules influencing two or even all three of these QAs. Other QAs are less represented or even not at all. While it makes sense that service interface design may not be related to all QAs to the same extent, especially the absence of rules with an influence on performance efficiency and security is surprising. Intuitively, interface design should have a substantial influence on these two QAs, but the focus of Massé's rules seems to lie elsewhere. To evaluate these other QAs, different rule catalogs or approaches are therefore necessary.

Lastly, the greatest benefit should in theory be achieved by adhering to those rules that were both rated with `high` importance and associated with several QAs at once. In our sample, 17 rules with `high` importance impact at least three QAs, 13 of them from the category *Interaction Design with HTTP*, three from *Representation Design* (rules 71–73), and rule 57 from *Metadata Design*. The majority of them are again related to the correct usage of HTTP verbs and status codes, while rules 71–73 are concerned with the consistent structuring of media type formats, media type schemas, and error responses. This again highlights the fact that, in addition to consistency, our participants perceived reaching Richardson maturity level 2 as very important. These 17 rules are primary candidates for automatic evaluation approaches.

As part of our future work, we therefore plan to create tool support to evaluate Web APIs based on design rules. We start with the rules that practitioners assessed with `high` importance. This rule-based evaluation tool should assess the quality of a Web API by reporting detected rule violations and calculating individual scores for the QAs linked to these rules. Abstraction and modularity should guarantee that rules can easily be added, altered, or removed, e.g. according to new empirical data or based on company or team preferences. Reported rule violations should also be actionable for practitioners, e.g. by reporting the concrete location of findings and potential ways to address the violations. The SODA-R approach by Palma et al. [16] may serve as inspiration for our tool from both a methodological as well as a technical point of view. Similar as with SODA-R, for each rule, we will first identify REST properties and then define heuristics that indicate whether the analyzed API satisfies the respective rule. These properties can be either static and obtainable from REST API documentation (e.g. the set of URI templates indicating whether an API contains underscores (_), see rule 4) or dynamic and can be inferred from response data at runtime (e.g. a response status code). In future work, we will translate these heuristics into algorithms that automatically verify rule compliance of Web APIs. The authors of the SODA-R approach introduced heuristics for 13 REST (anti-)patterns [16] that partially overlap with some of our studied rules and, therefore, might be adaptable to our tool (e.g. heuristics for rule 26 and 46). Further heuristics applicable to the rules 4, 5, and 14 are defined by Rodríguez et al. in [23]. Moreover, in a second work by Palma et al. [17], further approaches for detecting violations of the rules 1, 4, 5, 9, 10, and 14 are presented that might be incorporated in our tool.

6 Threats to Validity

Several threats to the validity of this study have to be considered, which we discuss according to dimensions proposed by Wohlin et al. [27].

Construct validity describes the relation between the studied high-level concepts and the operationalized metrics to measure them, in other words, whether we actually measured what we wanted to measure. We studied the importance and software quality impact of design rules and measured how our Delphi participants perceived these concepts, i.e. we collected expert opinions. While experts often have solid experience-based judgment, they can also be wrong, even collectively. As long as our results are clearly treated as expert opinions, this is not an issue, especially since industry perception is important for research in this field. However, once we start to implement some of these rules, we will need more empirical data to judge their effectiveness, preferably non-opinion-based data.

Internal validity is concerned with the applied empirical rigor and potentially hidden factors that may have influenced the results. While we diligently designed our research protocol, adhered to it during the study, and relied on an established consensus technique, it is possible that confounding factors could have partially influenced the results. One such factor could be the misinterpretation of a rule or QA. To mitigate this, all participants received materials describing these concepts, and we encouraged them to ask clarifying questions. We perceive the risk of this threat as very small. A more substantial threat could be that participants knew the original rule importance level, as Massé encoded it directly into the rule text (`may`, `should`, `must`). This could have led to an anchoring effect, thereby biasing participants in their own verdict. However, as shown above, our participants were not afraid to adjust the initial importance level, leading to substantial differences between their and Massé's ratings. Since rewriting the rule texts could have introduced even greater inconsistencies, we still perceive this threat as fairly low. A last threat in this area could be the final synchronous meeting in our Delphi study. By breaking anonymity and facilitating synchronous exchange, we opened the door for potential influences related to group dynamics. Nonetheless, this was only for six of the 82 rules and only related to the QAs, not the importance. Moreover, no complete consensus was reached in this discussion, thereby reducing the probability that most participants yielded to the reputation or behavior of a few.

