Checklist for the API Design of Web Services based on REST

Pascal Giessler, Roland Steinegger, and Sebastian Abeck

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

Michael Gebhart

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

***Abstract*—The trend towards creating web services based on the architectural style REpresentational State Transfer (REST) is unbroken. Several best practices for designing RESTful web services have been emerged in research and practice to ensure some degree of quality and share solutions to recurring challenges in the area of API design. But, these best practices are often described differently with the same meaning due to the nature of natural language. Also, they are not collected, categorized and presented in a central place but rather distributed across several pages in the World Wide Web, which impedes their application even further. Furthermore, it is often unclear which best practice has to be taken into account when designing a RESTful API for a particular scenario. In this article, we identify, collect, and categorize several best practices for designing APIs for RESTful web services and form a checklist. To support a prioritization of relevant best practices, we have mapped them on quality characteristics of the ISO/IEC 25010/2011. For illustration purpose, we apply the checklist on the CompetenceService as part of the SmartCampus ecosystem developed at the Karlsruhe Institute of Technology (KIT).**

Keywords–REST; RESTful; best practices; checklist; quality-driven; catalog; design; quality; api design; resource-orientation; SmartCampus

I. INTRODUCTION

This article is an extended version of [1]. It represents a collection of common best practices for designing Application Programming Interface (API)s for RESTful web services that have been derived from a range of articles, magazines, and pages on the World Wide Web (WWW). The motivation for this collection was because more and more web services based on the architectural style REST over Hypertext Transfer Protocol (HTTP) were developed and deployed compared to traditional web services with Simple Object Access Protocol (SOAP). This trend can also be seen at big companies, such as Twitter or Spotify, are using REST-like API for their services, which are shown in their API documentations [4] [5]. We are calling it REST-like at this point since we do not want to evaluate if this API considers all constraints of the uniform interface defined by Fielding [6] and can, therefore, be called RESTful. But, there is a lack of standards and guidelines on how to design an appropriate API, for example regarding the usability or the maintainability [2] [3].

Today, an own business model has been established around APIs when looking at the revenue of Salesforce or Expedia [7] [8]. For instance, "Salesforce.com generates 50% of its revenue through APIs," [7] according to the Harvard Business Review. That is why it is more important than ever that APIs have to be designed carefully especially when dealing with a large user base and heterogeneous platforms. For example, a change of an API should not break any consumer and, therefore, must be robust toward evolvability of the API. As discussed in [9], the API design and its strategy were also identified as a solution approach for the challenges of the digital transformation in software engineering.

To meet these challenges regarding the design of APIs, we have collected, categorized and formalized several best practices in the form of a checklist so it can be easily applied during the design of APIs for RESTful web services. To support a prioritization and selection of relevant best practices for a particular application scenario, we have mapped them on quality characteristics of the ISO 25010:2011 [10]. For instance, if it is important to support mobile platforms, then you should consider reducing the necessary requests to get the needed information.

For the purpose of illustration, we first show how we have integrated the checklist in our software development process to ensure an API design with quality in mind. Besides, we show exemplarily the applied best practices on the CompetenceService as part of the SmartCampus system at the KIT. The SmartCampus is a system that provides functionality for students, guests, and members of a university to support their daily life. Today, the SmartCampus is designed according to the trending microservice approach [11] and offers already 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 [12]. 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 article. Afterward, existing papers and articles will be discussed in Section III to show the necessity

of identification, collection, and categorization of existing best practices for the API design of RESTful web services. The CompetenceService is presented in Section IV of which the best practices will be exemplarily illustrated. Besides, our development process will be described to show how APIs will be designed and how the best practices are integrated. In Section V, the checklist containing the best practices will be explained in detail. Section VI focusses the mapping of quality characteristics of the ISO 25010:2011 [10] onto the previously introduced best practices by illustrating them on a concrete scenario. Finally, a summary of this article and an outlook on further work will be given in Section VII.

II. FOUNDATION

This section shall impart the necessary foundation for the scope of this article. First, the architectural style REST and its constraints will be described. Then, the term API will be described while a classification model for APIs based on REST will be introduced. In addition, informal characteristics of a API will be represented that can be found in literature.

A. REpresentational State Transfer (REST)

REST is an architectural style, which was developed and first introduced by Fielding [6] in his dissertation. According to Garlan and Shaw [13], 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." [13, p. 6].

