An Analysis of Public REST Web Service APIs

Andy Neumann, Nuno Laranjeiro, Jorge Bernardino

Abstract—Businesses are increasingly deploying their services on the web, in the form of web applications, SOAP services, message-based services, and, more recently, REST services. Although the movement towards REST is widely recognized, there is not much concrete information regarding the technical features being used in the field, such as typical data formats, how HTTP verbs are being used, or typical URI structures, just to name a few. In this paper, we go through the *Alexa.com* top 4000 most popular sites to identify precisely 500 websites claiming to provide a REST web service API. We analyze these 500 APIs for key technical features, degree of compliance with REST architectural principles (e.g., resource addressability), and for adherence to best practices (e.g., API versioning). We observed several trends (e.g., widespread JSON support, software-generated documentation), but, at the same time, high diversity in services, including differences in adherence to best practices, with only 0.8% of services strictly complying with all REST principles. Our results can help practitioners evolve guidelines and standards for designing higher quality services and also understand deficiencies in currently deployed services. Researchers may also benefit from the identification of key research areas, contributing to the deployment of more reliable services.

Index Terms— REST, RESTful, Web Services, API, HTTP, Web, Web Services Analysis

1 INTRODUCTION

The number of services providing publicly available web Application Programming Interfaces (APIs) has been growing rapidly [1]. Several studies point out that developers have moved from Simple Object Access Protocol (SOAP) or Remote Procedure Call (RPC) to deploying Representational State Transfer (REST) web services, as the means for consumers to use their services [1]–[4]. This is corroborated by major websites like Google, Facebook, or Twitter, which are now deploying REST services to provide easy access to their valuable data resources, while promoting their businesses [1].

The REST architecture was introduced in the year 2000, by Thomas Fielding, and is based on the principles that support the World Wide Web [5]. In summary, according to the REST principles [5], REST interfaces rely exclusively on Uniform Resource Identifiers (URI) for resource detection and interaction, and usually on the Hypertext Transfer Protocol (HTTP) for message transfer [3], [6], [7]. A REST service URI only provides location and name of the resource, which serves as a unique resource identifier. The predefined HTTP verbs are used to define the type of operation that should be performed on the selected resource (e.g., GET to retrieve, DELETE to remove a resource).

Possibly due to HTTP's features (which fit the REST architecture rather well), long-term presence, and general understandability, REST has become a *de facto* standard way for offering a service on the Web [2], [8]. Despite this, REST is merely an architectural style, provided without standard specifications. This implies that several decisions have to be made by developers when exposing service APIs, which may result in diverse APIs and, in some cases, in poor design decisions (e.g., using a single HTTP verb for retrieving or deleting a resource). These decisions will impact the client-side developer, that must adapt to the specific style being used, and may even affect the provider (e.g., when a poorly maintainable service is deployed).

Server-side developers must define how an API should be exposed (e.g., which URI design schema to use), which characteristics it should possess (e.g., which output formats are supported) or how the documentation is provided (e.g., created with documentation generation software). For instance, this last item is problematic because without standard means for documenting APIs, the tendency will be to use text to describe the API (many times in natural language), which may be diverse in form, structure, or depth and is obviously a problem for client developers [3]. The literature also suggests some heterogeneity in how HTTP features are used, with some being widely adopted (e.g., HTTP verbs, status codes) and others tending to be ignored (e.g., HTTP headers) [2], [3]. Also highly discussed is the adoption of the *Hypermedia As The Engine Of Application State* (HATEOAS) principle of the REST architecture, which is rarely used [1]–[3], [9], [10].

Due to the abovementioned reasons, it is a common belief that most online services available online claim to be REST services without truly following the REST principles, and present severe technical inconsistencies, diverse designs, and operating modes [1]–[3], [9], [10]. Thus, some web services expose Web APIs very close to the original style described by Fielding [5], whereas others simply change their existing RPC-style API to being directly accessible via HTTP [2]. To what extent this currently happens is not very clear, and there is the need of using new data as a means to provide an additional data point that supports or disputes findings of previous work.

In this paper, we empirically analyze the technical features of 500 public REST Web Service APIs, found in the top 4000 most popular sites on the Web (according to *alexa.com*). Our analysis goes through 26 API features, identified in the literature [1]–[4] and gathered during our observations, with the goal of understanding: i) the level of

• *A. Neumann, N. Laranjeiro, and J. Bernardino are with the CISUC, Department of Informatics Engineering, University of Coimbra, Portugal. E-mail: andy@finekey.de, cnl@dei.uc.pt, jorge@isec.pt*
• *J. Bernardino is also with the ISEC - Coimbra Institute of Engineering, Polytechnic Institute of Coimbra, Portugal.*

adoption of REST principles (e.g., the design schema used in the URIs); ii) the common design decisions made by service developers, e.g., is Extensible Markup Language (XML) or JavaScript Object Notation (JSON) a popular choice for service output data) and; iii) if REST service development best practices [2]–[4], [9], [11]–[17] are being followed by programmers (e.g., documenting services in a structured manner). In summary, we aim to be able to answer the following research questions:

RQ1: *Are web services being deployed following a pure REST architecture?*

REST was built as a model of how the Web should work [5], and violating any of its set of principles puts its objective at risk. The outcome is heterogeneity of service design that emphasizes systems heterogeneity, with impact on client developers and overall system interoperability [2], [3]. The client developer becomes dependent on the specific and error-prone decisions of the service programmer. The extension of this heterogeneity must be known so that proper measures can be taken by developers and so that the gaps are analyzed by researchers to advance the state of the art.

