SOA-Readiness of REST

Peter Gorski, Luigi Lo Iacono, Hoai Nguyen, Daniel Torkian

SOA-readiness of REST

Peter Leo Gorski, Luigi Lo Iacono, Hoai Viet Nguyen, Daniel Behnam Torkian

Cologne University of Applied Sciences, Germany
{peter.gorski,luigi.lo_iacono,viet.nguyen,daniel.torkian}@fh-koeln.de

Abstract. SOA is a core concept for designing distributed applications based on the abstraction of software services. The main strength lies in the ability to discover services and loosely-couple them with service consumers across platform-boundaries. The evolved service protocol SOAP and its accompanying standards provide a stable, rich and wide-spread technology stack for implementing SOA-based systems.
As an alternative approach to design and implement distributed systems based on services, the architectural style REST gains traction, due to its more light-weight and data format independent nature. Whether REST is also suited for acting as a basis for implementing SOA-based systems is still an open issue, however. This paper focuses on this question and provides an analysis on the SOA-readiness of REST. Both, a theoretical analysis and an empirical study of REST frameworks have been conducted in order to obtain a comprehensive understanding on this matter. The results show a lack of core SOA principles mainly related to the discoverability and the loose coupling of services.

Keywords: SOA, REST, Service Discovery, Service Coupling

1 Introduction

The SOA (service-oriented architecture) [1] concept has become a prerequisite when it comes to the design of adaptable distributed software systems. Based on the notion of loosely-coupled services, functionality encapsulated in software services can be discovered and consumed by service clients at runtime. SOAP (simple object access protocol) [2] and its ambient standards have been for a notable period the only technology stack for implementing SOA-based systems. In recent years, the architectural style REST (representational state transfer) [3] has emerged as an alternative concept for designing distributed service-based applications. The most common technological basis for implementing REST-based services is the plain HTTP, what makes REST more lightweight and efficient than SOAP and keeps it data format independent.

Although REST has advantages in simple service invocation scenarios, the question remains whether it can still compete in settings in which an ad hoc discovery and coupling of services deployed in heterogeneous environments during system runtime is required. This paper investigates on this by examining the SOA-readiness of REST. First of all, an accurate understanding about the fundamental conditions which arise when applying REST to SOA is necessary

for a well-founded conclusion on the SOA-readiness of REST. Since besides the dissertation of Roy Fielding [3] no further specifications are available at this point which can be used as a foundation, Section 2 lays the ground by recalling essential properties of REST. The main components of a SOA, i.e. service consumer, service provider and service registry, are the subject of a theoretical analysis in Section 3. Evolved technologies are available to implement these basic elements in SOAP-based environments. Along these lines, the availability and suitability of comparable approaches for REST are investigated and discussed. In addition to these theoretical considerations, an empirical study of distinct REST frameworks has been conducted to explore on the current state of implementation interoperability especially in the light of automatic service coupling. The methodology of the analysis is introduced in Section 4 and the gathered results are presented and discussed in Section 5. The paper concludes with a summary on the findings and gives an outlook on further research, development and standardisation demands in Section 6.

2 REST

REST has been introduced by Roy Fielding as an architectural style for designing distributed hypermedia systems. REST targets the scalability of application interaction, uniformity of interfaces and independent evolvement of components and intermediates with the intention to reduce latency, enable security and provide long-lived services. The uniform interface serves as the transfer protocol for identifying, accessing and manipulating resources. Furthermore, resources have different representations in terms of data formats which can vary according to the needs of the clients. Thus, services are able to deliver content for human-driven applications such as web browsers which use the data to render a web page. In turn, a machine-driven application would rather consume machine-readable representations from the same service. Data formats must be hypertext such that humans as well as autonomous processes are able to traverse hyperlinks in order to deduce the next state transition and action. Moreover, control data is used to define the semantic of the response by specifying—amongst others—resource states, error conditions and cache behaviour.

As being an architectural style, REST does not rely on any particular technology stack. REST does especially not depend on HTTP and URI as taking for granted in many publications. Still, HTTP and URI are the most common and widely used technologies for implementing REST-based architectures. HTTP is stateless in nature and it supports metadata (headers) to define various control and status information. It is intended to transfer resource representations marked up by MIME types. HTTP includes status codes to describe the semantic of a response and uses URIs to identify resources as well as a set of methods to determine actions, which makes up the unified interface.

