Towards Conformance Testing of REST-based Web Services

Luigi Lo Iacono and Hoai Viet Nguyen
Cologne University of Applied Sciences, Cologne, Germany

Keywords: REST, HTTP, URI, Conformance Testing, Web Services, SOA, Cloud.

Abstract: Despite the lack of standardisation for building REST-ful HTTP applications, the deployment of REST-based Web Services has attracted an increased interest. This gap causes, however, an ambiguous interpretation of REST and induces the design and implementation of REST-based systems following proprietary approaches instead of clear and agreed upon definitions. Issues arising from these shortcomings have an influence on service properties such as the loose coupling of REST-based services via a unitary service contract and the automatic generation of code. To overcome such limitations, at least two prerequisites are required: the availability of specifications for implementing REST-based services and auxiliaries for auditing the compliance of those services with such specifications.
This paper introduces an approach for conformance testing of REST-based Web Services. This appears conflicting at the first glance, since there are no specifications available for implementing REST by, e.g., the prevalent technology set HTTP/URI to test against. Still, by providing a conformance test tool and leaning it on the current practice, the exploration of service properties is enabled. Moreover, the real demand for standardisation gets explorable by such an approach. First investigations conducted with the developed conformance test system targeting major Cloud-based storage services expose inconsistencies in many respects which emphasizes the necessity for further research and standardisation.

1 INTRODUCTION

Building distributed systems based on the abstraction of software services has been and still is an important paradigm. First dominated by SOAP (Simple Object Access Protocol) (Gudgin et al., 2007) and its accompanying technology stack, the architectural style REST (REpresentational State Transfer) (Fielding, 2000) has gained traction as an alternative approach for designing service systems. REST targets the scalability of application interaction, the uniformity of interfaces and the independent evolvement of components and intermediates with the intention to reduce latency, enable security and provide long-lived services.

Besides the availability of a set of constraints defined in the dissertation of Fielding, there still does neither exist a definition nor a clear understanding of how to apply this architectural style to technical instantiations. As a consequence, REST is often enough mistaken as being a standard composed of its underlying foundations: URI (Berners-Lee et al., 2005) and HTTP (Fielding et al., 1999). The source for this diffuse view on the REST concept lies in the fact that the two aforementioned standards are currently the only notable technology choice for implementing REST-based service systems.

Issues arising from these shortcomings in standardisation have, e.g., an influence on the fusion of REST and SOA (Service-Oriented Architecture) (Gorski et al., 2014b). Founding a SOA-based system on a REST architecture is challenging due to missing service properties including the loose coupling of REST-based services. This is because of the lack of service contracts, which makes, moreover, the automatic generation of client-side code infeasible. In Cloud and Utility Computing this lack of standardisation causes cross-service incompatibilities promoting vendor lock-in and hindering the simultaneous adoption of multiple distinct Clouds in a user-intended manner.

Approaching these issues requires at least two determining factors: (1) the availability of specifications for applying the REST principles to technical foundations (in a SOA fashion) and (2) conformance testing instruments for auditing the compliance of REST-based services to these technical environments. This paper approaches these gaps by introducing a mean for the conformance testing of REST-based Web Services focussing on the dominating implementation

Lo Iacono L. and Nguyen H..
Towards Conformance Testing of REST-based Web Services.
DOI: 10.5220/0005412202170227
In *Proceedings of the 11th International Conference on Web Information Systems and Technologies* (WEBIST-2015), pages 217-227
ISBN: 978-989-758-106-9

base: HTTP/URI. This appears conflicting at the first sight, since there are no specifications available yet to test against. Still, by providing a conformance test tool and leaning it on the state of the art, the exploration of service properties such as the uniformness of requests and responses is enabled. In practice, however, a real demand for standardisation might actually not be present. With the proposed conformance test methodology and tooling this matter can be explored empirically. To prove the viability of this approach REST services offered by major Cloud storage providers are analysed. Other available research in this domain provides equivalent insights while focussing on REST frameworks instead of deployed real world services (Gorski et al., 2014b). Note, that the present topic has not been an active research area yet and that due to the specific nature of REST there is not much proper related work available so far. According studies in other Web Services technology stacks such as SOAP, e.g., do not share fundamental service properties including the Uniform Interface of REST-based services. Moreover, the available work in the SOAP-based Web Services arena is focussing on SOAP specifics such as the interoperable code generation from an WSDL service contract (Elia et al., 2014). Henceforth, an adoption and consideration in the REST domain is not feasible.

The rest of the paper is organised as follows. In Section 2 a specification for mapping the REST architectural principles and constraints to HTTP/URI is deduced from the current practice and related work. Following up in Section 3, a test methodology is introduced, which provides a framework for conformance testing of REST-based services. Available test tools are then analysed according to their capabilities of performing the required tests given in Section 4. The analysis of the state of the art reveals a lack of an adequate REST test tooling. This results in the development of an own approach which is described in Section 5. First findings obtained from investigations performed with the introduced methodology and the developed tooling against four major Cloud-based storage services, expose inconsistencies and heterogeneities in many respects. This is a violation of the Uniform Interface constraint. More details about these issues are discussed in Section 6. The paper concludes in Section 7 with a summary of the main contributions and findings as well as an outlook on future work.

