Evaluating the RESTfulness of "APIs from the Rough"

Arne Koschel[1], Irina Astrova[2], Maximilian Blankschyn[1], Dominik Schöner[1] and Kevin Schulze[1]
[1]*Faculty IV, Department of Computer Science, Hannover University of Applied Sciences and Arts, Ricklinger Stadtweg 120, 30459 Hannover, Germany*
[2]*Department of Software Science, School of IT, Tallinn University of Technology, Akadeemia tee 21, 12618 Tallinn, Estonia*

Keywords: REST (Representational State Transfer), RESTful, API (Application Programming Interface), Richardson Maturity Model.

Abstract: Nowadays, REST is the most dominant architectural style of choice at least for newly created web services. So called RESTfulness is thus really a catchword for web application, which aim to expose parts of their functionality as RESTful web services. But are those web services RESTful indeed? This paper examines the RESTfulness of ten popular RESTful APIs (including Twitter and PayPal). For this examination, the paper defines REST, its characteristics as well as its pros and cons. Furthermore, Richardson's Maturity Model is shown and utilized to analyse those selected APIs regarding their RESTfulness. As an example, a simple, RESTful web service is provided as well.

1 INTRODUCTION

Following Roy T. Fielding, the father of the REST architectural style, at least back in 2008, an apparently frustrating number of APIs calling themselves RESTful were not. Leonard Richardson therefore introduced a maturity heuristic for REST referred to as Richardson Maturity Model that allows web service APIs to be grouped into different levels of maturity.

This paper defines REST and its characteristics. Furthermore, Richardson Maturity Model with its four different levels is explained. This paper is an extension of our previous work (Koschel, 2019), where the maturity levels of ten freely accessible RESTful APIs have been evaluated. This evaluation is extended here by an in-depth analysis of one API, which reaches all the levels of Richardson Maturity Model, and another API, which does not. In addition, the advantages, disadvantages and challenges of creating a truly RESTful API are explained. Subsequently, an exemplary REST implementation based on Java Spring HATEOAS is presented. Finally, an overall conclusion is drawn as well as some outlook to future work.

2 RESTful API

A RESTful API is an API that uses HTTP requests to GET, PUT, POST and DELETE data. A RESTful API – also referred to as a RESTful web service – is based on the REST technology, an architectural style and approach to communications often used in web services development.

This section discusses the characteristics of RESTful APIs as defined by Roy T. Fielding in 2000: client-server model, stateless operations, caching, uniform interface, layered system and code on demand (Fielding, 2000). He formulated these characteristics as constraints that describe what REST is at different maturity levels.

2.1 Client-server Model

The first constraint concerns the introduction of the client-server model. Since it is the basis for almost all network applications, this constraint can be considered implicit. It states that a distinction is made between a client and a server, whereby the client makes requests to the server; the server in its turn offers a certain service, receives the request from the client and responds to the request. It follows from this model that the client and the server are largely independent and can therefore be developed independently of each other.

Koschel, A., Astrova, I., Blankschyn, M., Schöner, D. and Schulze, K.
Evaluating the RESTfulness of "APIs from the Rough".
DOI: 10.5220/0008166802810288
In *Proceedings of the 15th International Conference on Web Information Systems and Technologies (WEBIST 2019)*, pages 281-288
ISBN: 978-989-758-386-5

2.2 Stateless Operations

This means that all communications must be stateless, or the state must only be kept on the client side and the server has no knowledge of it. Each request must contain all information so that it can be understood and processed by the server. It is possible for each server to process each request. This restriction increases scalability. It does not matter which server answers which request.

A disadvantage of statelessness is that the requests are larger due to the increased information content compared to the state liability. This causes more network traffic.

2.3 Caching

The disadvantage of more network traffic, which is discussed in the case of statelessness, can be counteracted by the use of cache. Roy T. Fielding defines this as a further constraint to use the network more efficiently. Each response to a request must indicate whether the delivered content is cache enabled or not. This reduces the number of requests. A cached response can be answered locally from the cache and does not have to be sent to the server.

Caching also improves performance from the user's point of view, since cached responses are available immediately. Here it must be ensured that the cached responses are not obsolete, which otherwise impairs the reliability.

2.4 Uniform Interface

This constraint is a central point at REST. It indicates that the implementation of the interface is disconnected from the service provided. This allows both to develop independently of each other. This is an advantage over classic web services that work with WDSL and SOAP because the interface (defined in WDSL) has to be recreated when the service changes.