Unfortunately, REST suffers from poor standardisation. Consequently, REST services are largely misunderstood and often assigned to simple web applications. This is for instance reflected by RMM (Richardson maturity model) [4], which

defines four levels to categorise the "REST-fullness" of web services and applications. Due to this lacking clarity of the term and its meaning it is also questionable whether the maturity of REST has reached an acceptable level for being SOA-ready.

3 SOA-readiness of REST

As depicted in Figure 1 a SOA consist of three parties. The service provider offers its services $\{S_1, ..., S_n\}$ to service consumers. To discover offered services a service registry maintains a database of services' descriptions. The providers attached to a particular service registry publish their services by supplying a so-called service contract, which contains all necessary information on the functionality of that service, its interface and other constraints demanded from a service consumer in order to make use of the service.

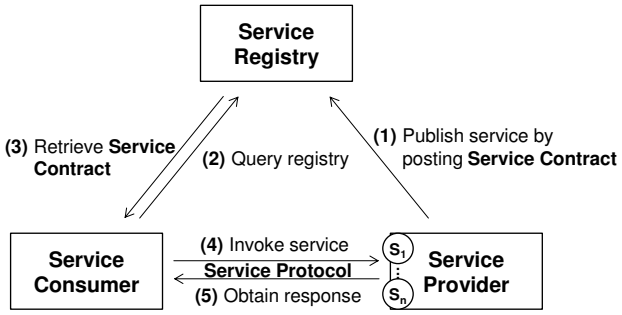

Fig. 1. Components of a SOA

Besides the discoverability of services, the service registry in conjunction with the service contract plays an important role in enabling the loose-coupling of service consumers and services. This means that service consumers are able to discover required services during runtime and to invoke such services without the need of adapting and recompiling the consumer code manually. This is accomplished by automatic code generation capabilities which rely on the rules and definitions specified in the service contract to generate the required code. Enriched with a platform-independent service protocol, the whole concept allows for the loose-coupling of even heterogeneous systems running on distinct platforms.

3.1 SOAP-implemented SOA

A SOA is described in a technology-independent manner and henceforth can be implemented with any suitable set of technologies as long as the mentioned

properties can be fulfilled. The most commonly used technology set is the service stack based on SOAP, where SOAP takes the role of the platform-independent service protocol (see Figure 2). The platform-independence of SOAP is achieved by it being based on XML. Due to this prerequisite, the stack further consists of other XML applications including WSDL (web service definition language) [5] as the standard for specifying a service contract and UDDI (universal description, discovery and integration) [6] as the standard for implementing the service registry.

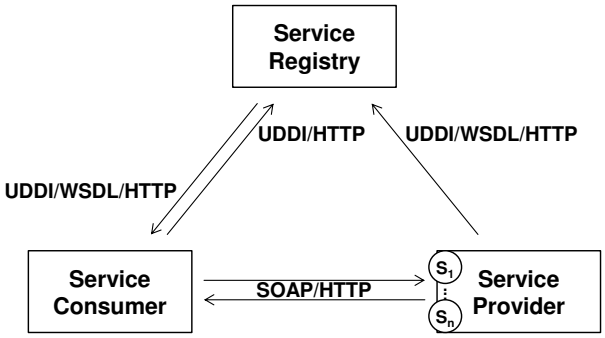

Fig. 2. SOAP-based SOA implementation

Thus, an SOA-based system implemented on SOAP and its accompanying standards contains all the properties defined in the SOA concept. Note, that the main protocol for transferring the service protocol messages is most commonly HTTP. This raises the question whether SOAP-based services can already be considered as REST-ful. In fact, most of the REST principles are also met by SOAP-based services, including the client-server, stateless, cache and layered system constraints. Still, the important constraint of a uniform interface cannot be fulfilled. This lies in the fact that SOAP-based services are operation-oriented which means that the interface is unique for each operation. REST instead is resource-oriented and defines the same four verbs for operating on each resource. The question arises, how a REST-implemented SOA then looks like and whether REST and its available implementation choices are already mature and comprehensive enough to implement SOA-based systems.

3.2 REST-implemented SOA

To ground a SOA on the REST architecture style, according standards and technologies are required for implementing the various SOA components. As further demand, these standards and technologies need to adhere to the specifics of REST. Figure 3 illustrates a common instantiation of the SOA infrastructure when implementing REST on HTTP.