2 SPECIFICATION OF REST-BASED WEB SERVICES

To test a REST-based service according to its conformance to a particular standard, such a standard needs to be present in the first place. Regrettably, this is not the case for the mapping of the REST architectural components and principles to any particular set of technologies. Thus, such a mapping specification needs to be derived in order to lay the ground for the goals of this paper. Based on the least common denominator obtained from the current practice and recent empirical investigations (Gorski et al., 2014b), the specification for this paper is defined as an instantiation of the REST principles based on the URI specification and HTTP while taking additional clarifications of HTTP semantics and contents (Fielding and Reschke, 2014) into account (see Table 1). URIs are denoted in the URI Template (Gregorio et al., 2012) notation. For the sake of readability but without the loss of generality the following specifications will focus on the core request and response components only and will leave aspects such as caching, versioning, security, streaming and response header fields for error treatment out of consideration.

A POST request creates a resource on the service side. The URI in the request denotes the set in which a resource is to be added. In order to do so, a complete representation of such a resource needs to be sent in the request payload. This requires according meta data in the request header to signal the type and length of the contained resource representation. In case the processing of a POST request has been completed successfully, the response contains the status code 201 (Created), meaning that the resource has been created. Such a response must contain a Location header in addition, which provides the identifier of the created resource in form of an URI. The response payload is empty. To notify the client on special conditions regarding the request processing, several other status codes are defined. The status code 202 (Accepted) must be used, when the request itself can be processed successfully but the creation of the resource is so time-consuming that the response needs to be returned by the service before finalizing the resource instantiation. The other status codes have all to do with issues in the request. If the addressed resource set does not exist, 404 (Not Found) is returned. In case the specified content type is not supported, 415 (Unsupported Media Type) is passed back. The status code 411 (Length Required) is used, if the content length is missing. When the POST method is not defined for the resource set but is invoked anyhow, then the 405 (Method Not Allowed) status code is re-

Table 1: Specification of REST to HTTP/URI mapping.

HTTP Method	Resource Identifier	Request Headers	Request Payload	Response Codes	Response Headers	Response Payload
POST	/{resources}	Content-Type Content-Length	Resource representation (complete)	201 Created 202 Accepted 400 Bad Request 404 Not Found 405 Method Not Allowed 411 Length Required 413 Payload Too Large 415 Unsupported Media Type 501 Not Implemented 505 HTTP Version Not Supported	Location	Empty
GET	/{resources} /{resources}/{id}	Accept	Empty	200 OK 400 Bad Request 404 Not Found 405 Method Not Allowed 406 Not Acceptable 501 Not Implemented 505 HTTP Version Not Supported	Content-Type Content-Length Link	Resource representation (complete)
PUT	/{resources}/{id}	Content-Type Content-Length	Resource representation (complete)	201 Created 202 Accepted 204 No Content 400 Bad Request 404 Not Found 405 Method Not Allowed 411 Length Required 413 Payload Too Large 415 Unsupported Media Type 501 Not Implemented 505 HTTP Version Not Supported	None	Empty
PATCH	/{resources}/{id}	Content-Type Content-Length	Resource representation (partial)	202 Accepted 204 No Content 400 Bad Request 404 Not Found 405 Method Not Allowed 411 Length Required 413 Payload Too Large 415 Unsupported Media Type 501 Not Implemented 505 HTTP Version Not Supported	None	Empty
DELETE	/{resources}/{id}	None	Empty	202 Accepted 204 No Content 400 Bad Request 404 Not Found 405 Method Not Allowed 409 Conflict 501 Not Implemented 505 HTTP Version Not Supported	None	Empty
OPTIONS	/* /{resources} /{resources}/{id}	None	Empty	204 No Content 400 Bad Request 404 Not Found 405 Method Not Allowed 501 Not Implemented 505 HTTP Version Not Supported	Allow	Empty
HEAD	/{resources} /{resources}/{id}	Accept	Empty	200 OK 400 Bad Request 404 Not Found 405 Method Not Allowed 406 Not Acceptable 501 Not Implemented 505 HTTP Version Not Supported	Content-Type Content-Length Link	Empty

turned. With the status code 413 (Payload Too Large) the service signals that an oversized request payload has been received. This is, e.g., the case when the request message is larger than the capabilities available to the service. Any other syntactical or semantical error in the request is repulsed by the status code 400 (Bad Request).

A GET request accesses a particular resource in a certain representation. The URIs for accessing resources differ in regards to whether a single instance or all instances of a resource are to be accessed. To declare the desired resource representations, the request must carry an Accept header. The payload of the request is empty, since no data is transferred to the service. If the resource is available in the requested representation, the service delivers it to the client in the response payload and sets the status code 200 (OK) for the according response. The content type and length are also stated in the response, but these entries are contained in the response header. Furthermore, a GET response can contain a Link header referencing another resource providing additional meta data of the requested resource (Nottingham, 2010). To signal error conditions the response contains the status code 404 (Not Found) in case the requested resource does not exist, 406 (Not Acceptable) if the requested resource representation is not provided, 405 (Method Not Allowed) when the GET method is not defined for the resource and 400 (Bad Request) in cases in which any other syntactical or semantical er-

ror is contained in the request.

A PUT request updates a particular resource. In case the addressed resource does not exist and the service allows for resource creation via PUT, it will be created. Hence, in some circumstances a PUT request can act like a creation request. A PUT always transfers a complete resource representation with the request to the service. The resource identifier addresses a concrete resource. In case the resource exists it is updated by the provided payload and the according response code is set to 204 (No Content). In case a PUT operation addresses a non-existing resource identifier, the resource enclosed in the request will be created and the server must return the response code 201 (Created). If the service only grants the PUT method for update operations and does not offer the functionality of resource creation, it must response with the code 405 (Method Not Allowed). In all other cases and especially the possible error conditions the behaviour corresponds to the one of the POST method.