RQ2: *Which are the common design choices taken by practitioners when developing REST web services?*

In addition to the REST architecture, developing services requires several choices regarding technical features, such as type of output format supported, authentication mechanism used, etc. Possessing this kind of data is crucial for practitioners, as available information is short, highly scattered, and, due to the dynamicity of the Web, also quickly outdated. In the end, it can help guiding the (re)definition of best practices.

RQ3: *Are best practices for REST web services being followed by service developers?*

Best practices are built by researchers and practitioners, as the result of research and experience, to guide service development. Identifying those not followed can help understanding the disconnection between theory and practice and this insight may be useful for, for instance, refining or creating new best practices. Also, understanding the degree to which practices are followed can be used to reinforce or improve best practices, which in the end contributes towards more reliable services.

Our results disclose several trends regarding decisions made by service developers when designing and exposing their REST web APIs. Examples include the popular use of software-generated API documentation, the definitive decay of XML in favor of JSON, which is now the most widely supported output format, the use of OAuth, or the enforcement of call limits, just to name a few. In some features, the APIs show some diversity, such as the way clients select the output format or the used documentation generation software. We observed significant differences regarding the adoption of the REST principles, resulting in a very low number (4 out 500 services) of *truly RESTful* services (i.e., compliant with all principles). Finally, we also observed high adherence to a subset of best practices (e.g., semantically correct use of HTTP verbs), while also showed residual appearances (e.g., HATEOAS [5]).

The rest of this paper is organized as follows. Section 2 presents related work in web API analysis. Section 3 describes the design of this study and section 4 presents the results. Section 5 discusses the main findings and Section 6 reviews existing threats to validity. Finally, section 7 concludes this paper.

2 Related Work

We identified the following two main groups of research work that provide information regarding web APIs: i) Works that empirically analyze a set of web services for specific API characteristics or features; and ii) Works that provide information about quality attributes of web services, including open issues and challenging aspects.

In what concerns **empirical analyses** of web services, the work in [1] is an interesting case that analyses 222 publicly available web API documentations, selected from *programmableweb.com*, against 20 features. Results show that RESTful web services had been relatively widely adopted by 2010, although the authors emphasize that developers tend to disregard the REST principles, which we confirm in this work in a larger-scale study and almost a decade later. In [2], twenty REST web services, selected among the APIs with higher number of mashups from *programmableweb.com*, are analyzed against 17 features. Results show that almost none of the services is actually truly RESTful, despite the small dataset.

The authors in [3] analyzed the 45 most popular web APIs of the *programmableweb.com* repository using the number of mashups as the sorting metric and against 17 features. Results show diversity in the adoption of REST principles or in common design decisions, which is attributed to missing standards regarding REST. The rather small dataset and the time passed since then make it necessary to take another up-to-date look on the current state of web APIs.

Authors in [8] aimed to understand the meaning of micro services for practitioners by questioning 42 companies. The focus of the work is on micro services, how they are technically implemented, and which difficulties are present. Main findings show that two thirds of the companies use external services, but also about two thirds of the services used are internally developed and operated. REST and HTTP are quite present, while SOAP rarely appears. JSON is more common than XML and the most often used programming languages for implementing the companies' internal REST services are Java, JavaScript, C#, Python and Ruby. The study analyzes a certain services niche, while we aim for a broader study.

The authors in [9] analyze the degree of compliance with the REST architectural principles, from a point of view of mobile applications. 78GB of data logs of HTTP requests collected from a large mobile operators are used to identify patterns and matched against 26 best practices for web services and the Richardson Maturity Model [18]. The largest part of their analyzed dataset complies with level 2 of this model, whereas only a few hosts reach level 3. This study is only based on the analysis of live HTTP requests and does not examine the corresponding API itself and respective documentation.

The services of three popular cloud providers are compared against a catalog of 73 best practices for REST APIs design in [13]. Results show that these cloud providers have reached an acceptable level of maturity, even though they follow only half to two thirds of the catalog guidelines. The work in [10] analyzes 286 Swagger API description documents and provide a framework for a structural analysis of REST APIs to identify the main characteristics and deficits. With roughly a third of the 286 APIs being hosted by Google and by analyzing only Swagger documents, the authors focus on a particular niche. As we will see in the next sections, we aim for a broader study, where even services with no structured documentation are analyzed.

In [19] 500 popular apps and 15 popular services for the android ecosystem are analyzed, with the results showing that application developers prefer official SDKs for accessing these services over simple HTTP clients. It is a small set of 15 services (and with focus on android applications), thus it is beneficial to have a broader view on this matter.

The authors in [14] analyze 18 popular web services according to 12 linguistic patterns and anti-patterns by semantically analyzing the URIs of web services. Results include the detection of syntactical URI design issues, although the API designers tend to use adequate resource names and no verbs in URIs (a REST best practice). The authors do not analyze other REST-relevant features, such as the use of HTTP headers.

Non-functional aspects (e.g., payment plans and business models) of 69 web services extracted from *market.mashape.com* and *programmableweb.com* are analyzed in [21] with the goal of identifying requirements of a governance model of realistic RESTful APIs. The authors observe a wider expression of API limitations (e.g., the client is limited to certain parts of the functionality, in a certain payment plan) if the API is not regulated by an API Gateway (e.g., *market.mashape.com*). We also go through non-functional aspects, such as existence of payment plans and call limit rates, but intend to provide a wider focus, by analyzing a broader set of technical features, including REST principles compliance and best practices.

Regarding the works that provide **information on quality attributes** of web services, including open issues and challenging aspects. It is relevant to mention the proposal of an approach for benchmarking the quality of web services considering geo-mobility of the service end-user presented in [17]. High-level architectural and engineering options (e.g., client-side caching, the use of a backup API that could leverage machine-readable API descriptions) are discussed as means to handle variable Quality of Service (QoS), observed during long-term benchmarking experiments. The authors do not present detailed insights on the features of the services under benchmark.