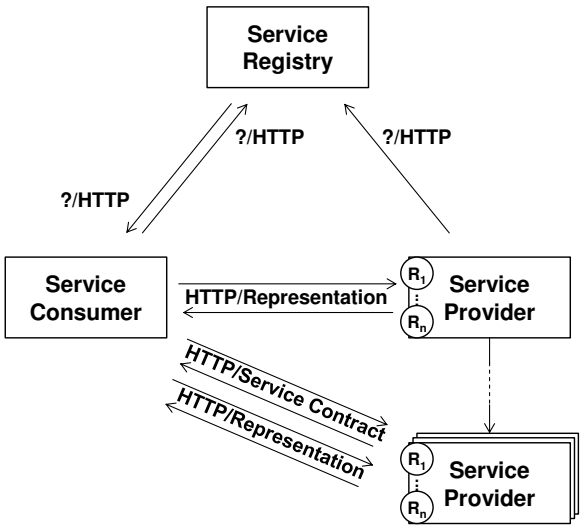

Fig. 3. REST-based SOA implementation

Instead of making solely use of a centralized service registry to discover services with REST this SOA principle can be extended to support decentralized discovery approaches, since REST services have the ability to refer to other services by providing hyperlinks [7]. Hence, a client is able to start at an initial (registry) service and henceforth discover a decentralized network of service referrals in order to find the looked for service. Figure 3 makes apparent, that for some of the SOA parts no standardised technological deployment choice exist yet. Especially the service registry and the service contract miss relevant standards and technologies in the REST universe, although a multitude of proposals exist to describe a REST service in a machine-readable manner [8–11]. Still, these approaches are yet in their infancy or not in conformance with the REST constraints, so that further work is required in order to distil the best fitting approach which may finally lead to a candidate for standardisation.

4 Empirical Analysis of REST Frameworks

In order to understand how the aforementioned shortcomings and especially the missing standardisation impact on concrete REST service development frameworks, an empirical analysis has been developed and conducted focusing on popular frameworks in distinct server-side programming environments, including Jersey (Java/JAX-RS), Play Framework (Java), ASP.NET (C#), restify (Javascript/Node.js), Ruby on Rails (Ruby) and Laravel (PHP). The goal of the study is to investigate on the processing of correct and intentionally flawed requests amongst the frameworks.

4.1 Test Application

To put all frameworks into a homogeneous test environment, the same application has been developed with all of them, which consists of a simple service that allows maintaining a to-do list. Entries to the to-do list can be created, read, updated and deleted to manage tasks. In addition, the service also offers information about the available actions by invoking the OPTIONS method. The data model is a simple list of tasks with two properties: a string describing the task and a boolean that denotes whether a task is still open. A client is able to create a to-do item via the POST method by sending a task name either in JSON format, in form of an XML or as a query string. The *isComplete* property is initially set to false by default, if a new task has been added. Reading to-dos can be performed by issuing GET or HEAD requests. With the accept header in a GET and HEAD request, clients can choose the content type of the requested resource. The PUT and PATCH methods are intended to update to-dos. Also the entity body in update requests can vary between different resource representations. For removing a to-do, the service specifies the DELETE method. The complete REST-API of the test service is as follows, given in URI Template notation [12]:

```
OPTIONS /todos
OPTIONS /todos/{id}
POST    /todos
HEAD    /todos
HEAD    /todos/{id}
GET     /todos
GET     /todos/{id}
PUT     /todos/{id}
PATCH   /todos/{id}
DELETE  /todos/{id}
```

4.2 Service Invocations

The test scenario assumes that the service provider and service consumer know each other and that no discovery step is required. The client is furthermore aware of the resource path and the data schema as introduced in the previous section. These prerequisites stem mainly from the fact that none of the considered frameworks include any form of service discovery or service exploration.

The analysis methodology is implemented in a specific client that executes the following test flow. First, the client adds a new task to the to-do list through the POST method. Afterwards, it checks the availability of a certain to-do item via the HEAD method. By usage of OPTIONS, the client gathers further information about available actions. Based on the gathered knowledge obtained from the OPTIONS method and HEAD request the client inquires to-dos via a sequence of GET requests. In order to mark an open task as completed, the PUT and PATCH methods are used. Once a task is completed the client takes it off the to-do list by invoking the DELETE action.