A PATCH (Dusseault and Snell, 2010) request updates a given resource virtually as a PUT request. In contrast to PUT a PATCH request, however, specifies only those data fields that need to be changed and does henceforth not provide the complete resource representation with the request payload. Thus, with a PATCH request a resource cannot be created but efficiently updated, since only the required changes need to be send to the service. PATCH responses follow the same status code policies as defined for PUT, leaving the codes related to resource creation apart.

A DELETE request erases a particular resource from the service side. No further meta data is required in the request and the response headers. The status code 204 (No Content) expresses the successful deletion of the targeted resource specified in the corresponding request. If the addressed resource does not exist, this is signalled back to the client by the status code 404 (Not Found). In case the DELETE request is issued although it is not implemented, the status code 405 (Method Not Allowed) is returned. In some circumstances the erasing of a resource is not allowed, due to reasons lying in the business logic and the current resource state. In such a case, the service must return the status code 409 (Conflict). As for any other request, syntactical and semantical errors are denoted by the status code 400 (Bad Request).

An OPTIONS request is the basis for gathering information on the provided methods regarding a particular resource or set of resources. The request is addressed towards a resource identifier with no particular header and an empty payload. The corresponding response discloses the offered actions in the Allow header. The occurring error conditions are treated and reported by the response codes as discussed for the other request types.

A HEAD request is used to retrieve status information and meta data for a certain resource. It behaves exactly as the GET method with the only difference that the response payload is empty and does henceforth not contain a resource representation.

If a client performs a request with an unfamiliar feature such as an unknown method, the service must reply with the status code 501 (Not Implemented) informing the counterpart that this functionality is not implemented. Last but not least, requests containing an unsupported HTTP version number must be rejected by the status code 505 (HTTP Version Not Supported).

3 TEST METHODOLOGY

With this mapping specification and by utilising the REST constrains, an ideal conformance testing of REST-based services is an automatic task which requires very little preparation and input. The core of the proposed methodology is based on a catalogue of standard tests allowing for a structured audit of REST-based services (see Figure 1).

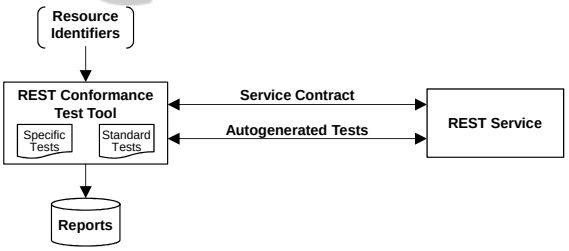

Figure 1: REST conformance test methodology.

As initial input, the conformance test tool requires the base resource identifier of the service to be tested only. By issuing a status request to this resource identifier, the service returns a response including the resource identifier referring to the service contract. In a REST architecture, a service contract is an information entity describing the Uniform Interface provided by the according resource. Basically, the Uniform Interface is set of instructions each composed of three main elements: the action (method), the resource identifier syntax and the supported resource representations (media types) (Erl et al., 2013). Others disclosures could also be delineations of non-functional meta data including authentication and authorization policies. By a subsequent read request the test tool retrieves the denoted service contract. The represen-

tation of the service contract can be of any format suitable to describe the data structure of the targeted resource, such as realised by RAML (RESTful API Modeling Language) (Sarid et al., 2014).

The retrieved service contract is then the entry point for generating a series of service invocations. Due to the interface uniformity, a standardised catalogue of test cases can be defined for REST-based services, containing a comprehensive combination of constructive and destructive tests, with which any REST-based service can be tested exhaustively. Such a standardised test catalogue for REST-based Web Services has been defined for the purpose of this paper, which is based on the specification laid in Section 2 and the test cases utilised in (Gorski et al., 2014b) for the empirical investigations of the SOA-readiness of REST frameworks for developing Web-based systems. The test cases of the standardised catalogue will not be described in detailed here, since they correspond with the 87 test cases presented in (Gorski et al., 2014b). The reader is henceforth referred to the original source for further reference. The ID scheme used for naming each test case in a unique manner—the first two letters of the HTTP verb followed by a dot and a sequence number—has been maintained in this paper in order to support cross-reading. If required, this standard set of tests can be accomplished by specific tests addressing unique aspects of a particular service environment and implementation. By this approach, the proposed conformance testing can be adopted to any REST-based Web Service equally straightforward. The marginal adoptions required for the empirical evaluation of major Cloud-based storage services are described in Section 6.

Each performed test run is logged and the results are stored in a report. The report entries contain the raw communication as well as a judgement on the conformity of the targeted service in respect to the defined specification. The conformance is verified by executing all tests and examining the obtained responses. For each of the 87 standard test requests a corresponding expected response has been defined. If a test request to a service returns a response which differs from the expected one, a candidate for a conformance breach has been found.

4 STATE OF THE ART

From the introduced REST conformance test approach the requirements for according tools got apparent and can henceforth be recorded as follows:

R1: A REST test tool must be able to manage a set of test cases for the uniform interface mapped to a certain service protocol—at the moment mainly HTTP.

R2: In order to be able to express a comprehensive set of constructive and destructive tests, many of the header entries including the request line, the Content-Type, the Content-Length and the Accept header need to be adjustable in a flexible manner.

R3: This is equally true for the resource representations. Here, various types of representations must be supported and modifiable according to the test cases' needs.

R4: A REST test client requires to perform the service inspection steps described in Section 3 to retrieve the service contract. The only required input for the test runs is a set of resource identifiers from which the test client constructs everything on its own.