Software engineering research opportunities for the consumption of web APIs are discussed in [22]. The challenges discussed are: i) service consumers have no control over the web service (the provider may change the API or the service); ii) clients may not be sure of the validity of calls to the API at run-time; iii) SDKs out of synchronization with the actual service; iv) and QoS issues. The authors identify a few research lines (e.g., static analysis for checking requests, documenting API signatures, coding practices and patterns for dealing with varying QoS, impact of web API usage on non-functional aspects). Our work can help complementing this kind of study, by providing further detailed data for research to take place.

A language for modeling REST client-server conversations is presented in [20]. Authors refer that most APIs are simply exposing low-level HTTP details without hypermedia controls (which creates obstacles for conversation), which are two features that we analyze in this work for an up-to-date view. In [4], the authors analyze the evolution of the interface of client-server technologies (including web APIs). The authors overview technologies used for web services and point out reasons for the misuse of REST principles. The authors emphasize that some designs and mechanisms (e.g., REST design, OAuth) are much more successful at web API reuse and discuss the usefulness of an interface description for REST web APIs. In this paper, we analyze services in the field for concrete evidence on these and additional aspects.

In this paper, we designed a study with the broad objective of understanding the state of the practice regarding REST web services. We are providing an additional and updated data point that may support previous work. In fact, due to the dynamicity of the Web environment, such works are quickly outdated, leaving practitioners and researchers with little information for designing and developing services, or pursuing research on this topic.

3 Methodology

The empirical analysis carried out in this paper was performed from February to July 2017. For simplicity, from this point onwards we use the term *REST (*accompanied by *web service API*, *service API* or *web API)* to designate a service that is claimed to be RESTful by some provider. We use the term *truly RESTful* if it adheres to all mandatory REST principles [5]. The study comprised the following steps, described in further detail in the next paragraphs:

1) Identification of best practices for REST API development [2]–[4], [9], [11]–[17];
2) Find a suitable URI source to use as basis for the analysis;
3) Filter duplicate sites (e.g., regional site variants);
4) Identify the site category (e.g., games, shopping);
5) Examine the site for available service APIs:
    a. Identify service APIs announced as REST by the site and select a subset for analysis;
    b. Identify other types of service offered by the site.
6) Analyze technical features (identified in previous works [1]–[4]) and gathered during our observations) of the REST service APIs, going through:
    a. Compliance with REST principles [5];
    b. Service design decisions;
    c. Adherence to best practices [2]–[4], [9], [11]–[17].
7) Verification of the results.

We started by **identifying best practices** in the literature by searching papers in Google Scholar, the results were filtered to only include peer-reviewed papers that have been cited (unless they have been published in the last year). This allowed us to identify the whole set of best practices. We then used Google Search Engine in search for a practitioner view on this matter, which allowed us to further support three practices that had been already identified in research work.

The second step involved **selecting a source of websites**, that we could use as basis for our whole analysis. There are a few options for this, such as *alexa.com*, *programmableWeb.com, or apis.guru*, among others. We selected *alexa.com* as site source, as it is a mature, well-known, and periodically updated source of popular sites. Certainly, there is no direct link between the popularity of a website and the popularity of its exposed web service, but its wide application in previous similar research contexts [23], [24] provides us assurances about its usefulness. The analysis, described in detail in the next section, was carried out using the top 4000 most popular web sites as basis.

The selected list of URIs was then processed to **filter duplicate sites**, as many times popular sites include regional variants, such as *google.pt* or *google.co.uk*. Including these variants would add an artificial bias to the results, as such sites are essentially mirrors deployed by the same company. Thus, we excluded repeated websites that only differ in their top-level domain.

Alexa.com is well-known for its top popular site lists, but it does not categorize websites. So, we opted to also add some context about the type of sites being analyzed (e.g., shopping, finance, games). For this purpose, we used *similarweb.com* (listed, at the time of writing, on position 1589 of *alexa.com*) to **identify each site category**, because it freely provides a classification of sites in categories based on user interests, and has been often used for website analytics in previous works [25]–[27].

The process of **examining the site for an available service API** involves, at first, browsing through the site and searching for information directed for developers. Whenever an API was not found (many times the main site pages do not mention any API) or when the website was not in English, we would direct the search effort to Google's search engine. In this case, we used the name of the site followed by "API", which was useful for a few times. The API was then marked as REST or not, based on what the service provider announced (unofficial APIs, found using Google, were not considered). In some cases, the website would provide several web APIs (we registered the number for later analysis), of which we would randomly select a single one to proceed with the analysis.

After identifying the REST web API, we **manually analyzed the service API official documentation against a set of technical features** (e.g., HTTP verb support, design schema used in URIs, versioning use), mostly extracted from previous studies [1]–[4], and which we present in the next paragraphs. Also, based on our observations, we gathered additional features which include the support for software development kits (i.e., client-side libraries), the latest API documentation update, the use of generated API documentation, presence of interactive API test consoles, usage charges, and enforced call limit rates.