The whole test scenario consists of multiple test cases which can be grouped by the HTTP methods and one non-existing method EVIL as defined hereafter. Each test of a corresponding method owns a set of test cases denoted with a unique ID that starts with the first two characters of the method and a sequence number. The ID serves as a reference for the test case. The whole analysis encompasses the following test cases:

POST Test Cases
PO.1 Regular creation: This action adds a resource via a supported data format (here XML and JSON) with semantically and syntactically correct content.
PO.2 Unsupported content type: This test creates a resource represented by an unsupported media type.
PO.3 Content type and payload mismatch: This test issues a request containing a mismatch between the specified content type and the actual data format of the payload, such as a request containing, for instance, a content type header with the value *application/json* while the body is XML-formatted. In further test variations the content type is completely absent.
PO.4 Flawed content length: This test case observes the service behaviour in case the value of the content length header differs from the actual size of the body. Test variations assign the content length value a alphanumerical string or remove the header entirely.
PO.5 Flawed resource identifier: This test examines the service response on wrong URIs including e.g. typing errors and URIs including IDs.
PO.6 Malformed data: Requests generated within this test class contain malformed payload data in their bodies such as a missing quotation mark in an JSON object or an absent greater-than sign in an XML document.
PO.7 Unprocessible content: The requests issued by this test category contain flawless headers and well formed data in the body, but encloses an unprocessible data schema that is not in conformance with the service's data model.
PO.8 Unknown protocol version: This test case invokes the service with a not existing HTTP version specified in the request line.

HEAD and GET Test Cases
HE.1/GE.1 Regular reading: These test runs perform regular GET and HEAD requests with supported media types (here XML and JSON) specified within the accept header.
HE.2/GE.2 Unsupported media types: Requests unsupported resource representations form the service.
HE.3/GE.3 Flawed resource identifier: This test checks invocations on misspelled URIs and non-existing resources.
HE.4/GE.4 Containing content: This test issues HEAD and GET requests that contain content type headers and corresponding bodies which is actually not in conformance with the HTTP standard.
HE.5/GE.5 Missing accept header: This test constructs requests without an accept header.

HE.6/GE.6 Unknown protocol version: This test case invokes the service with a not existing HTTP version specified in the request line.

OPTIONS Test Cases
OP.1 Ping *: The HTTP specification defines a ping functionality by including an asterisk (*) to the URI of an OPTIONS request. This test figures out whether this feature is supported.
OP.2 Regular invocation: The requests in this test case invokes regular OPTIONS actions.
OP.3 Supported media types in accept header: The goal of this test is to execute regular OPTIONS requests including supported media types (here XML and JSON) in the accept header.
OP.4 Unsupported media types in accept header: This test issues requests with unsupported media types specified in the accept header.
OP.5 Flawed resource identifier: Requests generated by this test class contain mis-spelled or missing resource identifiers.
OP.6 Containing content: The requests issued by this tests contain entity bodies, although the HTTP specification does not define any usage of payloads inside OPTIONS requests.
OP.7 Unknown protocol version: This test case invokes the service with a not existing HTTP version specified in the request line.

PUT and PATCH Test Cases
PU.1/PA.1 Regular update: This test executes regular updates via the PATCH and PUT methods including supported content types (here XML and JSON) with semantically and syntactically correct content.
PU.2/PA.2 Unsupported content type: This test sends resource updates with unsupported resource representations.
PU.3 Partial update / PA.3 Complete update: This test issues partial updates via the PUT method and entire updates via PATCH with supported media types which are also semantically and syntactically correct.
PU.4/PA.4 Content-Type and payload mismatch: Requests generated by this test class contain a mismatch between the specified content type and the actual payload format.
PU.5/PA.5 Flawed content length: Intentional corruption of the length header value.
PU.6/PA.6 Flawed resource identifier: These tests perform updates specifying mis-spelled and non-existing URIs.
PU.7/PA.7 Malformed data: This test modifies resource representations so that they become syntactical and semantical incorrect.
PU.8/PA.8 Unprocessible content: Requests that contain well-formed and supported data, but it is not in conformance with the service's data schema.
PU.9/PA.9 Unknown protocol version: This test case invokes the service with a not existing HTTP version specified in the request line.