However, to achieve this, there are further constraints. All resources of the service are identified by URI (Uniform Resource Identifier). A resource can be anything: a customer, a shopping cart, a PDF document, a collection of other resources, etc. A change of a resource is done by representing the resource. This can lead to more persistent entities than in a non-REST design (Tilkov, 2015).

The representation of the resource can be in different formats – often XML, JSON or HTML is used. Depending on which client requests the resource, it can also be present in several representations and sent in a certain format depending on the application.

The uniform interface also includes the HATEOAS principle (see Section 4).

2.5 Layered System

The constraint of the layered system means that resources can run in different layers and therefore on different servers. For example, a RESTful API is on Server A, an authentication service is on Server B and the data is on Server C.

A use of layers also increases scalability. They are independent of each other and can be added to other servers in a layer depending on the load.

A potentially disadvantage is that this results in overhead of higher latencies. However, there is the option to use shared caching, as mentioned above.

2.6 Code on Demand

Most of the time a server will send back static representations of resources in the form of XML or JSON. However, when necessary, servers can send executable code to the client.

3 RICHARDSON MATURITY MODEL

This is a model used to determine the maturity of a web service in terms of its "REST" characteristics, described above. Richardson's Maturity Model (Fowler, 2010), (Richardson, 2009), (Betten, 2011) has four maturity levels – from Level 0 to Level 3 – to represent the degree of the use of HTTP (see Figure 1).

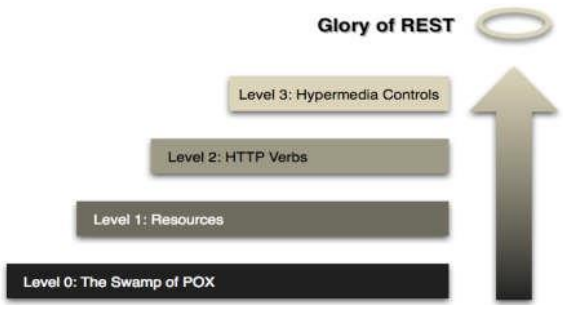

Figure 1: Richardson Maturity Model (Fowler, 2010).

Richardson Maturity Model starts at Level 0 and adds architectural conditions at each next level. In

principle, it depends on how much of the possibilities of HTTP are used. It should be mentioned that a web service that does not fully meet the requirements of one level can no longer reach the next level.

3.1 Level 0

Typical representatives at this level are SOAP and XML-RPC. Level 0 is often called Swamp of POX (Plain Old XML). Services at Level 0 only use HTTP as a transport protocol for RPC calls. A service is viewed as a black box with unaddressed resources. Example: http://example.com.

3.2 Level 1

Resources are the central concept of a REST architecture. At this level, each resource is given its own URI. As a result, communication does not take place via URIs as it was done by services at Level 0. Rather, resources are addressed by using URIs. Example: http://example.com/user/1234

3.3 Level 2

At this level, most (if not all) HTTP methods also known as HTTP verbs are used correctly on resources and according to their specification. These methods are:

- GET – It is used for the retrieval of information identified by a URI in the form of a representation.
- POST – It is used for the creation of a new resource and for all purposes in which none of the other methods fit.
- PUT / PATCH – It is used for updating an existing resource.
- HEAD – It is used for querying metadata (e.g., to check resource status).
- DELETE – It is used for the deletion of an existing resource.
- OPTIONS – It is used for querying resource metadata (e.g., to find out which methods a resource supports).

In addition, the HTTP status codes are used correctly to inform the client about the resource status. For example, a response error message should not return status code 2xx, but 4xx (Fowler, 2010).

3.4 Level 3

Hypermedia turns a web service into REST. The idea is to link all content (resources or their representations) in one website. The client does not need to type in a URI itself and can easily follow links. There are two types of links: links that lead directly to other resources and links that change the status of a resource, e.g., using the HTTP methods such as PUT or DELETE.

4 HATEOAS

A RESTful API can implement the HATEOAS (Hypermedia as the Engine of Application State) principle, which states that a resource responds to a request with all possible state transitions in addition to answering it. A special characteristic of REST is that all state transitions of the application are performed by methods of hypermedia (i.e., URIs on resources). Thus, from the current state, a RESTful API must output all possible state transitions and the method of transition. This results in a self-describing RESTful API that theoretically no longer requires external documentation.