Some of the collected features imply analyzing the APIs multiple endpoints (i.e., URIs), such as the "design schema used in URIs". For APIs with a large number of endpoints (e.g., Facebook has more than 200 endpoints) we examined a subset (but our observations showed that the design was always consistent throughout the API). In short, if the number of endpoints was less than 20 we checked all, else if it was lower than 50 we checked about half, otherwise we checked about one fourth. Similar to [3], and for presentation clarity, we grouped the several features in the following dimensions:

- **REST Architecture –** This dimension mostly covers core REST features of the API [5]: i) number of operations; ii) design schema used in the URIs (i.e., does the URI only contain resource information or does it also contain information about the operation); iii) the technique of the output format selection; iv) the scoping information (i.e., how the server is informed about on which data it should operate, for instance using data specified in the URI path or in an HTTP header field); v) API versioning support (i.e., is it possible and how to select among different API versions, how many major versions were deployed); vi) support for response caching; and vii) use of links to related resources in response messages.
- **HTTP Use –** This dimension refers to: i) the supported HTTP verbs; ii) how HTTP method overrides are implemented (e.g., how to delete a resource if the HTTP DELETE verb is not supported); iii) if and how the messages specify their content type; and iv) the use of adequate HTTP status codes in responses (e.g., sending a success response using a 2xx status code).
- **I/O and Software Development Support –** This covers the input and output formats supported by the services. It also addresses the support of Software Development Kits (SDKs) for the API consumer.
- **Security Mechanisms –** This refers to the presence of user authentication mechanisms (is user authentication required and if yes, which mechanism is supported) and if communication channel encryption is mandatory, optional, or simply not supported.
- **Usage Policies –** This dimension covers the type of registration that is necessary for using the API, whether payment plans for using the API are available, if call limits are enforced and in this latter case, how many daily API calls are allowed.
- **Documentation and Application Use** – This dimension covers general documentation aspects, namely if the APIs provide: i) documentation updated in the current calendar year (at the time of data collection), i.e., between January to July 2017; ii) generated API documentation (i.e., produced by a tool, such as the Swagger UI [28]); iii) interactive developer consoles (i.e., for testing requests and observing responses); iv) explanations of error messages; v) example requests and responses. In addition, we also examine the reported number of applications that use the analyzed APIs.

Of the above dimensions, the **REST architecture** is particularly important, especially due to the wide proliferation of this kind of APIs. Thus, besides the analysis of the REST-related features mentioned above, we also analyze, whenever possible, the level of compliance with Roy Fielding's REST architecture principles [5] (labelled from (1) to (6) in the next paragraphs).

The *Client-Server* principle *(1)* states that the server holds resources, and the client wants to interact with these resources. The client sends requests to the server to access and manipulate data. A REST system must function according to the client-server model and must separate user interface concerns from data storage concerns.

The *Stateless* principle *(2)* requires that no client context is being stored at the server between requests. Each client request contains all necessary information for the server to respond and is treated standalone. Responses must define themselves as *Cacheable (3)* or not cacheable to potentially improve scalability.

A *Uniform Interface (4)* is an essential principle of the REST architecture that allows simple API usage. It consists of the following four properties:

- *Resource addressability*: Every REST service has a URI as a unique address. This address identifies a resource, which standardizes the way of accessing the supplied service.
- *Manipulation of resources through representations*: A client has enough information to manipulate a resource if it holds a representation of this resource and any attached metadata. For instance, if a client has a representation of a user resource and wants to change the user id, the client would only need to send this representation and the new id to the proper endpoint (e.g.,

by using the HTTP PUT verb). A client must not, for instance, send instructions to the server about how to update a resource.

- *Self-descriptive messages*: Requests and responses must be self-descriptive, which means that the recipient receives all necessary information to understand the message and, for instance, must not wait for another message that explains how to interpret the data. By sending REST messages using standard means (e.g., correct HTTP verbs) to manipulate resources, the recipient of the message (server or client) receives all necessary information to complete its task.
- *Hypermedia As The Engine Of Application State (HATEOAS)*: Server response messages must provide links to related resources. These resource links provide clients with other currently available actions, based on the client's current state, which work as a navigation system throughout the API.

The *Layered System* principle (5) states that a client cannot know whether it is connected to the end server, or to an intermediate server. These layers should be hidden, to simplify the architecture. Finally, there is an optional *Code on Demand (6)* principle that states that, only in case of need, the server may send code for the client to execute locally (e.g., the server sending JavaScript code).

Notice that, in the case of the principles, our analysis will be limited as we will not be able to gather complete information for at least the *Stateless* and *Layered System* principles (as we are dependent on the information provided). Still, all service APIs found support the *Client-Server* paradigm.

Due to the lack of common structure and standards of the APIs documentations and due to the complexity of the analysis, it is not possible to automate this kind of work. Thus, extracting and analyzing the data is essentially a manual process. Due to the huge amount of effort involved we **verified** about one fourth of the results by randomly selecting some features among each of the already classified APIs. During this verification procedure, which was carried out by the same person performing the analysis, we found only residual errors. The results of this process are presented in the following section.

4 Results

In this section, we present the results obtained during our analysis, which followed the approach described in the previous section. We begin by overviewing the types of sites (as classified by *similarweb.com*) and identifying the types of APIs provided (e.g., REST, SOAP). We then present the results of the analysis of the 500 REST web APIs identified during this process, from Section 4.1 to Section 4.6, going through the features [1]–[4] presented in Section 3. We discuss common designs, identify compliance with the REST principles [5], and adherence to best practices (identified in the text with *Bp_bestPracticeName*) [2]–[4], [9], [11]–[17]. Detailed results are available at [29].

From the Alexa Top 4000 sites, 315 sites were found to be duplicates (only differing in the top-level domain) and were excluded from the analysis. From the remaining 3685 sites, we found that 681 (18%) provide some kind of API. Fig.1 shows the distribution of the sites and APIs per category (according to *similarweb.com*). Fourteen categories holding values below 1.5% (of the 3685 sites) and websites belonging to undetermined categories (not found in *similarweb.com*) were placed under "Other".