Available REST test client software for HTTP/URI based instantiations such as Advanced REST Client (Psztyc, 2014), Postman (Postdot Technologies, 2014) and the REST testing features of SoapUI (SmartBear Software, 2014) do not fulfil all the demanded requirements. Such tools enable the creation and maintenance of multiple test requests, but lack in functionalities regarding the flexible adjustment of header fields, the request line, or resource representations and do not include any procedure to obtain the service contract automatically. Consequently, test cases implemented based on these tools need to be configured manually, which comes at a high cost due to the many distinct parameters to provide and set. Thus, for the purpose of this paper, an own development has been undertaken that fulfils the described set of requirements.

5 IMPLEMENTATION

The developed conformance test system is shown in Figure 2. The underlying idea is to contribute a software for managing conformance test projects targeting REST-based Web Services. To investigate the compliance of a service, a project needs to be created, first. Then a service auditor can launch test runs which execute all test cases of a defined catalogue. The tool itself is designed according to the architectural style REST. Via a REST-based API clients are able to create, read, update and delete test, project, run and case resources. A browser-based client implements this API and provides a user interface to create projects, set up and run tests and access the generated reports.

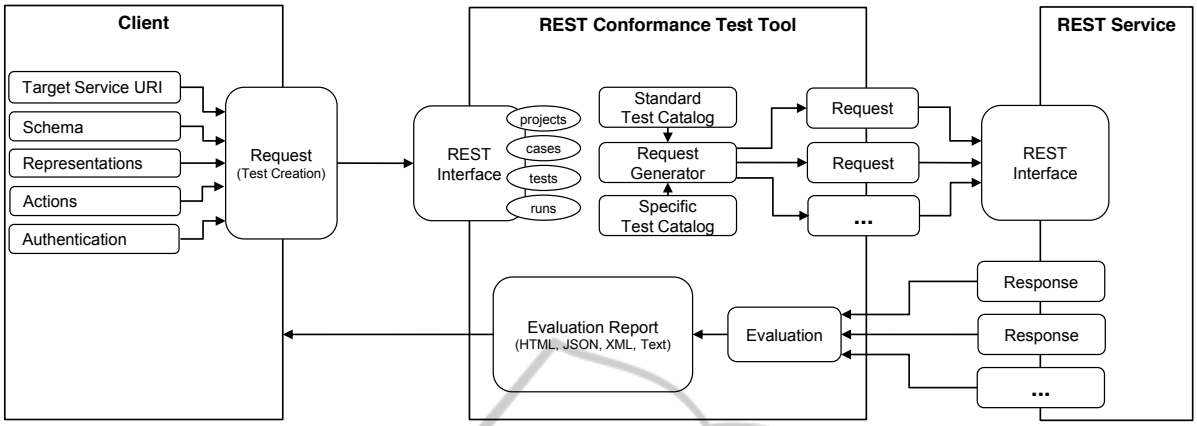

Figure 2: Architecture of the developed REST conformance test tool.

As introduced by the test methodology in Section 3, the test tool integrates a catalogue of standard tests at its core. Test projects can benefit from the standardised catalogue of tests, which can be adapted and performed on any interface specified by an according resource identifier. As mentioned in Section 3, the tests contained in the standard tests catalogue are based on the 87 tests defined in (Gorski et al., 2014b). Additional domain- and application-specific tests can be added as required. When executing a test suite, the test run is generated out of the standardised as well as the project specific tests. When a test run is completed, the client can retrieve an evaluation report in various resource representations such as in HTML, XML, plain text or JSON advising the service examiner about which test case has passed the conformance audit and which not.

The developed system fulfils all requirements except the one regarding the discovery and retrieval of the service contract (R4). An implementation of this feature is currently not feasible due to it being still a research and development topic (Verborgh et al., 2011; Bennara et al., 2014; Amundsen, 2014). The support in operational systems is henceforth lacking. Thus, the developed conformance test tool integrates mechanisms to manually enter information on the service contract as an interim work around. This includes the schema of the tested resources in distinct representation formats as well as the HTTP methods to test. The latter controls the amount of test cases contained in the various catalogues to be issued in a test run. As an additional set of parameters, the implemented test system enables to specify access credentials for the requests requiring authentication. This is especially true for the analysed Cloud storage services which deploy a custom HTTP authentication mechanism each. Besides the implementation of the requirements R1 and R3 the tool fulfils also the requirement R2, as it allows customizing any single header entry including the Content-Length header and the Host header as well as the request line. R2 is crucial requirement to enable destructive testing.

6 CONFORMANCE TESTING OF CLOUD-BASED STORAGE SERVICES

The intention of the present paper is to contribute a methodology for the conformance testing of REST-based services, although a required technical mapping of the REST principles to a concrete technology stack is lacking. Still, going the other way round enables to explore the effective need for such a standardisation. The introduced methodology together with the implemented test tool have been used to analyse various REST-based Web Services on their conformity with the specification given in Section 2, but also on their consistent understanding of REST and the uniform realisation of REST services. The investigations have been focused on Cloud-based file storage services including Amazon S3[1], Google Cloud Storage[2], Microsoft Azure Storage[3] and HP Helion Public Cloud Object Storage[4], also because they are understood to be particularly mature. The results obtained by exploring their REST-based service interfaces are discussed subsequently.