For the design of REST, Fielding [6] has identified four key characteristics, which were important for the success of the current WWW [14]. To ensure these characteristics, the following constraints were derived from existing network architectural styles together with another constraint for the uniform interface [6]:

1) Client and Server: A client component sends a request to a server component for executing a remote operation. It is incumbent upon the server component to perform or reject the request [6].
2) Statelessness: Each request from client to server has to contain all necessary information to perform the request, which leads to the following advantage: "There is no need for the server to maintain an awareness of the client state beyond the current request" [6, p. 119].
3) Layered Architecture: A layered-client-server architecture enables the application of the Separation of Concern (SoC) principle and the opportunity to add features like load balancing or caching mechanisms to multiple layers [6] [15].
4) Caching: This constraint allows a client to match its request to a previous response from the server with the result that no request has to be transmitted over the network [14].
5) Code on Demand: With the usage of Code on Demand, additional programming logic can be requested from the server that is needed for processing received information from the server [14].
6) Uniform interface: The term "uniform interface" (hereinafter API) can be seen as an umbrella term, since it can be decomposed into four sub constraints [14]: 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 be implemented by a web service. The mentioned constraints are illustrated in Figure 1.

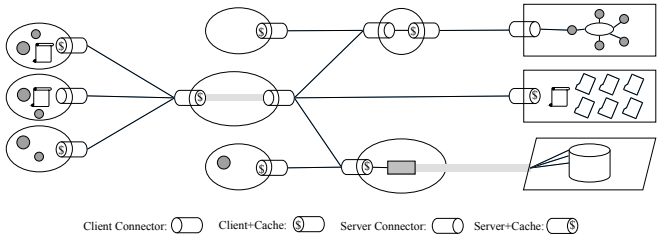

Figure 1. REST style [6, p. 84]

B. (Web) API

An Application Programming Interface (API) is a "local interface from higher-level component to a lower-level component" and can also be called as a horizontal interface [16, p. 915]. It acts as a contract between the service and the service consumer in the area of web services. It describes how the client can communicate with the service and how the request will be processed and responded to. An API can be called a web API, when the interface can be accessed via the internet or via internet-enabled technologies, such as HTTP.

For the classification of APIs based on REST, Richardson et. al. have developed a maturity model that classifies the API according their compliance of the mentioned preconditions for the uniform interface (see Section II-A 6.1 - 6.4) [17]. This so-called Richardson Maturity Modell (RMM) consists of three different maturity levels: 1) Usage of a resource-oriented styles rather than Remote Procedure Call (RPC) style, 2) Usage of HTTP methods according to their semantic and, 3) Usage of Hypermedia so that the API is self-documenting. For better illustration, the RMM is represented by Figure 2.

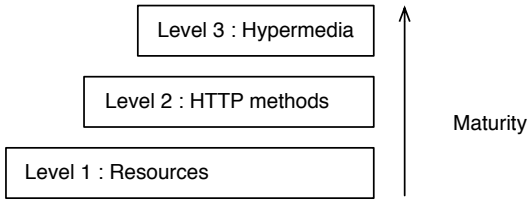

Figure 2. Richardson Maturity Model

Besides this classification, there are also informal criteria that can be found in literature especially when looking for integration technology. According to Newman [11], breaking changes should be avoided. This means that the API should be robust in terms of its evolution since it acts as a contract between service consumer and service provider. Also, the API should be technology-agnostic and hide internal implementation details not to increase the coupling. That is why the

API should abstract from a particular implementation by not exposing internal details, such as the used technology or any proprietary standard. Another important characteristic is the simplicity, and the comprehension from a service consumer perspective since other developers or development teams are the primary target group.

III. Background

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 [6], 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.

In [11], Newman presents a book about the microservice approach that is followed by several companies. Although, there is dedicated chapter regarding the ideal integration technology, it lacks on concrete best practices or guidelines especially for APIs based on REST. Instead of this, the book provides rather an overview about the technology choices that have to be made when choosing an integration technology.

Mulloy [18] 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 [19] summarizes ten best practices for a RESTful API, which represent, in essence, a subset of the described best practices by Mulloy [18] and a complement of new best practices. Comparable with [18], 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 [20] 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 [21], 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 [14] cover in their book as a successor of [22], 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. s In [23], 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.

In [24], Guinard and Trifa provide a guide on how to design and implement Internet of Things (IoT) solutions on the Web to ensure interoperability across different platforms. They recommend using a hypermedia approach when designing an API. But, common challenges, such as troubleshooting of an API, naming of resources and error handling are not discussed in detail.

IV. Scenario

This section addresses the domain at which the best practices for the API design will be illustrated. Besides, it shows the engineering principle behind the design and the development of the SmartCampus at the KIT to highlight the importance of APIs and the necessity of best practices.

A. Domain

The SmartCampus is a modern web application, which simplifies the daily life of students, guests, and members at the university (see Figure 3). Today, it offers several services, such as the ParticipationService for decision-making [12], the SmartMeetings for discussions or the CampusGuide for navigation and orientation on the campus, and the WorkspaceService to find a free working place by using smart devices. By using non-client specific technologies, the services can be offered to a wide range of different client platforms, such as Android or iOS.