Lastly, *external validity* refers to the generalization potential of results, i.e. how well the results obtained in our specific setting carry over to different contexts. Only the 82 rules from Massé's catalog were considered in this study. While they provide a comprehensive collection of REST API design knowledge and substantially overlap with other catalogs, they are not the only available source and were published in 2011, i.e. roughly 10 years ago. Moreover, our software quality analysis revealed a focus on usability, maintainability, and compatibility. Rules with different focus and level of abstraction should be considered in future research. Additionally, only eight experts participated in our Delphi study. While they reached strong consensus and were from different companies and divisions, their general contexts were still fairly similar (large IT enterprises, enterprise application development, located in Germany). All in all, it is possible that experts with very different background would come to a different verdict, e.g. when considering the importance of HATEOAS-related rules.

7 Conclusion

In this paper, we presented the results of a Delphi study with the goal to analyze the perceived importance (RQ1) and positive software quality impact (RQ2) of REST API design rules from the perspective of industry practitioners. As study objects, we selected the catalog of 82 REST API design rules compiled by Massé. As participants, we recruited eight industry experts from four different IT companies with between six and 15 years of experiences with REST. As consensus results, our participants rated 37 rules with `low` importance (45%), 17 with `medium` importance (21%), and 28 rules with `high` importance (34%). From the 45 rules with `medium` or `high` importance, the quality attributes impacted the most were usability (35), maintainability (35), and compatibility (26). Other attributes from ISO 25010 were less impacted or not at all. Overall, the experts rated the rule importance differently than Massé, choosing the lowest level for many more rules and the highest level for a few more. They also perceived rules for reaching level 2 of the Richardson maturity model as very important, but not the ones for level 3. The acquired consensus data may serve as valuable input for designing a tool-supported approach for the automatic quality evaluation of RESTful APIs. Follow-up research should carefully analyze the automation potential of these rules, e.g. for static or dynamic analysis, as well as how to report and aggregate the findings. For finally creating such rule-based tool support, existing approaches may serve as a methodological or technical basis. To support such endeavors, we make our study artifacts available online (see Footnote 2).

Acknowledgments. We kindly thank Tobias Hallmayer for his assistance with study planning, execution, and analysis. This research was partially funded by the Ministry of Science of Baden-Württemberg, Germany, for the doctoral program *Services Computing* (https://www.services-computing.de/?lang=en).

References

1. Bogner, J., Fritzsch, J., Wagner, S., Zimmermann, A.: Microservices in industry: insights into technologies, characteristics, and software quality. In: 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), Hamburg, Germany, pp. 187–195. IEEE (2019)
2. Bogner, J., Wagner, S., Zimmermann, A.: Collecting service-based maintainability metrics from RESTful API descriptions: static analysis and threshold derivation. In: Muccini, H., et al. (eds.) ECSA 2020. CCIS, vol. 1269, pp. 215–227. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-59155-7_16
3. Dalkey, N., Helmer, O.: An experimental application of the DELPHI method to the use of experts. Manag. Sci. **9**(3), 458–467 (1963)
4. Fielding, R.T., Taylor, R.N.: Principled design of the modern web architecture. ACM Trans. Internet Technol. **2**(2), 115–150 (2002)
5. Fielding, R.T.: Architectural styles and the design of network-based software architectures. Ph.D. thesis, University of California, Irvine, Irvine, CA, USA (2000)
6. Haupt, F., Karastoyanova, D., Leymann, F., Schroth, B.: A model-driven approach for rest compliant services. In: Proceedings of the IEEE International Conference on Web Services (ICWS 2014), pp. 129–136. IEEE (2014)
7. Haupt, F., Leymann, F., Scherer, A., Vukojevic-Haupt, K.: A framework for the structural analysis of REST APIs. In: 2017 IEEE International Conference on Software Architecture (ICSA), Gothenburg, Sweden, pp. 55–58. IEEE (2017)
8. Holey, E.A., Feeley, J.L., Dixon, J., Whittaker, V.J.: An exploration of the use of simple statistics to measure consensus and stability in Delphi studies. BMC Med. Res. Methodol. **7**(1) (2007). Article number: 52. https://doi.org/10.1186/1471-2288-7-52
9. International Organization for Standardization: ISO/IEC 25010 - systems and software engineering - systems and software quality requirements and evaluation (SQuaRE) - system and software quality models (2011)
10. Landis, J.R., Koch, G.G.: The measurement of observer agreement for categorical data. Biometrics **33**(1), 159 (1977)
11. Lilja, K.K., Laakso, K., Palomki, J.: Using the Delphi method. In: PICMET: Proceedings of the Portland International Center for Management of Engineering and Technology, Portland, OR, USA. IEEE (2011)
12. Fowler, M.: Richardson Maturity Model (2010). https://martinfowler.com/articles/richardsonMaturityModel.html. Accessed 26 Mar 2021
13. Masse, M.: REST API Design Rulebook, 1st edn. O'Reilly Media Inc., Sebastopol (2011)
14. Neumann, A., Laranjeiro, N., Bernardino, J.: An analysis of public REST web service APIs. IEEE Trans. Serv. Comput. **PP**(c), 1 (2018)
15. Okoli, C., Pawlowski, S.D.: The Delphi method as a research tool: an example, design considerations and applications. Inf. Manag. **42**(1), 15–29 (2004)