[1] https://aws.amazon.com/s3/
[2] https://cloud.google.com/products/cloud-storage/
[3] https://docs.hpcloud.com/object-storage/
[4] https://azure.microsoft.com/en-us/services/storage/

6.1 Service Tests

As one core foundation of the conformance test methodology, the standardised catalogue of test cases forms the common ground for evaluating the compliance of a particular REST-based Web Service in respect to the introduced specification. A series of constructive and destructive requests containing a combination of all sorts of valid and invalid request settings are used to observe the behaviour of a REST-based Web Service. The test cases are grouped by the HTTP methods GET, HEAD, POST, PUT, PATCH, DELETE and OPTIONS including one non-existing method named "EVIL". Required adaptations to the tests contained in the standard catalogue have been minimal. The test case "Non-erasable resource" (DE.2) in the DELETE request group has been disabled. This test tries to erase a resource which must not be deleted according to the business logic and the particular resource's state. As the explored Cloud-based file storage services do not provide an option to write-protect a file stored in the Cloud. The providers promote only an Access Control List (ACL) which allows defining write permission for a distinct set of users, but do not offer a feature to protect a resource against a general deletion. Thus, trying to remove a resource by a unauthorized user will result in a permission error and not a delete rejection due to business logic constraints. Thus, the intention of test case DE.2 is not being covered by the Cloud storage services, leading to it being left out.

The test case "Flawed content length" (PU.5) has been extended with one new test run, which checks the behaviour of the service in the presents of a discrepancy in the content length value being in this test run actually smaller than the de facto payload size. The goal is to observe how many bytes are stored by the service if any. A last test specific to storage services has been added, which tests the behaviour when exceeding the maximum allowed file size. This test has been denoted as "Content-Length exceeding the allowed payload size" (PU.10). It performs a file upload with a file size going beyond the limit set by the storage provider. No further tests have been identified as missing or specific to the underlying services.

6.2 Test Results

The obtained results from testing the four REST-based Cloud storage services with the introduced tool reveal that none of the services is in conformance with the specification introduced in Section 2. The broad range of inconsistencies amongst the services and between each service and the specification can be observed in many respects (see Table 2 in the Appendix).

Although all services provide an equivalent behaviour in case of the HEAD, GET, PUT and DELETE methods, where the status codes of the GET and HEAD results are even uniform, still, the picture for the remaining methods looks more diverse. All tested services provide, e.g., a distinct interpretation of the POST method. Amazon offers the POST method as an upload function for clients which do not support the PUT method—such as Web browsers. This induces the requirement for the client to encode the payload as "multipart/form-data" (Amazon, 2014). Due to this missing precondition almost all status codes obtained from the POST tests are 412 (Precondition Failed), since the POST requests contain a resource representation in "application/json" or "application/xml" instead of the expected "multipart/form-data". Google provides the POST method for different functionalities. Clients can create access control rules, add new storage area, compose files, copy files and create new files or meta-data. If the service consumer intends to create new files via POST, it has to use a different URL (Google, 2014) than defined by the specification in Section 2. HP serves POST as an operation to add meta data to accounts, storage areas and files. This is realised via custom HTTP headers while the payload of the POST request remains empty. Also, the HP services do not permit uploading files with the POST method (Hewlett-Packard, 2014). For a file upload into a storage area a PUT request needs to be issued instead. Likewise, Microsoft does not advertise any utilization of the POST method. Any upload or update operation is triggered by the PUT method, even when a client wants to append meta data to storage areas (Microsoft, 2014).

Other inconsistencies can be observed by focusing on the status codes contained in the response messages. One example of this divergence is the non-uniform treatment of regular PUT requests by all evaluated services (see PU.1 and PU.2). Amazon and Google reply with a status code of 200 (OK) whether a stored resource is updated or a new resource is created. HP and Microsoft, on the contrary, reply with 201 (Created) for both cases: the creation of a new file or the update of an existing file. Other discrepancies can, e.g., be found in the answers to a PUT request providing a wrong resource identifier (PU.6). Amazon and Google behave in the same way and react by sending back a status code of 400 (Bad Request). HP interprets the resource identifier as intention to create a new storage area and hence returns 202 (Accepted). Microsoft responds with 404 (Not Found) in order to

signal that this resource is not available.

The biggest discrepancy can be illustrated by OP.7, where all providers reply with diverse response codes. In this destructive test, a wrong HTTP version is provided in the request line. Amazon responds with the expected code 505 (HTTP Version Not Supported) according to the defined specification (see Listing 1 and Table 2).

```
*** OPTIONS -- OP.7: Unknown Protocol Version ***
Request:
OPTIONS /rssblobs/blob HTTP/1.2
Accept: application/json
Connection: Close
User-Agent: REAL SOA Security, REST-CTT
Host: s3-eu-west-1.amazonaws.com
[...]

Response:
HTTP/1.1 505 HTTP Version Not Supported
Server: AmazonS3
Connection: close
[...]
```

Listing 1: OPTIONS request with an unknown protocol version (Amazon).

HP returns a 501 (Not Implemented) and indicates in the response payload that the method may not be implemented although the OPTIONS method is supported (see Listing 2 and Table 2).

```
*** OPTIONS -- OP.7: Unknown Protocol Version ***
Request:
OPTIONS /v1/11876826348381/blobs/blob HTTP/1.2
Accept: application/json
Connection: Close
User-Agent: REAL SOA Security, REST-CTT
Host: region-a.geo-1.objects.hpcloudsvc.com
[...]

Response:
HTTP/1.0 501 Not Implemented
Content-Length: 28
Content-Type: text/html
[...]

This method may not be used.
```

Listing 2: OPTIONS request with an unknown protocol version (HP).