DELETE Test Cases
DE.1 Regular deletion: This test launches a deletion process of a single resource.
DE.2 Non-erasable resource: This test deletes a resource which must not be deleted according the business logic (here an uncompleted to-do task).
DE.3 All resources: The intention of this test run is to remove all the resources in one single request.
DE.4 Flawed resource identifier: Requests issued by this test class include mis-spelled and on non-existing URIs.
DE.5 Containing content: The DELETE request generated by this test case contains an entity body and a corresponding content type header.
DE.6 Unknown protocol version: This test case invokes the service with a not existing HTTP version specified in the request line.

EVIL Test Cases
EV.1 Regular accept header: This scenario tests a non-existing EVIL request with supported media types (here XML and JSON) in the accept header.
EV.2 Unsupported media types: This test generates EVIL requests with unsupported resource representations specified in in the accept header.
EV.3 Flawed resource identifier: The EVIL requests contain mis-spelled or non-existing URIs.
EV.4 Containing content: This test generates an EVIL request with a supported content type and a well-formed entity body.
EV.5 Unknown protocol version: This test case invokes the service with a not existing HTTP version specified in the request line.

5 Results

The results gathered from the empirical studies with six distinct REST frameworks based on the methodology introduced in the previous section are manifold. In general, they underline the need for a more stringent standardization of technologies suited to enrich REST-based services with service properties known from the SOA domain. By analysing the REST frameworks it became visible, that the missing standardisation leads to quite diverse implementations, causing incompatibilities, especially between system components deployed on heterogeneous platforms. This in turn means that such incompatibilities need to be resolved manually, which requires the involvement of an experience REST developer. In complex settings this might even not be feasible at all, yielding to constraints in respect to implementation choices.

The test runs show that when implementing the application scenario introduced in Section 4.1 with the examined frameworks, the main discrepancies appear in the usage of HTTP headers and the contained meta data therein as well as the status codes selected to signal the request processing state back to the service consumer (see Figure 4). As can be observed, the received status codes

differ amongst the frameworks for identical requests. This hinders the implementation of automatic code generation tools and loose coupling, due to the absence of an uniform behaviour. Especially a state transition based on the control data provided with the status codes is not feasible in an automatic manner across heterogeneous platforms.

An interesting and at the same time important result is the fact that the frameworks even when returning identical response codes sometime still are very different in other respect. One example can be noted in ASP.NET when a resource is created. The response code 201 created is returned as expected, but in addition to the URI of the freshly generated resource contained in the location header, the response carries the created object as its payload as well. This does not influence on the focussed SOA-readiness, but might raise other issues in terms of data volume to be transmitted.

Some of the tests even brought vulnerabilities to light which might get exploited in DoS (denial of service) attacks. The requests for which no responses have been received from the service are the potential attack targets. For the ones listed in Figure 4 the communication channel remains open. Thus, an attacker can occupy and block networking resource easily, by issuing such requests.

6 Conclusions and Outlook

REST is an established architecture style for designing distributed systems based on services and offers in conjunction with HTTP a wide-spread and well-understood implementation ground. It is less comprehensive as the SOA concept, though. Crucial service properties such as the discoverability and loose coupling are lacking for REST, including the related components and artefacts such as a service registry and a service contract. Still, such properties are required in order to enable the ad hoc usage of services during service consumer runtime. In this paper the need for such technologies has been motivated and initial contributions on REST-ful embodiments of a service registry and a service contract have been introduced. Still, there is a bunch of research and development challenge to solve in order to enrich REST with SOA service properties. According standardisation efforts need to be initiated and aligned with these developments as well.

The lack of standardisation in the REST domain got also apparent in another more elementary context. When conducting the study of REST frameworks one observation has been that the distinct frameworks construct and process service requests and responses very differently. This reveals that currently a heterogeneous system environment needs to be manually programmed and deployed by experienced REST developers in order to assimilate the system components. A stringent specification and mapping of the REST ingredients to HTTP is henceforth required, to lay the ground for a higher compatibility of REST implementations with less manual code interventions which finally forms the required foundation for automated code generation and the loose-coupling of REST ser-

vices. Also, service security approaches require a REST/HTTP specification as a stable and reliable ground [13].

Acknowledgment

This work has been funded by the European Union within the European Regional Development Fund program.