As we can see in Fig. 1, about half of the categories nearly one third of the sites possess some kind of Web API (e.g., "Computer and Electronics", "Shopping, "Internet and Telecom"), while in other categories this tendency seems to fade (e.g., "Games", "Adult"). Overall, what we observe is that the APIs fit in quite diverse categories, emphasizing the large proliferation of web APIs that currently extends to nearly all kinds of sites.

In 13.1% (of the 681 APIs) there is no API documentation in English and in 3.1% there is actually no documentation. Also, in 4.7% it is necessary to create an account to view the documentation and in 2.5%, it is necessary to apply for the API use just to view the documentation. For simplicity, and due to the relatively low number of these cases, we decided to exclude these APIs from the analysis, ending up with a total of 522 APIs.

From the 522 sites, 91% provide only a single API which we analyzed. The remaining 9% provide more than one API, with nearly half (4.4%) providing less than 6 APIs, up to a maximum of 100 APIs by *google.com*. Each API, of these sites that offer several APIs, serves a different purpose. For instance, *ebay.com* offers APIs for selling, buying, or accounting. We observed that, in general, the characteristics and documentation of the different APIs of the same provider tend to be similar. Due to this, and due to the relatively low number of these cases (which could however add some bias towards the decisions of a particular provider), we randomly selected one of the available APIs per site to be part of the analysis.

Before proceeding with the core of the analysis, we must refer that, **based on the providers' claims, we identified APIs that fit in three main technologies or models: REST, XML-RPC and SOAP. A huge amount of the 522 APIs (500 APIs – 95.8%) provide an API based on REST**, which we thoroughly analyze in the next sections. Of the remaining 22 APIs, 2.5% are SOAP APIs and 1.7% are XML-RPC APIs.

About 3.2% of the 500 sites claiming to provide REST APIs also deploy additional alternative APIs, with 2.4% providing also SOAP and 0.6% providing an XML-RPC API. For example, *Flickr.com* provides SOAP and XML-RPC APIs in addition to its REST API (and is actually the only one providing all three main types of APIs). The *Samsung Artik Cloud* is the only site providing other types of APIs, such as WebSockets, MQTT, and CoAP, in addition to REST. These are a few exceptions, but most web APIs (96.8%) are not providing any additional technologies or protocols. These observations clarify the extension of the movement towards the REST architecture. It is not a trend

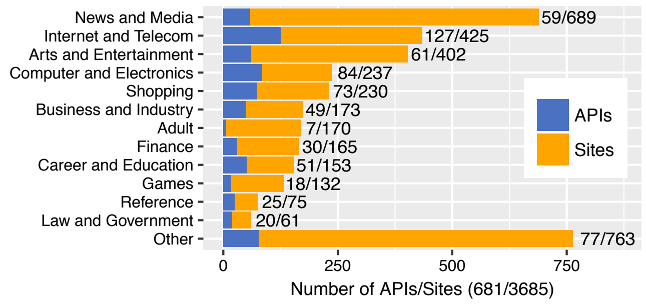

Fig. 1. Number of sites and APIs per category.

anymore, it is now an established practice, built for and put in place by developers [8].

The analysis presented, from this point onwards, refers only to the 500 REST services (as claimed by the service provider) identified during this work and following through the features discussed in the previous section. Note that the complexity and specificity of the documentation resulted in a few cases being incompletely analyzed, which we identify in the next sections. This already suggests deficiencies resulting from the arbitrary way the documentation is produced, which in turn introduces complexity to the typical tasks of a developer (i.e., finding or understanding how to use a certain service).

4.1 REST architecture

We begin by discussing the **number of operations** provided by the 500 APIs, which is not exactly a core REST feature, but provides insight on the typical size of this kind of services. It is also a frequently analyzed feature in similar work [1], [3], [10], [30]. Fig. 2 shows the distribution of the number of operations in the set of 500 services.

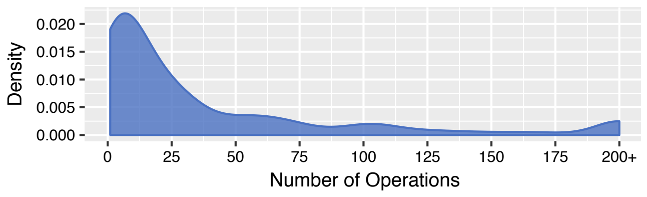

Fig. 2. Distribution of the number of operations.

About two thirds of the APIs (64%) have between 2 and 50 operations. Roughly half of the analyzed services (56.2%) provide at most 20 operations and about one third offer between 2 and 10 operations. In the analyzed set, 5.2% provide a huge number of operations (over 200 operations) and these range from social platform services, such as the *facebook.com Graph API,* to shopping sites, such as the *ebay.com Sell API*. Overall, REST web service APIs tend to provide a relatively small number of operations.

We identified the presence of two types of **design schemas in the 500 URIs**: i) resource-oriented URIs; and ii) operation-oriented *URIs*. In the following paragraphs, we will refer to the resource-oriented URIs as *REST-style URIs* and the operation-oriented URIs as *RPC-style URIs*.

REST-style URIs are structured around resources such as *www.api.com/user/{name}***.** Depending on the HTTP verb that is used, an API call can then either get the name of the user, add, change or delete the name of a user. RPC-style URIs are structured around operations such as *www.api.com/user/getname*. The last part of the URI will change if the user name should be added, changed or deleted, which means that the type of operation does not exclusively rely on the HTTP verb used. Of the 500 web APIs, 87.8% use REST-style URIs (***Bp_RestURI*** [2]–[4], [9], [11]–[16]), while 12.2% use RPC-style. As mentioned, the analysis of 222 APIs in 2010 [1] uncovered only 32.4% using REST-style URIs. More recently, in 2016 [9], authors observed about 90% of URIs avoiding CRUD operations as resource names, which is in line with our observations. Obviously, the datasets are different, but the tendency seems to be the adoption of the *Resource Addressability* property of the *Uniform Interface* REST principle.