In case of Microsoft and Google, both services do not conform to the HTTP specification in relation to this destructive test case. These services ignore the unfamiliar protocol and still process the request or reply with another error code. Listings 3 and 4 demonstrate this with the request/response messages logged during the test execution of the Google and Microsoft storage services. The latter one even replies with a self-defined reason phrase instead of the standardised reason for the status code 400 which is Bad Request (see Listing 4).

```
*** OPTIONS -- OP.7: Unknown Protocol Version ***
Request:
OPTIONS /rssblobs/blob HTTP/1.2
Accept: application/json
Connection: Close
User-Agent: REAL SOA Security, REST-CTT
Host: storage.googleapis.com
[...]

Response:
HTTP/1.1 200 OK
Connection: close
Content-Length: 0
Content-Type: text/html; charset=UTF-8
[...]
```

Listing 3: OPTIONS request with an unknown protocol version (Google).

```
*** OPTIONS OP.7 Unknown Protocol Version ***
Request:
OPTIONS /blobs/blob HTTP/1.2
Accept: application/json
Connection: Close
User-Agent: REAL SOA Security, REST-CTT
Host: realsoasecurity.blob.core.windows.net
[...]


Response:
HTTP/1.1 400 A required CORS header is not present.
Content-Length: 293
Content-Type: application/xml
[...]
```

Listing 4: OPTIONS request with an unknown protocol version (Microsoft).

None of the analysed services supports the PATCH (Dusseault and Snell, 2010) method. A reason might be that a partial update of files is not regarded as necessary, although, partial updates to overwrite certain blocks within a file or append data to an existing file might be sensible operations which are actually in parts available via the PUT method instead.

Besides this incoherent behaviour of the services, the evaluation reveals a critical behaviour of all services in cases in which a request contains a content length value that is larger than the actual payload size. Such a destructive request confuses the service, keeping it waiting for the anticipated missing bytes that will never arrive and causing the TCP connection to remain open (see Table 2). This weakness could potentially lead to DoS (denial of service) vulnerabilities.

Other interesting evaluation results expose that the services do not issue an error message if a request contains a mismatch in the specified content type and the actual payload format. All audited storage services store a JSON payload, e.g., as XML, when the Content-Type header contains the value “application/xml”. As a consequence, if the file will be downloaded it embodies this mismatch too.

Another observation is that the services do not care about the media type value within the Accept header and do not check if the media type in the Accept header matches with the inquired resource. In other words, if clients want to access an object as JSON and denote this by setting the value “application/json” in the Accept header, but the addressed resource is an XML document, the server ignores the

Accept header and sends back the XML file without further notice.

One of the two added tests, which sets the Content-Length header to a smaller value than the actual body size (PU.5, test case 59), exposes a different behaviour between the evaluated services. Microsoft, HP and Google store the amount of bytes denoted by the Content-Length header value and do not consider any further bytes possibly contained in the payload. Thus, a request carrying the text string "HelloWorld" as its payload and specifying a Content-Length of five will result in storing the first five bytes of the supplied string, i.e. "Hello". Due to the signature protection scheme enforced by Amazon for their services, such a mismatch leads here to an error. The signature validation procedure is not documented, but with the test tool it could be deduced, that the content length value is used to control the amount of data that is hashed during signature verification. Any mismatch will lead to a negative verification, which is signalled by 400 (Bad Request).

The second added test uploads an oversized file to the storage service which goes beyond the maximum allowed file size (PU.10, test case 60). This test revealed a homogeneous behaviour between Amazon and Microsoft according the request payload processing. According to the obtained results, the two services inspect the Content-Length header in order to find out whether a request contains an outsized payload or not. However, both providers differ by the returned status codes. Microsoft returns a specification compliant status code 413 (Request Entity Too Large) while Amazon responds with a non-compliant status code 400 (Bad Request). HP does not consider the value of Content-Length header. Instead, it counts the bytes on the TCP stream until the threshold of the allowed maximum file size is reached. Google does not state any official information about the maximum file size rendering this test not feasible.

Amongst the discussed findings are many shortcomings at the service access and service communication layer, but also issues in each of the services' business logic. This emphasizes the benefits of the proposed conformance testing. It further highlights the current state in REST-based service practice and the impact of missing standardisations. The heterogeneity of all services relating to status codes and meta data headers are not in line with the idea of a Uniform Interface which is a crucial constrain of the Fielding's architectural style. This violation cumbers automatic code generation, loose coupling of services and interoperability. Still, in comparison to (Gorski et al., 2014b) the analysed Cloud storage services are much more coherent with the REST-ful HTTP specification defined for the purpose of this paper than what is produced by REST frameworks. Thus, an experienced developer but unversed in respect to REST will not be able to produce services which are aligned with the REST principles.

7 CONCLUSIONS AND OUTLOOK

Having more reliable standards in terms of technical specifications which define clear mappings between the general REST principles and concrete technological instantiations is vital in order to address the explored issues. Standardisation ambitions such as the proposed REST-ful HTTP specification introduced in Section 2 would foster approaching advanced requirements with REST and would leverage the interoperability of REST-based services. Service properties known from SOA such as the discoverability, and the loose coupling of services are otherwise not implementable across platforms and systems. The latter property requires to a certain extent the automatic generation of code, which is a lacking feature in the REST domain so far, requiring, e.g., the development of REST client code from scratch without any meaningful tool support. Furthermore, this inhomogeneity even affects Cloud Computing concepts such as Elastic Computing (Dustdar et al., 2011), where software systems are adjustable automatically, depending on required resources, quality of service parameters and operational costs.