References

1. T. Erl, *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2007.
2. M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon, "SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)," W3C Recommendation, W3C, 2007.
3. R. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.
4. E. Wilde and C. Pautasso, eds., *REST: From Research to Practice*. Springer, 2011.
5. R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana, "Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language," W3C Recommendation, W3C, 2007.
6. L. Clement, A. Hately, C. von Riegen, and T. Rogers, "UDDI Version 3.0.2." Organization for the Advancement of Structured Information Standards, UDDI Spec Technical Committee Draft, 2004.
7. C. Pautasso and E. Wilde, "Why is the web loosely coupled?: A multi-faceted metric for service design," in *Proceedings of the 18th International Conference on World Wide Web (WWW)*, ACM, 2009.
8. M. Amundsen, "Hold Your Nose vs. Follow Your Nose, Observations on the state of service description on the Web," in *5th International Workshop on Web APIs and RESTful Design (WS-REST)*, 2014.
9. MuleSoft, Inc, "RAML Version 0.8: RESTful API Modeling Language," tech. rep., 2013.
10. M. Bennara, M. Mrissa, and Y. Amghar, "An Approach for Composing RESTful Linked Services on the Web," in *5th International Workshop on Web APIs and RESTful Design (WS-REST)*, 2014.
11. R. Verborgh, T. Steiner, D. Van Deursen, J. De Roo, R. Van de Walle, and J. Gabarró Vallés, "Description and Interaction of RESTful Services for Automatic Discovery and Execution," in *Proceedings of the FTRA 2011 International Workshop on Advanced Future Multimedia Services (AFMS)*, 2011.
12. J. Gregorio, R. Fielding, M. Hadley, M. Nottingham, and D. Orchard, "URI Template," RFC 6570, IETF, 2012.
13. P. Gorski, L. Lo Iacono, H. V. Nguyen, and D. B. Torkian, "Service Security Revisited," in *11th IEEE International Conference on Services Computing (SCC)*, 2014.