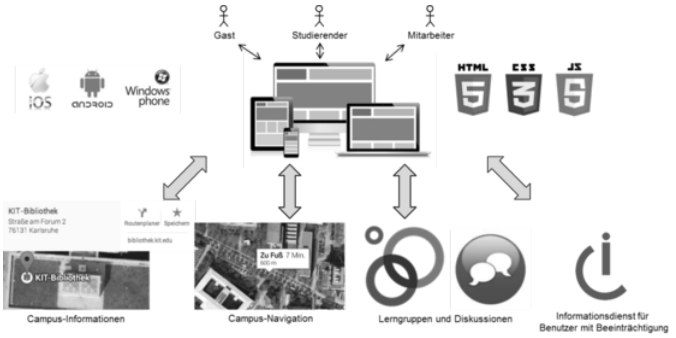

Figure 3. SmartCampus [25].

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 4, the previously described CompetenceService is illustrated in the form of a component diagram. As integration

technology according to Newman [11], REST was chosen since we are dealing with semantic information that should be reflect by the API. For implementation of the Competence-Service, the Java framework Spring was used.

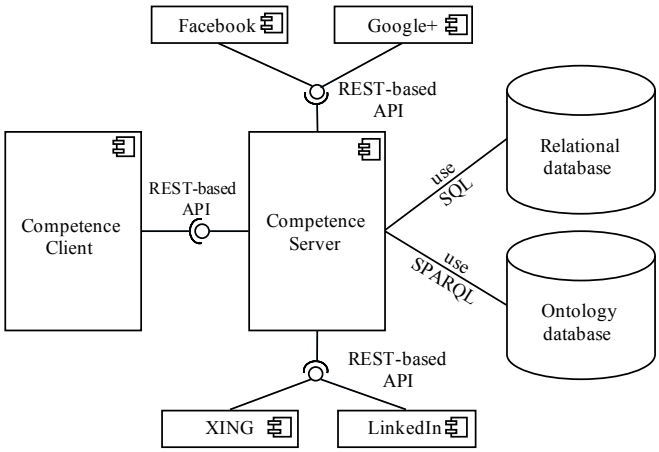

Figure 4. Component model of the CompetenceService.

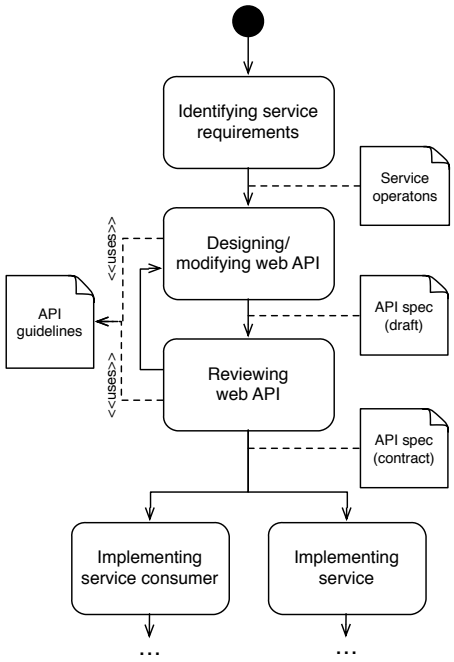

Figure 5. API-First engineering principle.

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.

B. API-Driven Development Process

For the design and development of our services as part of the previously introduced SmartCampus, we have decided to choose the API-First key engineering principle. The API-First engineering principle allows us to focus on the quality of the API design before any implementation effort is made. In our eyes, the APIs are an important and highly valuable business asset since they define what we can do with the system.

In Figure 5, our API-First engineering principle is shown. First, we identify the service requirements that we have gathered from scenarios. Scenarios are described from the perspective of a user that interacts with the software by using its user interface [26]. The resulting service operations are used for the first draft of the API. For the design of the API, we have defined some API guidelines based on the checklist in this article to ensure the consistency, maintainability, and usability of our service landscape. The API guidelines represent decisions that we have taken in previous development projects. After the first draft of the API specification is created, an ample peer review based on the checklist and the application scenario will be conducted by a dedicated team to ensure the quality of the API. For the API specification, we rely on the OpenAPI as a vendor neutral API specification format (former Swagger specification) since it is open source and is supported by a vast majority of big companies, such as Google, Microsoft, and IBM [27]. When reviewing the API, the responsible developers have to explain their design decisions. If the resulting quality report revealed some issues, the iteration cycle starts from the beginning with the modification of the API specification draft. In the other case, the API gets the status of a contract and will be handed over to the appropriate development teams.