The navigation between the resources is implemented by relations, so that the resource can change, but the relation remains the same. The completed server domain leads to a high degree of independence between a client and a server and they can be developed independently of each other.

The resources offered by RESTful API are dynamic, depending on the state of the application. For example, an administrator can be shown a more extensive context menu than an ordinary user. It is typical for HATEOAS to provide not only the possible state transitions but also the corresponding method and further information if needed. When HTTP is used, a strict distinction is made between verbs (GET, POST, DELETE, PUT) and idempotent and changing methods (Tilkov, 2015). That is, "the point of hypermedia controls is that they tell us what we can do next, and the URI of the resource we need to manipulate to do it" (Inden, 2016).

5 ADVANTAGES, DISADVANTAGES AND CHALLENGES OF RESTFUL APIS

These are the advantages of using a REST architecture (Stringfellow, 2017):

- **Separation between Client and Server:** The independence a client from a server enables

developments in different areas of a project independently of each other.

- **Language Independence:** RESTful APIs can use PHP, Java, Python or Node.js servers. It is just important that responses to requests are always in the language used for information exchange, usually XML or JSON.
- **Scalability:** The separation between a client and a server allows a product to be easily scaled by a development team.
- **Flexibility and Portability:** Migration from one server to another is possible at any time.
- **Simplicity:** REST is based on HTTP, so the concept is easy to learn.

Disadvantages and challenges of using a REST architecture (Kumari, 2015), (Little, 2013) are:

- **Complexity:** HATEOAS is complex to implement, dependencies and processes must be clear.
- **Higher Payload:** Mobile devices with poor Internet connection have to deal with higher payload
- **Limited Usage:** REST is not suitable for large amounts of data. Moreover, when RESTful APIs are used in social media, WEB chat and mobile services, downward compatibility must be ensured, since the client side can be unknown.

6 ANALYSIS OF PUBLICLY AVAILABLE RESTFUL APIS AND THEIR MATURITY LEVELS

In the following, we examine ten RESTful APIs that have an Alexa Traffic Rank <= 1000 in Germany. As described in Section 3, RESTful APIs can be divided into different maturity levels: from 0 to 3. This paper focuses on RESTful APIs that have been publicly available and professionally developed. Richardson's hypothesis that many RESTful APIs are not REST compliant or not RESTful - and thus are not RESTful APIs in the true sense of the word - is to be proved on the basis of the selected RESTful APIs.

6.1 Procedure

The interaction with the RESTful APIs was used to find out the maturity levels. It was checked if the RESTful APIs meet the criteria of Richardson Maturity Model from Level 0 to Level 3. It should be mentioned that an API that does not fully meet one level can no longer reach the next level.

Next Twitter RESTful API and PayPal RESTful API are examined on the degree of maturity after Richardson. It is shown exemplarily how such examination was done. The achievement of each maturity level was documented with extracts of the responses and thereby, the procedure was presented.

6.2 Analysis of Twitter RESTful API

Twitter RESTful API is used to create, retrieve and delete tweets. Figure 2 shows the existing functions of Twitter RESTful API. As can be seen, only POST and GET methods are used. The POST method is also used as the "destroy" function. According to Richardson Maturity Model, this is a violation of the criteria at Level 2 – a proper verb for the destroy function might be DELETE.

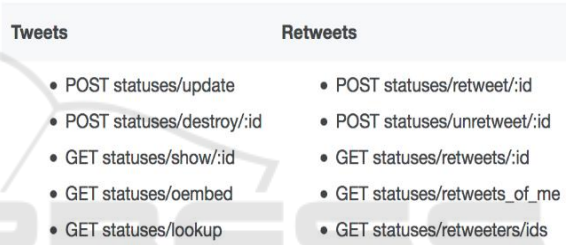

Figure 2: Functions of Twitter RESTful API.