We further delved into the difficulties for structuring URIs around resources. Of the 12.2% that use RPC-style URIs, 11.6% could be easily changed to REST-style, as they allow just a few basic operations (e.g., /createfolder, /listfolder) that could be offered on a single resource and handled with (at most) the four common HTTP verbs. The remaining 0.6% offer a great number of operations on a single resource (e.g., one API supports 17 different operations) that go far beyond the four common HTTP verbs, such as archive, unarchive, setTopic, setPurpose or mark. In this kind of case, using REST-style URIs would involve a much larger refactoring of the API, possibly involving specifying the type of operation in the request.

The developers may have reasons for simply using a single HTTP verb and specifying the type of operation directly in the URI. However, using RPC-style URIs is a violation of the *Resource Addressability* principle, because the URI must only contain resource information and cannot expose information about the type of operation.

The **output format selection** feature refers to how the format of the service output is selected. The distribution of the different ways of performing this selection is shown in Fig. 3.

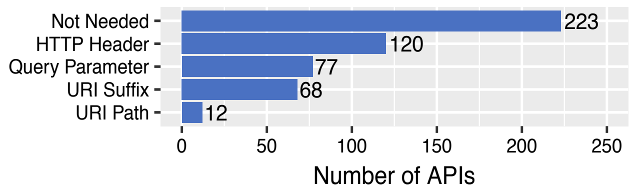

Fig. 3. Output format selection techniques.

The results show that almost half (45% – 223 APIs) of the 500 APIs support only one output format, and do not require any kind of output format selection to be performed. In the remaining cases, the most popular way to select the type of output is to use an HTTP header field (24.0% – 120). Then, 2.4% (12) use the URI path, 13.6% (68) use a URI Suffix, and 15.4% (77) use a query parameter to specify the format. In [9], authors found selection of the output type via query parameters in 3% of the services (vs 15.4% in our case) and via URI suffix or URI path in 27% of the services (16% in our case).

Using the URI to specify the output format violates the *Resource Addressability* principle, because the URI must only refer to resource information. It is not clear, why only about one fourth of the services use the HTTP header fields as the way of selecting format, when the HTTP protocol specifies an HTTP *accept* header field (***Bp_AcceptHeader*** [3], [11], [12]), precisely for this purpose. Specifying the format in the query parameter is a simple option that may not be a best practice, but also does not violate the *Resource Addressability* principle.

The information that informs the server on which data or resource it should operate is named **scoping information**. Scoping information is associated with the method information (in *lato sensu*) that determines the operation the server should perform (e.g., create or update a resource). For instance, in the following API call example:

GET /user/andy

The method information is represented by "GET" and the scoping information is */user/andy*. This "GET" action is performed on the */user/andy* resource, meaning that the client wants to retrieve the available data of user andy. This example only uses the URI path as the scoping information. We identified four ways of providing the scoping information in the 500 service APIs: i) URI path; ii) query parameters; iii) HTTP header fields; and iv) URI paths and query parameters. Fig. 4 shows the distribution of the different ways of encoding the scoping information.

As we can see in Fig. 4, almost two thirds of the APIs (63.8% – 319 APIs) encode the scoping information exclusively in the URI path. According to REST, this is the correct way (***Bp_ScopingURI*** [2]–[4], [9], [11]–[16]), which agrees with the *Resource Addressability* principle that states that the scoping information should be encoded in the URI path as a unique resource identifier. About 28% (141) of the APIs use query parameters to specify additional scoping information, while the URI path specifies the main resource information. For example, *cnet.com* uses this latter technique, as in the following example:

GET */author?personIds=9*

The URI path specifies that the service should operate on the author resource and the query parameter *personIds* specifies the Id of the author. 7% (35) of the APIs require scoping information to be encoded only in the query parameter (half of these 35 APIs mostly provide search operations and the search query is encoded as a query parameter). Five APIs (1.0%) encode the scoping information only in an HTTP header field.

Regarding versioning, about one fifth of the APIs (19%, 95 APIs) do not provide any information about API versions, but we were able to determine the major version number of the remaining 89%. In 15.6% (78) this information is part of the documentation and in 65.4% (although not documented) the version number is exposed within the API request call (in different ways). Versioning the API **(*Bp_Versioning*)** is obviously a best practice [31], as it is unlikely that a web API will remain static. The distribution of the **number of major API version releases** is shown in Fig. 5, where we can see that more than two thirds of the APIs are deploying either their first or second major version.

After an API provider deploys a new major API version, developers that want to start using the new version may be forced to change their applications [32]–[34], depending on which version selection technique they are using. Fig. 6 shows the different techniques that are used for **selecting the API version**, which is possible for about two thirds of the analyzed web APIs (65.4% – 327 APIs). The remaining (34.6%, i.e., 173 APIs) do not support any kind of version selection).

Nine out of ten APIs that allow version selection are using the URI path to select the version, which may violate the *Resource Addressability* principle, because the URI must only contain information about the location and name of a resource. Selecting the version in an HTTP header field is a more REST-conform way to do it [13], [36], but also adds complexity to client calls, because the client needs to build a header field appropriately and cannot just use a simple URI. Selecting the version within the URI (or as a query parameter) is, in comparison, much simpler. Although this is a controversial matter [35], some practitioners recommend to expose the API version in the URI path **(*Bp_VersionSelectionURI*** [2], [3], [12], [15], [16]**)**. Another point of view on this subject is that a version 2 API will, in general, refers to different resources from those used by a previous version 1 API, which means that the use of terms like v1 or v2 in the URI is, in practice, also providing information about the location of a certain resource (i.e., in such cases, the *Resource Addressability* principle may not be violated).