V. Checklist for the API Design of Web Services Based on REST

This section presents eight different categories of best practices for designing REST-based web services, whereby each one is represented by a subsection (see Figure 6). The categories semantically group the found best practices and shall act as a checklist of important aspects that have to be considered when designing APIs for web services based on REST. Furthermore, we are providing recommendations for each category based on literature, that we have found during our conduction. The best practices should not be seen as strict guidelines. 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 RMM can be used to analyze the preconditions of a REST-based web service (see Section II-B).

A. 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 [18]. 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 [18]. But, web services based on REST do not need to be versioned due to hypermedia. The hypermedia aspect allows us to update the hyperlinks at runtime since the client does not hardcoded them in its

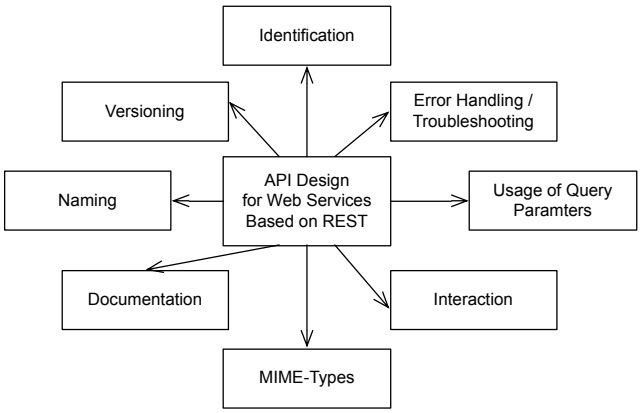

Figure 6. Categories of best practices for designing REST-based web services

client code and is, therefore, up to date as soon as the service provides a new representation of a requested resource.

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" [28] since it increases the effort for maintaining the web service.

B. Naming

The naming and description of resources correlates with the usability of the web service since the resources represent or abstract the underlying domain model. Furthermore, by defining guidelines for naming, we can ensure a consistent naming style across several services within the service-oriented architecture. This leads to a consistent look and feel regardless of the development team, who has designed and developed it. For this category, five best practices could be identified:

1) According to Vinoski [21], Papapetrou [20] and Mulloy [18], nouns should be used for resource names. Since a subresource is simply a resource with a composition relationship to another resource, we think this rule should be applied here as well.
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 [18] [20].
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 [18].
4) The mixture of plural and singular by naming resources should be prevented to ensure consistency. In addition, a resource should be able to handle several entities instead of just one one. But, there may be exceptions, such as Spotify's me resource [18] [19].
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 [3] [18] [29]. For instance, Google has defined an extensive styleguide [30].

Figure 7 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 7. Example for description of resources.

C. Resource Identification

According to Fielding [6], URIs should be used for unique identification of resources. If we take it accurately, there is no need for declaring best practices for resource identification when following a hypermedia approach since only the meta-information of the hyperlink will be evaluated and processed by the service consumer. But, we have found several best practices regarding resource identification. On the basis of this result, we assume that most so-called REST-based APIs are positioned on the second level of the RMM rather than third one. That is why we have decided to list the found best practices to improve the usability for this kind of API:

1) An URI should be self-explanatory according to the affordance [18]. The term affordance refers to a design characteristic by which an object can be used without any guidance. Since the main part of a URL consists of resource names, we have to ensure that these names are also domain-related and not termed in an abstract way.
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 [18].
3) The identifier of a specific state should be difficult to predict [20] and not references objects directly according to the Open Web Application Security Project (OWASP) [31], 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 [18] [19].

Figure 8 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 8. Example for identification of resources.

D. Error Handling and Troubleshooting

As already mentioned, the API represents the central access point web service, which is comparable with a provided interface of a software component [32]. Each information about the implementation of the service is hidden by the interface. Therefore, only the outer behavior can be 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. At this time, there are over 60 different status code with different semantic [18] [19].
2) Specific HTTP status codes should be used according to the official HTTP specification [33] and the extension [34] [19] [21] [20].
3) A detailed error message should be given as a hint for the error cause on client side [18] [19]. That is why an error message should consist of six ingredients: 3.1) An absolute Uniform Resource Locator (URL) that identifies the problem type, 3.2) A short summary of the problem type, that is written in english and comprehensible for software engineers, 3.3) The HTTP status code that was generated by the origin server, 3.4) An application specific error code, 3.5) A detailed human readable explanation specific to this occurrence of the problem and 3.6) An URL with further information about the specific error and occurrence.
4) For operational troubleshooting, it would be beneficial to use an application-specific or unique identifier that will be send with each request. This allows a filtering of service logs, when an issue was reported.

Figure 9 illustrates the mentioned ingredients of an error message according to the third best practice of error handling.

```
1  HTTP/1.1 503 SERVICE UNAVAILABLE
2  /* More header information */
3  {
4   "problem":
5   {
6     "type": "http://httpstatus.es/503",
7     "status": 503,
8     "error_code": 173,
9     "title": "The service is currently under
          maintenance",
10    "detail": "Connection to database timed out",
11    "instance": "http://.../errors/173"
12   }
13 }
```

Figure 9. Example for detailed error message.