	Consolidated View	PHP	Play	Node	Jersey	RoR	ASP
POST							
PO.1 application/x-www-form-urlencoded	**201**	201	201	201	201	400	No Response
PO.1 application/json	**201**	201	201	201	201	406	201
PO.1 application/xml	**201**	201	201	201	201	406	400
PO.2 Unsupported content type	**415**	415	415	415	415	400	500
PO.3 Content type and payload mismatch	**400**	500	400	400	400	400	400
PO.3 No content type but with payload	**400**	415	500	415	500	400	500
PO.4 Wrong content length	**400**	No Response	No Response	No Response	No Response	No Response	No Response
PO.4 Content length as String	**400**	No Response	No Response	No Response	400	400	400
PO.4 No content length	**411**	No Response	400	No Response	400	411	411
PO.5 Wrong action on resource	**405**	404	404	405	405	404	400
PO.5 Wrong resource identifier	**404**	404	404	404	404	404	404
PO.6 Malformed XML	**400**	500	400	400	400	400	400
PO.6 Malformed JSON	**400**	500	400	400	400	400	400
PO.7 Wellformed JSON, unprocessible content	**400**	500	400	400	400	400	400
PO.7 Wellformed XML, unprocessible content	**400**	500	400	400	400	400	400
PO.8 Unknown protocol version	**505**	201	201	201	505	406	201
HEAD							
HE.1 application/json	**200**	200	200	200	200	200	405
HE.1 application/xml	**200**	200	200	200	200	200	405
HE.2 Unsupported media type	**406**	200	406	406	500	406	405
HE.3 Wrong resource identifier	**404**	404	404	404	404	404	404
HE.3 Not exsiting resource	**404**	200	400	404	404	404	405
HE.4 Containing content	**400**	200	200	200	200	400	405
HE.5 No accept header	**200**	200	200	200	200	200	405
HE.6 Unknown protocol version	**505**	200	200	200	505	200	405
OPTIONS							
OP.1 Ping *	**200**	200	400	404	200	404	400
OP.2 Regular	**200**	200	200	405	200	404	405
OP.2 Regular with resource id	**200**	200	200	405	200	404	405
OP.3 application/json	**200**	200	200	405	200	404	405
OP.3 application/xml	**200**	200	200	405	200	404	405
OP.4 Unsupported media type in accept header	**406**	200	200	406	200	404	405
OP.5 Wrong resource identifier	**404**	404	404	404	404	404	404
OP.5 Not existing resource	**404**	200	400	405	200	404	405
OP.6 Containing content	**400**	200	400	405	200	400	405
OP.7 Unknown protocol version	**505**	200	200	405	505	404	405
GET							
GE.1 application/json	**200**	200	200	200	200	200	200
GE.1 application/xml	**200**	200	200	200	200	200	200
GE.2 Unsupported media type	**406**	200	415	406	500	406	200
GE.3 Wrong resource identifier	**404**	404	404	404	404	404	404
GE.3 Not existing resource	**404**	200	400	404	404	404	200
GE.4 Containing content	**400**	200	200	200	200	400	200
GE.5 No accept header	**200**	200	200	No Response	200	200	200
GE.6 Unknown protocol version	**505**	200	200	No Response	505	200	200
PUT							
PU.1 application/x-www-form-urlencoded	**204**	204	204	204	204	400	500
PU.1 application/json	**204**	200	204	204	204	204	500
PU.1 application/xml	**204**	204	204	204	415	204	500
PU.2 Unsupported content type	**415**	415	415	415	415	400	500
PU.3 Partial update	**400**	500	204	204	406	204	500
PU.4 Content type and payload mismatch	**400**	500	400	400	400	400	500
PU.4 No content type but with payload	**400**	415	500	415	415	400	500
PU.5 Wrong content length	**400**	No Response	No Response	No Response	No Response	No Response	No Response
PU.5 Content length as String	**400**	No Response	No Response	No Response	400	400	400
PU.5 No content length	**411**	No Response	400	No Response	400	411	411
PU.6 Wrong action on resource	**404**	404	404	404	404	404	404
PU.6 Not existing resource	**404**	500	400	500	415	404	500
PU.7 Malformed XML	**400**	500	400	400	415	400	500
PU.7 Malformed XML isComplete=EVIL	**400**	500	400	500	415	404	500
PU.7 Malformed JSON	**400**	500	400	400	400	400	500
PU.7 Malformed JSON isComplete=EVIL	**400**	500	400	400	400	400	500
PU.8 Wellformed JSON, unprocessible content	**400**	500	400	400	400	400	500
PU.8 Wellformed XML, unprocessible content	**400**	500	400	400	415	400	500
PU.9 Unknown protocol version	**505**	500	400	500	505	404	500
PATCH							
PA.1 application/x-www-form-urlencoded	**204**	500	204	204	204	400	204
PA.1 application/json	**204**	500	204	204	204	204	204
PA.1 application/xml	**204**	204	204	204	415	400	500
PA.2 Unsupported content type	**415**	415	415	415	415	400	500
PA.3 Complete update	**204**	200	204	204	204	204	204
PA.4 Content type mismatch and payload	**400**	500	400	400	400	400	500
PA.4 No content type but with payload	**400**	415	500	415	415	400	500
PA.5 Wrong content length	**400**	No Response	No Response	No Response	No Response	No Response	No Response
PA.5 Content length as String	**400**	No Response	No Response	No Response	400	400	400
PA.5 No content length	**411**	No Response	400	No Response	400	400	500
PA.6 Wrong action on resource	**404**	404	404	404	404	404	404
PA.6 Not existing resource	**404**	500	400	404	415	404	500
PA.7 Malformed XML	**400**	500	400	400	415	400	500
PA.7 Malformed XML isComplete=Evil	**400**	500	400	404	415	404	500
PA.7 Malformed JSON	**400**	500	400	400	400	400	204
PA.7 Malformed JSON isComplete=Evil	**400**	500	400	404	406	404	204
PA.8 Wellformed JSON, unprocessible content	**400**	500	400	404	404	404	204
PA.8 Wellformed XML, unprocessible content	**400**	500	400	404	415	404	500
PA.9 Unknown protocol version	**505**	500	400	404	505	404	204
DELETE							
DE.1 Regular	**204**	204	204	204	204	204	204
DE.2 Regular isComplete=false	**403**	204	403	403	403	204	403
DE.3 all	**405**	404	404	405	405	404	404
DE.4 Not existing resource	**404**	500	400	404	404	404	500
DE.5 Containing content	**400**	204	204	204	204	204	403
DE.6 Unknown protocol version	**505**	204	204	204	505	204	403
EVIL							
EV.1 application/json	**501**	501	404	No Response	405	500	404
EV.1 application/xml	**501**	501	404	No Response	405	500	404
EV.2 Unsupported media type	**501**	501	404	No Response	405	500	404
EV.3 Wrong resource identifier	**501**	501	404	No Response	404	500	404
EV.4 Containing content	**501**	501	404	No Response	405	500	404
EV.5 Unknown protocol version	**501**	501	404	No Response	505	500	404

Fig. 4. Status codes received by the frameworks for the test cases