From these arguments, the demand for a more elaborated standardisations gets evident. In practice, however, a real need for adequate standards might still not be given for some reasons. By means of the contributed conformance testing methodology and tooling a field study has been conducted to determine the impact of this lack of standardisation on real world services. The analysis of distinct REST-based Web Services from four major Cloud-based storage providers made it apparent that the tested services differ in many aspects amongst each other and do so also in respect to the defined specification. This inhomogeneous picture emphasises the consequences resulting from the diffuse understanding of REST and its technical instantiations.

The introduced method for performing the conformance testing of REST-based services would have a broad influence on the overall development of REST. Not only it would provide a mechanism to increase the comprehension of REST and its mapping to HTTP/URI, it would also increase the interoperability of REST components implemented and operated on

heterogeneous platforms. The latter ultimately paves the way for features such as automatic code generation and the loose coupling of REST-based services.

Many other aspects will benefit from a stringent standardisation and an accompanying enhancement in REST service testing. Security is one such facet. A stable and reliable foundation for building REST services is the basis for a deep integration of security means into REST services, which is a required prerequisite in order to provide an answer to the challenging security demands in cross-organizational distributed services systems (Gorski et al., 2014a).

This specification does not constitute the ultimate standard for building REST-based Web Services. Some definitions might be arguable or need to be discussed in more detail by future standardisation activities. Some aspects have even been left out of scope such as security, caching, versioning, streaming and error treatment. Nevertheless, such kind of technical documentation is a crucial basic assistance for software developers and architects to design interoperable REST-based service applications.

REFERENCES

Amazon (2014). Authenticating Requests in Browser-Based Uploads Using POST (AWS Signature Version 4). http://docs.aws.amazon.com/AmazonS3/latest/API/sigv4-UsingHTTPPOST.html.

Amundsen, M. (2014). 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)*.

Bennara, M., Mrissa, M., and Amghar, Y. (2014). An Approach for Composing RESTful Linked Services on the Web. In *5th International Workshop on Web APIs and RESTful Design (WS-REST)*.

Berners-Lee, T., Fielding, R., and Masinter, L. (2005). Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, IETF. http://www.ietf.org/rfc/rfc3986.txt.

Dusseault, L. and Snell, J. (2010). PATCH Method for HTTP. RFC 5789, IETF. https://tools.ietf.org/ html/rfc5789.

Dustdar, S., Guo, Y., Satzger, B., and Truong, H.-L. (2011). Principles of elastic processes. *IEEE Internet Computing*, 15(5).

Elia, I. A., Laranjeiro, N., and Vieira, M. (2014). A Field Perspective on the Interoperability of Web Services. In *11th IEEE International Conference on Services Computing (SCC)*.

Erl, T., Carlyle, B., Pautasso, C., and Balasubramanian, R. (2013). *SOA with REST - Principles, Patterns and Constraints for Building Enterprise Solutions with REST*. Pearson Education.

Fielding, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine. https://www.ics.uci.edu/∼fielding/pubs/dissertation/top.htm.

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T. (1999). Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, IETF. http://www.ietf.org/rfc/rfc2616.txt.

Fielding, R. and Reschke, J. (2014). Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, IETF. http://tools.ietf.org/html/rfc7231.

Google (2014). Google Cloud Storage - API Reference. https://developers.google.com/storage/docs/json_api/v1/.

Gorski, P. L., Lo Iacono, L., Nguyen, H. V., and Torkian, D. B. (2014a). Service Security Revisited. In *11th IEEE International Conference on Services Computing (SCC)*.

Gorski, P. L., Lo Iacono, L., Nguyen, H. V., and Torkian, D. B. (2014b). SOA-Readiness of REST. In *3rd European Conference on Service-Oriented and Cloud Computing (ESOCC)*.

Gregorio, J., Fielding, R., Hadley, M., Nottingham, M., and Orchard, D. (2012). URI Template. RFC 6570, IETF. http://tools.ietf.org/html/rfc6570.

Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.-J., Nielsen, H. F., Karmarkar, A., and Lafon, Y. (2007). SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). Recommendation, W3C. http://www.w3.org/TR/soap12-part1/.

Hewlett-Packard (2014). HP Helion Public Cloud Object Storage API Specification. https://docs.hpcloud.com/api/object-storage#4.RESTAPISpecifications.

Microsoft (2014). Microsoft Developer Network - Blob Service REST API. http://msdn.microsoft.com/en-us/library/dd135733.aspx.

Nottingham, M. (2010). Web Linking. RFC 5988, IETF. https://tools.ietf.org/html/rfc5988.

Postdot Technologies (2014). Postman. http://www.getpostman.com/.

Psztyc, P. (2014). Advanced REST Client. http://chromerestclient.appspot.com/.

Sarid, U., Hervery, M., Lazarov, I., Rexer, P., Harnon, J., Lane, K., Musser, J., Gullotta, T., and Choudhary, S. (2014). RAML Version 0.8: RESTful API Modeling Language. Specification. http://raml.org/spec.html.

SmartBear Software (2014). SoapUI. http://www.soapui.org/REST-Testing/getting-started.html.

Verborgh, R., Steiner, T., Van Deursen, D., De Roo, J., Van de Walle, R., and Gabarró Vallés, J. (2011). 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)*.

APPENDIX

Table 2: Obtained status codes by Cloud-based storage services for test cases.