E. Documentation

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” [35]. 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 [35], 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) [36], 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.

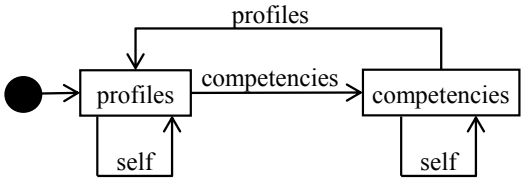

Figure 10. Example for documentation of the Web API.

Figure 10 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*.

F. Usage of Query Parameters

Each URI of a resource can be extended with parameters to forward optional information to the service. This is important when operating on the result set before transmitting over the network. For instance, the selection of relevant information can reduce the transmitted payload. We have identified five different use cases since they will be supported by several platforms, such as 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 11 illustrates how a special user group can be fetched by using a query language [19].

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

Figure 11. Filtering information by a using query language.

2) Sorting: For information sorting, Jauker [19] 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 12 illustrates how information can be sorted by using the attributes *education* and *experience*.

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

Figure 12. 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 [18] and Jauker [19] recommend a comma separated list of attributes and the term *fields* as the URI parameter. Figure 14 represents an example how the desired information can be selected before transmitting over the network.

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

Figure 13. A selection of resource information.

4) Search: The search for relevant information is a common use case. It is recommended to use the default query parameter *q* or using entity attributes, such as *id* or *uuid*.

```
1 GET /profiles?q=Java;Scala
```

Figure 14. A search of relevant information.

5) 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*).

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

Listing 1. Requesting 10 profiles by using pagination.

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 [18] [19]. 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 [18] 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" [18, p. 12]. Figure 1 illustrates a request using pagination on the resource *profiles*.

Although, the mentioned pagination technique is often recommended in literature, there are some issues especially when dealing with big data volumes or when fetching two virtual pages during inserting or deleting operation. These issues are outlined in [37].

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 five different best practices:

1) According to Jauker [19] and Mulloy [18], 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 [21].
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 or when dealing with Cross-Origin Resource Sharing CORS) [38].
3) A compression mechanism, such as GZIP should be supported to reduce the payload. By using the HTTP header field "Accept-Encoding", the client can indicate that he expects an appropriate encoding while on the other side, the server can set the "Content-Encoding" field when using content encoding. If the client does not set the "Accept-Encoding", the server should use compression by default.

4) 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 [21].
5) The support of partial updates should be considered so that the client does not have to send unchanged information. This is relevant when sending a large amount of data since the bandwidth of upstream is usually much lower than for downstream.

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
OPTIONS	Yes	Yes
PATCH	No	No

H. 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) [18].
2) JSON should be the default representation format since its increasing distribution [18].
3) Existing MIME types should be used, which already support hypermedia, such as JSON-LD (JSON for Linking Data), Collection+JSON and Siren [21].
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 [21].

VI. QUALITY-DRIVEN PRIORITIZATION OF BEST PRACTICES

After describing the identified categories of best practices for the API, the next logical step is the selection of relevant best practices for a given scenario. That is why we have mapped them on quality characteristics of the ISO/IEC 25010:2010 [10] (see Figure 15). For illustration purpose, we take the assumption that API of the current CompetenceService has to be optimized for mobile platforms. This means, for example, that the necessary requests for getting the needed information of the service have to be minimized and transmitted as fast as possible to reduce the latency and improve the responsiveness for the consumers. If we look at Figure 15, this will comply with the time behaviour quality sub characteristic as part of the performance efficiency quality characteristic.

(Sub)Characteristic
Functional suitability
Functional completeness
Functional correctness
Functional appropriateness
Performance efficiency
Time behaviour
Resource utilization
Capacity
Compatibility
Co-existence
Interoperability
Usability
Appropriateness recognizability
Learnability
Operability
User error protection
User interface aesthetics
Accessibility

Reliability
Maturity
Availability
Fault tolerance
Recoverability
Security
Confidentiality
Integrity
Non-repudiation
Accountability
Authenticity
Maintainability
Modularity
Reusability
Analysability
Modifiability
Testability
Portability
Adaptability
Installability
Replaceability

Figure 15. Product quality model of the ISO/IEC 25010:2011 [10]

After gathering the non-functional requirements or quality requirements for the API, we can select the appropriate categories and best practices that have to be considered. The method for the quality-driven prioritization approach is illustrated in Figure 16.