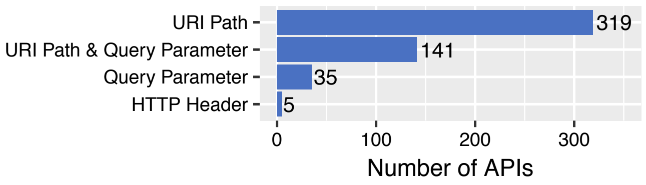

Fig. 4. The different ways of encoding scoping Information.

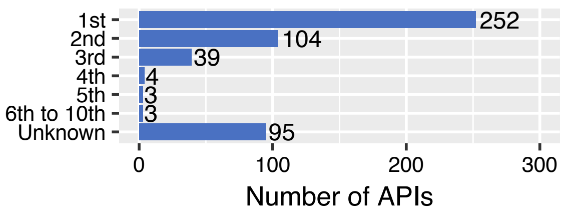

Fig. 5. Distribution of currently deployed API major versions.

Only about one fifth (20.6% – 103 APIs) of the 500 APIs **declare response messages as cacheable or not cacheable** (***Bp_Caching*** [3], [11], [13], [15], [17]). All of the former use the *HTTP Cache-Control* headers [36] for caching. The remaining 79.4% (397) do not provide any information about caching. Caching is directly supported by the HTTP protocol and can help reducing API calls, by keeping frequently needed response messages. As previously mentioned, the authors in [3] found that 27% of the analyzed APIs explicitly stated their support for caching. Further analysis over the remaining APIs using test invocations allowed to discover an additional 13% supporting caching (which may also occur in our case, but is out of the scope of this work). With direct support from HTTP, it is unclear why only 20.6% of the APIs state their support for caching or for not caching, in agreement with the *Cacheable principle*.

The last feature of the REST architecture dimension addresses the use of **links to related resources in response messages**, named by Fielding as *Hypermedia As The Engine Of Application State* (*HATEOAS*), another property of the *Uniform Interface (4)* principle [5] and also a REST service development best practice (***Bp_Hateoas*** [2]–[4], [4], [9], [11]–[13]). Such links provide clients with available actions, based on the clients' current state, which work as a navigation system throughout the service and relieves a client from calculating whether it is allowed to execute a specific operation or not [37]. For instance, in a file hosting service, a successful login response message would also contain links to possible next operations (e.g., downloading a file, logging off). Letting the client calculate possible operations is error prone and may lead to unnecessary attempts by the client to perform operations that will fail. We found only 4.2% of the APIs complying with *HATEOAS*. This apparently useful REST property is scarcely adopted, which opens space for discussion. Implementing *HATEOAS* obviously adds complexity to the server and client sides. Authors in [15] mention it may also require further standardization efforts.

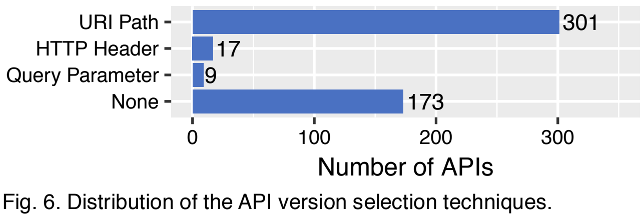

Fig. 6. Distribution of the API version selection techniques.

Overall, it is already clear that there is heterogeneity in the adoption of certain features, compliance with some of the REST principles, and adherence to best practices. In Section 5, we further discuss these findings and the ones presented in the following subsections.

4.2 HTTP use

In this section, we present the results regarding the use of HTTP verbs (e.g., GET, POST), how method overrides are implemented, and if adequate HTTP status codes are used. Fig. 7 shows the distribution (including overlaps) of the **HTTP verbs used** (we were unable to determine the supported verbs for 2 APIs). As we can see, about a third of the analyzed APIs allow GET as the only HTTP verb. 22% support only GET and POST and the third major combination involves 20.8% of the APIs, which support the four HTTP verbs GET, POST, PUT and DELETE. Not in the figure, but also present were PATCH (8.6%) and HEAD (3.2%).

Each HTTP verb should be used for the corresponding behavior (***Bp_HTTPVerbs*** [2]–[4], [9], [11]–[13], [15], [16]). GET, for instance, should be used for receiving resources and not for changing resources. If developers adhere to this kind of rules, API requests can more easily comply with the *self-descriptive messages* principle. Assuming that the server knows how to interpret the data (see the *Bp_ContentHeader* best practice), if it receives the resource name, location, and the right HTTP verb, then the server has all information needed to complete the operation.

We found out that most APIs (88%) use the HTTP verbs as originally intended, complying, in this sense, with the *self-descriptive messages* property. However, 12% of the APIs offer GET and also POST as the two only HTTP verbs, to support operations (e.g., updating and deleting resources) that should, according to the HTTP rules, not be performed with GET or POST. This mode of operation was also observed in previous work [3].

When not using the right HTTP verbs, to let the server know which operation to perform, the API allows a **method override**, which is in 9.4% carried out by specifying the method type as part of the URI path. This percentage breaks down to 5.0% for RPC-style URIs, because these always define the method information in the URI besides the resource information (e.g., *POST /deleteuser?user=andy*) and to 4.4% for REST-style URIs, where the method override is added after the resource path (e.g. *POST /user/andy/delete*). 2.6% specify the method override as a query parameter and the remaining 0.2% include the method override information in an HTTP header field. A more REST-conform way to allow interacting with clients that do not support all HTTP verbs (e.g., old browsers) would be to support all necessary HTTP verbs and then also allow method overrides with one verb, which is actually offered by 6 APIs (1.2%).