Figure 3 shows a response of the GET method. As can be seen, HTTP was used as the transport protocol. Thus, Level 0 is reached.

```
<- "GET
/1.1/statuses/show.json?id=10826535751662100
49 HTTP/1.1 Accept-Encoding:
gzip;q=1.0,deflate;q=0.6,identity;q=0.3
Accept: */* User-Agent: OAuth gem v0.5.4
Content-Type: application/x-www-form-
urlencoded
Authorization:
OAuth
oauth_consumer_key=\"***\",
oauth_nonce=\"***\",
oauth_signature=\"***\",
oauth_signature_method=\"HMAC-SHA1\",
oauth_timestamp=\"1546960223\",
oauth_token=\"*****\",
oauth_version=\"1.0\"
Connection: close Host: api.twitter.com
Content-Length: 0"
<- ""
   -> "HTTP/1.1 200 OK"
```

Figure 3: GET Response – Header Detail 1.

Furthermore, it was also deduced from the request of the GET on a certain tweet that resources

are used. This is a criterion for Level 1. That is, the tweet is a resource of its own, whose representation can be requested using the GET method. The returned result in JSON format is shown in Figure 4.

```
   -> "content-disposition: attachment;
filename=json.json"
-> "content-encoding: gzip"
-> "content-length: 780"
-> "content-type:
application/json;charset=utf-8"
```

Figure 4: GET Response – Header detail 2.

Figure 4 is an excerpt from the response header. In addition to the output format JSON, the character set used (UTF-8) is also included in the response. This ensures that the client can decode the response correctly. The representation of the resource in JSON format is shown in Figure 5.

Figure 5 illustrates that although resources are used, the URI is not returned as such. So the request to the GET method must be compiled from the “id_str”. As a result, a criterion of self-describing messages required at Level 3 is not fulfilled. Moreover, HATEOAS is not used. Therefore, we came to the conclusion that Twitter RESTful API does not reach Level 3 yet and Level 2 is not complete because not all HTTP verbs are used.

```
GET
/1.1/statuses/show.json?id=108265357516621
008
   {
     "created_at": "Tue Jan 08 15:01:41
+0000 2019",
     "id": 1082653575166210000,
     "id_str": "1082653575166210049",
     "text": "Test Tweet using the
Twitter RESTful API and twurl",
     "truncated": false,
     "entities": {
       "hashtags": [],
       "symbols": [],
       "user_mentions": [],
       "urls": []
     }
```

Figure 5: GET Response – JSON detail.

Figure 5 illustrates that although resources are used, the URI is not returned as such. So the request to the GET method must be compiled from the “id_str”. As a result, a criterion of self-describing messages required at Level 3 is not fulfilled. Moreover, HATEOAS is not used. Therefore, we came to the conclusion that Twitter RESTful API does not reach Level 3 yet and Level 2 is not complete because not all HTTP verbs are used.

6.3 Analysis of PayPal RESTful API

As described in Section 3, the only criterion for reaching Level 1 is that HTTP is used as a transport protocol. This is given by PayPal RESTful API, as can be seen from an HTTP response header of PayPal RESTful API in Figure 6. Thus, the requirements for Level 0 are fulfilled; however, it is not yet possible to speak of a RESTful API - as mentioned above, classic RPCs also use HTTP as a transfer protocol with SOAP.

```
   * TLSv1.2 (IN), TLS handshake,
Finished (20):
* SSL connection using TLSv1.2 / AES256-
SHA256
* ALPN, server did not agree to a protocol
* Server certificate:
*  subject: C=US; ST=California; L=San
Jose; O=PayPal, Inc.; OU=PayPal
Production;
CN=api.sandbox.paypal.com
*  start date: Aug 21 00:00:00 2018 GMT
*  expire date: Aug 20 12:00:00 2020 GMT
*  subjectAltName: host
"api.sandbox.paypal.com" matched cert's
"api.sandbox.paypal.com"
*  issuer: C=US; O=DigiCert Inc;
CN=DigiCert Global CA G2
*  SSL certificate verify ok.
> GET
/v1/invoicing/invoices?<param>  HTTP/1.1
```

Figure 6: HTTP Response Header.

Since Level 0 is reached, a next maturity level can be examined now. According to Richardson, Level 1 states that each resource is assigned a URI. As shown in Figure 6, the GET method request is made for a specific resource (here `Invoice list`). To make it clear that resources and URIs are available, a reference is also made to the fact that each transaction is mapped as a separate resource and therefore, the criteria of Level 1 are fully met. An example URI of a PayPal transaction is shown in Figure 7.

As shown in Figure 7, the forward slash is used to map the hierarchical relationship between resources (Massé, 2012). From the response of this resource to the GET method request Level 2 and Level 3 can be justified.