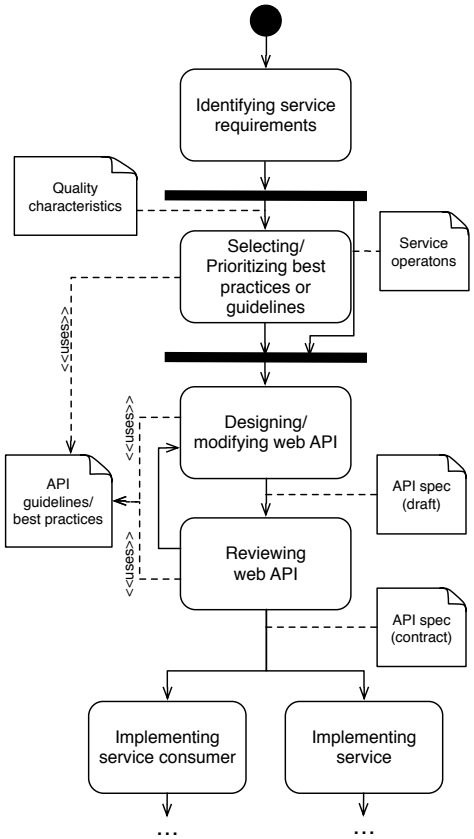

Figure 16. Revised API-First engineering principle with quality-driven prioritization

A. Best Practices for Performance Efficiency

For the improvement of the performance efficiency according to the ISO/IEC 25010:2010 [10], we have identified five relevant best practices that should be considered when designing an API. In particular, we make no statement regarding the completeness of the upcoming list of best practices for the performance efficiency.

1) Usage of Query Parameters (see Section V-F): The required network bandwidth load for transferring the response can be further reduced by a preprocessing step on service side based on given optional parameters. The optional parameters can be handed over through query parameters. For instance, a subset of representation fields can be selected by using the appropriate query parameter *fields*. Besides, the parsing effort on service consumer side can also lead to an increasing responsiveness.
2) Support of HTTP-OPTIONS (see Section V-G-2) should be implemented to allow requesting the supported HTTP methods. Keep in mind, the support is necessary when dealing with CORS and asynchrony.
3) Use Compression (see Section V-G-3): Each transferred information over the shared network should be compressed since mobile networks have often higher latency and lower bandwidth compared to traditional networks.
4) Support of conditional GET (see Section V-G-4) should be implemented by the service so that only new information will be send to the client.
5) Support partial updates (see Section V-G-5): The bandwidth for upstream is lower than for downstream so that the service should be able to handle partials updates and not expect a full representation of a resource.

Besides these mentioned best practices, we have also recognized that the number of requests should be minimized in a mobile scenario. But, we have found no best practice that can be directly mapped onto this. In our eyes, this can be achieved by one of the following approaches. For the sake of completeness, we have also list a non REST-like approach at the end.

1) Combining resources to a new more abstract resource tailored to the specific use case that is needed on the mobile client.
2) Providing an additional orchestration layer (e.g. Backend-For-Frontend (BFF) [11]) that splits the client request in multiple server requests and responds with a collected response.
3) Using a technology-driven and not REST-like approach, such as GraphQL that offers a declarative way for fetching the required information [39].

By following the mentioned best practices, we have could improve the perceptible responsiveness of the mobile client application. Figure 17 illustrates the average latency of two different versions of one mobile client - 1) before and 2) after application of the mentioned best practices that results in an API change. At the beginning, the mobile client application requests some initial data plus some app configuration so the latency is typically higher compared to the further course of the usage.

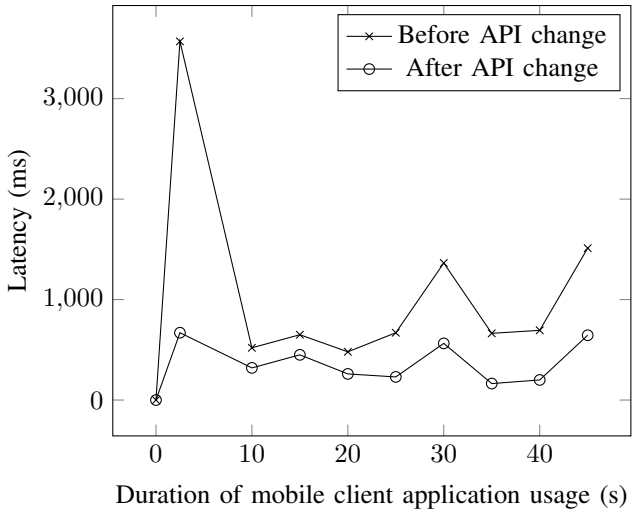

Figure 17. Mobile client application latency before and after applying the quality-driven selection of best practices

B. Mapping of Best Practices and Quality Characteristics

In the previous section, we have illustrated the quality-driven approach by selecting best practices according to their influence on the performance efficiency. This section shall focus the whole set of the quality characteristics by presenting a influence map from the mentioned best practices onto the quality characteristics of the ISO/IEC 25010:2011 [10] (see Figure 18). The dashed lines represent positive impacts of the best practices on the quality characteristics.