16. Palma, F., Dubois, J., Moha, N., Guéhéneuc, Y.-G.: Detection of REST patterns and antipatterns: a heuristics-based approach. In: Franch, X., Ghose, A.K., Lewis, G.A., Bhiri, S. (eds.) ICSOC 2014. LNCS, vol. 8831, pp. 230–244. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-45391-9_16
17. Palma, F., Gonzalez-Huerta, J., Founi, M., Moha, N., Tremblay, G., Guéhéneuc, Y.G.: Semantic analysis of RESTful APIs for the detection of linguistic patterns and antipatterns. Int. J. Coop. Inf. Syst. **26**(02), 1742001 (2017)
18. Palma, F., Gonzalez-Huerta, J., Moha, N., Guéhéneuc, Y.-G., Tremblay, G.: Are RESTful APIs well-designed? Detection of their linguistic (anti)patterns. In: Barros, A., Grigori, D., Narendra, N.C., Dam, H.K. (eds.) ICSOC 2015. LNCS, vol. 9435, pp. 171–187. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48616-0_11
19. Pautasso, C.: RESTful web services: principles, patterns, emerging technologies. In: Bouguettaya, A., Sheng, Q., Daniel, F. (eds.) Web Services Foundations, pp. 31–51. Springer, New York (2014). https://doi.org/10.1007/978-1-4614-7518-7_2
20. Petrillo, F., Merle, P., Moha, N., Guéhéneuc, Y.-G.: Are REST APIs for cloud computing well-designed? An exploratory study. In: Sheng, Q.Z., Stroulia, E., Tata, S., Bhiri, S. (eds.) ICSOC 2016. LNCS, vol. 9936, pp. 157–170. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46295-0_10
21. Renzel, D., Schlebusch, P., Klamma, R.: Today's top "RESTful" services and why they are not RESTful. In: Wang, X.S., Cruz, I., Delis, A., Huang, G. (eds.) WISE 2012. LNCS, vol. 7651, pp. 354–367. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35063-4_26
22. Richardson, L., Ruby, S.: RESTful Web Services, 1st edn. O'Reilly Media Inc., Sebastopol (2007)
23. Rodríguez, C., et al.: REST APIs: a large-scale analysis of compliance with principles and best practices. In: Bozzon, A., Cudre-Maroux, P., Pautasso, C. (eds.) ICWE 2016. LNCS, vol. 9671, pp. 21–39. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-38791-8_2
24. Schermann, G., Cito, J., Leitner, P.: All the services large and micro: revisiting industrial practice in services computing. In: Norta, A., Gaaloul, W., Gangadharan, G.R., Dam, H.K. (eds.) ICSOC 2015. LNCS, vol. 9586, pp. 36–47. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-50539-7_4
25. Vernon, W.: The Delphi technique: a review. Int. J. Ther. Rehabil. **16**(2), 69–76 (2009)
26. Webber, J., Parastatidis, S., Robinson, I.: REST in Practice: Hypermedia and Systems Architecture, 1st edn. O'Reilly Media Inc., Sebastopol (2010)
27. Wohlin, C., Höst, M., Henningsson, K.: Empirical research methods in software engineering. In: Conradi, R., Wang, A.I. (eds.) Empirical Methods and Studies in Software Engineering. LNCS, vol. 2765, pp. 7–23. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45143-3_2