Figure 8 shows how the requested transaction (with the self-representation per rel. “self”) can be

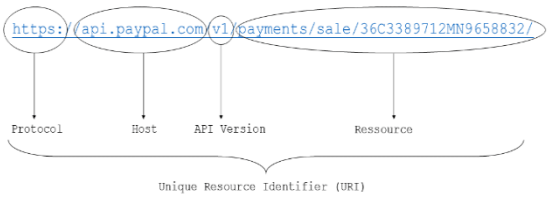

Figure 7: Representation of URI.

objected to or how a refund can be applied for (rel. "refund"). Here not only the representation requested by the GET method is returned, but also further hypermedia links. This follows the basic principle of HATEOAS. It should be noted that the RESTful API implementation of PayPal meets the high demands of Richardson Maturity Model; this is not self-evident in the APIs examined. Also in the PayPal documentation, it was mentioned that HATEOAS was considered.

```
{
     "links": [{
       "href":
"https://api.paypal.com/v1/payments/sale/3
6C38912
   MN9658832",
       "rel": "self",
       "method": "GET"
     }, {
       "href":
"https://api.paypal.com/v1/payments/sale
/36C38912MN9658832/refund",
       "rel": "refund",
       "method": "POST"
     }, {
       "href":
"https://api.paypal.com/v1/payments/
   payment/PAY-5YK922393D847794YKER7MUI",
       "rel": "parent_payment",
       "method": "GET"
     }]
```

Figure 8: HATEOAS in PayPal's RESTful API.

6.4 Summary

We analysed the ten RESTful APIs: Twitter, PayPal, Google Maps, Spotify, Youtube, Instagram, Github, Wunderlist, LinkedIn and OneDrive. Table 1 summarizes the results of our analysis.

Our analysis showed that the majority of the RESTful APIs (viz., 6 out of 10) did not reach Level 3 yet. This is probably due to a significant increase in the development efforts compared to Level 1 or Level 2. HATEOAS further increases the development efforts, but the fact that there are no established standards for this is aggravating the situation. On the other hand, our hypothesis that REST would be fully implemented in the rarest cases was not confirmed. For example, 4 of 10 RESTful APIs implemented HATEOAS and in addition, provided a very detailed documentation on their APIs. Almost all the examined RESTful APIs (viz., 9 of 10) reached Level 2. This could indicate that REST and its characteristics were given special consideration during the development. However, it has to be considered that an API on Level 2 is not worse than an API on Level 3 – this only describes the level of maturity according to Richardson.

Table 1: Summary of analysis results (Koschel, 2019).

API	Level 0	Level 1	Level 2	Level 3	Comment
Twitter	Yes	Yes	Partly	No	Not all http verbs used correctly, no HATEOAS
PayPal	Yes	Yes	Yes	Yes	HATEOAS implement.
Google Maps	Yes	Yes	No	No	Not all http Verbs used correctly
Spotify	Yes	Yes	Yes	Yes	HATEOAS implement.
Youtube	Yes	Yes	Yes	No	No HATEOAS
Instagram	Yes	Yes	Yes	Yes	HATEOAS implement.
Github	Yes	Yes	Yes	Yes	Very accurate HATEOAS implement.
Wunderlist	Yes	Yes	Yes	No	No HATEOAS
LinkedIn	Yes	Yes	Yes	No	No HATEOS
OneDrive	Yes	Yes	Yes	No	No HATEOS

7 EXAMPLE OF IMPLEMENTATION OF RESTFUL API – LEVEL 3

In this section, we present an example of a simple web service, which can be considered as a RESTful

API at Level 3. This service allows to access, list, create and modify a resource. It is based on Java Spring Boot and uses Spring HATEOAS, which already provides some APIs to ease creating REST representations that follow the HATEOAS principle.

First of all, this example models a simple `Employee` service that manages employees of a company. In Figure 9 the domain model is shown.

```
class Employee implements
Identifiable<Long> {

          @Id @GeneratedValue
          private Long id;
          private String firstName;
          private String lastName;
          private String role;

}
```

Figure 9: Employee domain model.