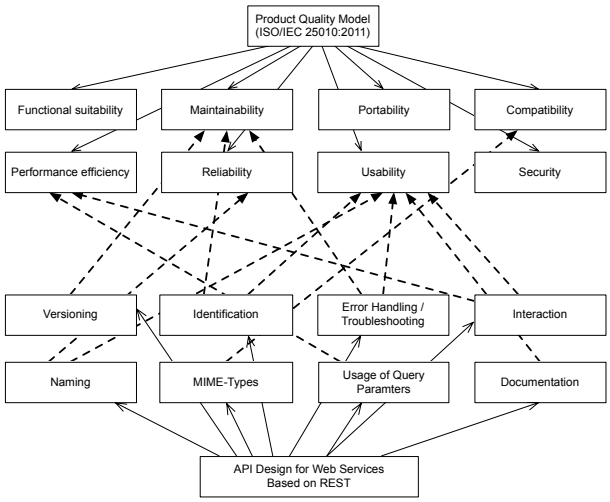

Figure 18. Influence map of best practices for API design for web services based on REST onto the ISO/IEC 25010:2011 [10]

The intention is to support API designers especially for web services based on REST by a kind of prioritization of best practices. In our eyes, the selection of best practices based on quality characteristics is more intuitive than traditional approaches, such as the requirement-level indication of the RFC 2119 [40]. The reason for this is that the importance of different best practices can vary from organization to organization.

VII. Summary and Outlook

In this article, we created a checklist for the API design of web services based on REST. More precisely, we identified and collected best practices for the API design. These best practices were classified into eight different categories that focus on different aspects of an API. The categories and best practices represent the checklist: Each category describes an aspect that has to be considered when designing an API. The best practices within one category represent recommendations that have to be kept in mind when the category as aspect is important for a specific API. For instance, one category is versioning. If versioning is something that is important for the specific API, then the best practices within this category should be kept in mind when designing the API. Finally, we associated the categories with quality characteristics of the ISO/IEC 25010:2010 [10] to show the impact of best practices on the quality of the API.

The intention of this article was not to reinvent the wheel. For this reason, the best practices within this article were re-used from existing work. Instead, the focus of this work was to identify and collect existing best practices, to unify them, and to associate them with certain aspects of an API design. Best practices are only helpful when software architects and developers know when to consider them and what they are for. For this reason, our classification into eight categories helps to decide, whether a certain best practice should be considered or not. Furthermore, our best practices are not meant to be complete. They are more a recommendation about what should be kept in mind when a certain aspect (category) is relevant for an API design.

To illustrate the applicability of our checklist, we applied the checklist, i.e., its categories and their best practices on a concrete scenario. As a scenario, we chose the SmartCampus of the Karlsruhe Institute of Technology. SmartCampus is a modern web application. Its purpose is to simplify the daily life of students, guests, and members at the university. The SmartCampus consists of several provided services. One service is the CompetenceService that semantically searches competences in the area of information technology. By applying our checklist, we could identify relevant best practices for our API. The checklist helped us to identify, which of the best practices are relevant for this service and which ones are not. The best practices are not strict guidelines; they are more recommendations that helped us to keep certain aspects in mind. By using the checklist during the API design, we had a concrete list about what to consider. This helped us to not forget relevant aspects when creating the API.

Summarized, the checklist, i.e., the categories and their best practices help software architects and developers to design the API of web services with certain recommendations kept in mind. As today, best practices are distributed across several existing works, until now, it was hard to find a unified set of best practices. Furthermore, the best practices were isolated. It was not clear, whether a certain best practice should be considered or not. Its impact was not obvious. With our classification into eight categories, software architects and developers get the possibility to filter the best practices and to understand, which ones are necessary and which ones are not relevant for a certain API design. This reduces the amount of best practices to the relevant ones and simplifies the application of best practices. Even though our checklist is not meant to be complete, it is a helpful list of best practices, i.e., recommendations. This list reminds software architects and developers of aspects they should consider when designing an API with certain categories being relevant. Furthermore, for software architects and developers it is not necessary any more to lookup best practices in literature. As the checklist is a first collection and unification of widespread best practices, software architects and developers can directly start with this checklist in their daily projects. As we build on existing work and reference this work, detailed descriptions can be looked up if necessary. But it is not necessary to spend time to collect best practices from scratch. With the association to quality characteristics of ISO/IEC 25010:2010 [10], software architects and developers get an understanding about the impact of the best practices and certain design decisions on the quality of the API. This increases the awareness of design decisions and helps to create APIs in a quality-oriented manner.