#	Test identifier and description	Amazon	Google	HP	Microsoft	Specification
1	PO.1 Content-Type application/json	412	400	204	400	201
2	PO.1 Content-Type application/xml	412	400	204	400	201
3	PO.2 Unsupported Content-Type	412	400	204	400	415
4	PO.3 Content-Type and payload mismatch	412	400	204	400	400
5	PO.3 No Content-Type but with payload	412	400	204	400	400
6	PO.4 Content-Length bigger than payload size	412	No Response	No Response	400	400
7	PO.4 Content-Length as String	400	400	400	400	400
8	PO.4 No Content-Length	412	411	204	411	411
9	PO.5 Wrong action on resource	405	400	404	405	405
10	PO.5 Not existing resource	412	400	204	405	404
11	PO.6 Malformed application/json	412	400	204	400	400
12	PO.6 Malformed application/xml	412	400	204	400	400
13	PO.7 Wellformed application/json, unprocessible content	412	400	204	400	400
14	PO.7 Wellformed application/xml, unprocessible content	412	400	204	400	400
15	PO.8 Unknown protocol version	505	400	501	400	505
16	OP.1 Ping *	400	200	200	400	200
17	OP.2 Regular	400	200	200	400	200
18	OP.2 Regular with resource id	400	200	200	400	200
19	OP.3 Accept application/json	400	200	200	400	200
20	OP.3 Accept application/xml	400	200	200	400	200
21	OP.4 Unsupported media type in accept header	400	200	200	400	415
22	OP.5 Wrong resource identifier	400	200	200	400	404
23	OP.5 Not existing resource	400	200	200	400	404
24	OP.6 Containing content	400	200	200	400	400
25	OP.7 Unknown protocol version	505	200	501	400	505
26	HE.1 Accept application/json	200	200	200	200	200
27	HE.1 Accept application/xml	200	200	200	200	200
28	HE.2 Unsupported media type	200	200	200	200	406
29	HE.3 Wrong resource identifier	404	404	404	404	404
30	HE.3 Not existing resource	404	404	404	404	404
31	HE.4 Containing content	200	400	200	200	400
32	HE.5 No Accept header	200	200	200	200	200
33	HE.6 Unknown protocol version	505	200	501	200	505
34	GE.1 Accept application/json	200	200	200	200	200
35	GE.1 Accept application/xml	200	200	200	200	200
36	GE.2 Unsupported media type	200	200	200	200	406
37	GE.3 Wrong resource identifier	404	404	404	404	404
38	GE.3 Not existing resource	404	404	404	404	404
39	GE.4 Containing content	200	400	200	200	400
40	GE.5 No Accept header	200	200	200	200	200
41	GE.6 Unknown protocol version	505	200	501	200	505
42	PU.1 Content-Type application/json	200	200	201	201	204
43	PU.1 Content-Type application/xml	200	200	201	201	204
44	PU.2 Unsupported Content-Type	200	200	201	201	415
45	PU.3 Partial update with Content-Type application/json	200	200	201	201	400
46	PU.3 Partial update with Content-Type application/xml	200	200	201	201	400
47	PU.4 Content-Type and payload mismatch	200	200	201	201	400
48	PU.4 No Content-Type but with payload	200	200	201	201	400
49	PU.5 Content-Length bigger than payload size	No Response	No Response	No Response	No Response	400
50	PU.5 Content-Length as String	400	400	400	400	400
51	PU.5 No Content-Length	411	411	411	411	411
52	PU.6 Wrong resource identifier	400	400	202	404	404
53	PU.6 Not existing resource	200	200	201	201	404
54	PU.7 Malformed application/json	200	200	201	201	400
55	PU.7 Malformed application/xml	200	200	201	201	400
56	PU.8 Wellformed application/json, unprocessible content	200	200	201	201	400
57	PU.8 Wellformed application/xml, unprocessible content	200	200	201	201	400
58	PU.9 Unknown protocol version	505	200	501	201	505
59	PU.5 Content-Length smaller than payload size	400	200	201	201	400
60	PU.10 Content-Length exceeding the allowed payload size	400	?	413	413	413
61	PA.1 Content-Type application/json	405	405	501	400	204
62	PA.1 Content-Type application/xml	405	405	501	400	204
63	PA.2 Unsupported Content-Type	405	405	501	400	415
64	PA.3 Complete update with Content-Type application/json	405	405	501	400	204
65	PA.3 Complete update with Content-Type application/xml	405	405	501	400	204
66	PA.4 Content-Type and payload mismatch	405	405	501	400	400
67	PA.4 No Content-Type but with payload	405	405	501	400	400
68	PA.5 Wrong Content-Length	405	No Response	501	400	400
69	PA.5 Content-Length as String	400	400	501	400	400
70	PA.5 No Content-Length	405	405	501	400	411
71	PA.6 Wrong resource identifier	405	405	501	400	404
72	PA.6 Not existing resource	405	405	501	400	404
73	PA.7 Malformed application/json	405	405	501	400	400
74	PA.7 Malformed application/xml	405	405	501	400	400
75	PA.8 Wellformed application/json, unprocessible content	405	405	501	400	400
76	PA.8 Wellformed application/xml, unprocessible content	405	405	501	400	400
77	PA.9 Unknown protocol version	505	405	501	400	505
78	DE.1 Regular	204	204	204	202	204
79	DE.3 All resources	409	409	409	400	405
80	DE.4 Not existing resource	204	404	404	404	404
81	DE.5 Containing content	204	400	204	202	400
82	DE.6 Unknown protocol version	505	204	501	202	505
83	EV.1 Accept application/json	405	502	501	400	501
84	EV.1 Accept application/xml	405	502	501	400	501
85	EV.2 Unsupported media type in accept header	405	502	501	400	501
86	EV.3 Wrong resource identifier	405	502	501	400	501
87	EV.4 Containg content	405	502	501	400	501
88	EV.5 Unknown protocol version	505	502	501	400	501