To reach Level 1, each identifiable resource must have its own URI. This can be achieved with the function shown in Figure 10. In a class `EmployeeController`, each employee has their own endpoint with an ID, which can be reached by using the following curl command:

```
$ curl -v localhost:8080/employees/1
```

```
@GetMapping("/employees/{id}")
Resource<Employee> one(@PathVariable Long
id) {
 Employee employee =
repository.findById(id)
  .orElseThrow(() ->
  new  EmployeeNotFoundException(id));

  return new
Resource<>(employee,linkTo(methodOn(Employ
eeController.class).one(id)).withSelfRel()
,linkTo(methodOn(EmployeeController.class)
.all()).withRel("employees"));
}
```

Figure 10: EmployeeController Class (1).

To create a RESTful API at Level 2, the HTTP verbs must be used correctly. With Spring HATEOAS, this can be easily implemented using Mapping Annotations, as shown in Figure 10. When looking at these figures, it becomes clear that the functions are mapped to the respective request (GET, POST and PUT) by their annotations. Thus, the verbs can be used in their intended way.

For a RESTful API at Level 3, resources and representations must be linked by hyperlinks (HATEAOS). As can be seen in Figure 11, this example provides extra information in addition to the resources' representation. A link to the employee overview is added, thereby enabling exploration from this resource to the next one within the answer.

```
{
  "id": 1,
  "name": "Bilbo Baggins",
  "role": "burglar",
  "_links": {
    "self": {
      "href":
"http://localhost:8080/employees/1"
    },
    "employees": {
      "href":
"http://localhost:8080/employees"
    }
  }
}
```

Figure 11: Employee HATEOAS response.

Our example showed the simple web service that meets the basic requirements of RESTful API at Level 3. Using Spring HATEOAS, the core problem of link creation and the representation assembly can be simplified.

8 CONCLUSION

In this paper, it was shown what REST is and its characteristics were described. Then Richardson Maturity Model was presented and its four levels were described. In addition, the advantages and disadvantages of RESTful APIs were highlighted and the field of application of REST was demonstrated. Finally, the ten freely available RESTful APIs were evaluated

During this evaluation, it was shown which of the maturity levels the APIs reach. The evaluation ultimately showed that 4 of 10 APIs meet the criteria for Level 3. An implementation with Java Spring Boot and Spring HATEOAS was shown as an example.

It should be mentioned that the classification of RESTful APIs according to Richardson Maturity Model does not provide any information about the quality of the API. Rather, it only shows that REST as an architectural style is widespread and used worldwide, but the glory of RESTfulness is not necessarily achieved. There is a need to consider whether HATEOAS should be introduced, which may be easier with new developments than with already existing APIs. HATEOAS can bring added value and simplify machine communication.

ACKNOWLEDGEMENTS

Irina Astrova's work was supported by the Estonian Ministry of Education and Research institutional research grant IUT33-13.

REFERENCES

Betten, S., 2011. Richardson Maturity Model, http://www.se.uni-hannover.de/pub/File/kurz-und-gut/ws2011-labor-restlab/RESTLab-Richardson-Maturity-Model-Sascha-Betten-kurz-und-gut.pdf

Fielding, R., 2000. Architectural Styles and the Design of Network-based Software Architectures, https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Fowler, M., 2010. Richardson Maturity Model, https://martinfowler.com/articles/richardsonMaturityModel.html

Inden, M., 2016. *Der Java-Profi: Persistenzlösungen und REST-Services*, dpunkt.verlag.

Koschel, A., Blankschyn, M., Schulze, K., Schöner, D., Astrova, I., Astrov, I., 2019. RESTfulness of APIs in the Wild, In *IEEE World Congress on SERVICES – Concise Papers*, 2 pages, to appear in IEEE.

Kumari, V., 2015. Web Services Protocol: SOAP vs REST, In *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)*, Volume 4 Issue 5. http://ijarcet.org/wp-content/uploads/IJARCET-VOL-4-ISSUE-5-2467-2469.pdf

Little, M., 2013. What Are The Drawbacks Of REST?, https://www.infoq.com/news/2013/05/rest-drawbacks

Massé, M., 2012. *REST API Design Rulebook*, O'Reilly.

Richardson, L., 2009. Act Three: The Maturity Heuristic, https://www.crummy.com/writing/speaking/2008-QCon/act3.html

Stringfellow, A., 2017. SOAP vs. REST: Differences in Performance, APIs, and More, https://dzone.com/articles/differences-in-performance-apis-amp-more

Tilkov, E., Schreier, W., 2015. *REST und HTTP*, dpunkt.verlag. Trans. Roy. Soc. London, vol. A247.