In the future, we plan to investigate the impact of our checklist on the development speed, as well as the quality. To evaluate the usefulness of the best practices for API design, we consider setting up two teams of students, Team A and Team B. Both teams get 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 best practices for API design. However, Team A will be equipped with our checklist. We expect that Team A will spend much less time searching appropriate best practices. The checklist will provide Team A appropriate best practices about how to design the API. Figure 19 shows the expected results.

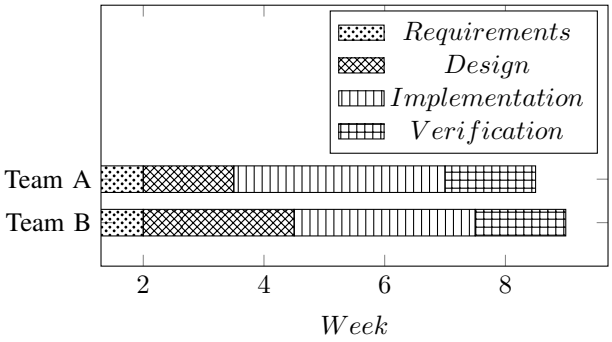

Figure 19. Duration of the development phases in weeks.

In addition, we plan to extend our checklist with further best practices and 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 [41]. Besides, we will use the checklist on upcoming projects an exemplifies the whole API design process in a more detail so that each development team is capable of applying it on its own projects.

References

[1] P. Giessler, M. Gebhart, D. Sarancin, R. Steinegger, and S. Abeck, "Best Practices for the Design of RESTful web Services," International Conferences of Software Advances (ICSEA), 2015. [Online]. Available: http://www.thinkmind.org/download.php?articleid=icsea_2015_15_10_10016

[2] 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].

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

[4] Spotify, "Web API Endpoint Reference," URL: https://developer.spotify.com/web-api/endpoint-reference/ [accessed: 2016-09-30].

[5] Twitter, "REST APIs," URL: https://dev.twitter.com/rest/public [accessed: 2016-09-30].

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

[7] B. Iyer and M. Subramaniam, "The Strategic Value of APIs," URL: https://hbr.org/2015/01/the-strategic-value-of-apis [accessed: 2016-07-26].

[8] Deloitte, "API economy - From systems to business services," URL: http://dupress.com/articles/tech-trends-2015-what-is-api-economy/ [accessed: 2016-09-30].

[9] M. Gebhart, P. Giessler, and S. Abeck, "Challenges of the Digital Transformation in Software Engineering," International Conferences of Software Advances (ICSEA), 2016. [Online]. Available: http://www.thinkmind.org/download.php?articleid=icsea_2016_5_30_10067

[10] ISO/IEC, "Std 25010:2011 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models," International Organization for Standardization, Geneva, CH, Standard, 2011.

[11] S. Newman, Building Microservices. O'Reilly Media, Incorporated, 2015.

[12] 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.

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

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

[15] Evans, Domain-Driven Design: Tacking Complexity In the Heart of Software. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[16] C. Pautasso and E. Wilde, "Why is the web loosely coupled? a multi-faceted metric for service design," in 18th World Wide Web Conference (WWW2009), ACM. Madrid, Spain: ACM, April 2009, pp. 911–920.

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

[18] 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].

[19] 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].

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

[21] 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

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

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

[24] D. D. Guinard and V. M. Trifa, Building the Web of Things With examples in Node.js and Raspberry Pi. Manning, 2016.

[25] C&M, "The system SmartCampus and the project SmartCampusbarrier-free," URL: http://cm.tm.kit.edu/smartcampus.php [accessed: 2016-09-30].

[26] M. B. Rosson and J. M. Carroll, "The human-computer interaction handbook," J. A. Jacko and A. Sears, Eds. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 2003, ch. Scenario-based Design, pp. 1032–1050. [Online]. Available: http://dl.acm.org/citation.cfm?id=772072.772137

[27] OAI, "Open API Initiative," URL: https://openapis.org/ [accessed: 2016-09-30].

[28] 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].

[29] 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].

[30] Google, "Google JSON Style Guide," 2015, URL: https://google.github.io/styleguide/jsoncstyleguide.xml [accessed: 02.12.2015].

[31] 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].

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

[33] 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.

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

[35] 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].

[36] 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].

[37] M. Winand, "We need tool support for keyset pagination," 2014, URL: http://use-the-index-luke.com/no-offset [accessed: 2015-12-23].

[38] W3C, "Cross-Origin Resource Sharing," URL: https://www.w3.org/TR/cors/ [accessed: 2016-09-30].

[39] Facebook, "GraphQL," URL: https://facebook.github.io/graphql/ [accessed: 2016-09-30].

[40] S. Bradner, "Rfc 2119, key words for use in rfcs to indicate requirement levels," 1997, URL: http://www.rfc-base.org/txt/rfc-2119.txt].

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