Another important aspect of the *self-descriptive messages* property is that each message (client request and server response) must contain information about how the transferred data must be interpreted (i.e., the **content type**). To comply with this property, the information about how to interpret should be encoded in the HTTP *content-type* header (***Bp_ContentHeader*** [2], [3], [9], [11]–[13], [15]). We analyzed the services and found out that only one fourth (24%) set the *content-type* header in messages appropriately, which does not comply with the *self-descriptive messages* property (although this property is highly supported when it comes to the adequate use of HTTP verbs, as discussed previously). Obviously, the information inside the application message is also important to fully understand compliance with the property, but this kind of analysis is application-specific and, thus, out of scope of this paper. The remaining 380 APIs (76%) do not provide insight about the header fields of the client/server messages.

We further investigated the adoption of HTTP features, by examining the APIs **usage of HTTP response codes**. The HTTP protocol has a defined status code vocabulary that specifies issues between server and client (or correct behavior), which can simplify the overall communication process, including error handling. Sending adequate HTTP response codes (***Bp_StatusCodes*** [2], [3], [11]–[13], [15], [16]) will make the server response message easily understandable by clients.

Of the 500 services, almost two thirds (64.6%) exchange adequate HTTP response status codes, which is aligned with the *self-descriptive messages* principle. Such services send HTTP status codes in the 2xx range, if the task is successfully completed. For Client or Server errors, a status code in the 4xx or 5xx range is returned. In 2016, the analysis of a large data set of API responses showed the richness of the use of status codes [9], pointing out an improvement regarding web services running in 2005. With nearly two thirds of our analyzed services also documenting their existing HTTP response codes, we highlight the high adoption of this feature by service developers. 2.8% send custom error codes inside the response body, which complicates the error handling process for clients. About one third (32.6%) do not provide insight on which (or if) HTTP status codes are used in responses.

4.3 I/O and Software Development Support

In this section, we analyze the input and output formats being used by the services. Fig. 8 shows the distribution of the supported formats, in which JSON leads (note that some APIs support more than one format). Encoding a representation of a resource in the request, to let the server manipulate the actual resource based on this, agrees with the *Manipulation of resources through representations* principle. This applies to 66.2% of the services, which are essentially services that allow data manipulation via the HTTP verbs POST, PUT and PATCH. This percentage breaks

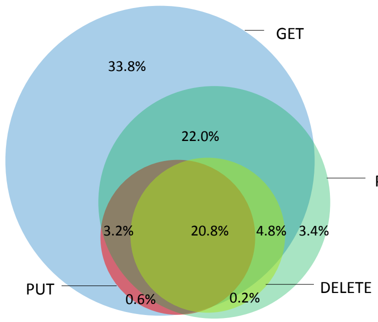

Fig. 7. API support of HTTP verbs.

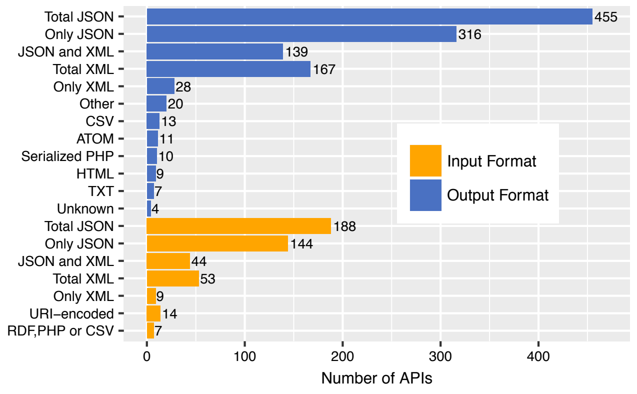

Fig. 8. Supported input and output formats.

down to 43.2% that send the resource representation encoded in the request body and 23% that only use the URI or query parameters to send resource representations. The remaining 33.8% only provide GET operations and do not allow resource manipulation.

Almost all APIs (99.2%) provide information about available **output formats**, with JSON nowadays leading as the most popular format by a clear difference to XML (the second most popular). In the past, developers argued that JSON´s main weakness is its lack of well-defined standards, which made them use XML [38], but the fact is that JSON has gained a lot of popularity [3], [8], [9], which may be explained by the growing importance of JavaScript runtime environments (e.g., Node.js) [8]. It seems like that this lightweight compact data format could completely replace XML in the future. We base this argument on the fact that nearly two thirds of the APIs (63.2%) support JSON without supporting XML, whereas only 5.6% support XML without supporting JSON. We also identified 14 other output formats in 20 APIs. These include TXT, RDF, PAM (Portable Arbitrary Map), RSS, HTML, JavaScript, PDF, TSV, XLS, TL-schema, TSV, VCF (Variant Call Format), QRCode and PNG.

Input and output formats are definitely linked to software development support, which usually comes in the form of **Software Development Kits (SDKs)** offered to client developers. SDKs are essentially programming language wrappers for sending API requests. They simplify application development by automating tasks that are error-prone or time-consuming for developers (e.g., parsing a complex JSON response or going through a complex authorization mechanism) [39]. The need for an SDK in the case of REST is much smaller, as the technology is fairly easy to use (i.e., REST messages are simple HTTP requests). Previous research suggests that developers prefer official SDKs over plain HTTP clients [19], but, at the same time, SDKs can also bring drawbacks as they can get out of synchronization with the actual API [22]. Of the 500 API documentations, 73% do not mention any SDKs, whereas 27% officially provide SDKs. Fig. 9 shows the distribution of the number of supported SDKs.

We further extended this analysis by crossing the SDK data with programming language popularity (as indexed by *tiobe.com* in July 2017). Fig. 10 shows the distribution of the programming language support of the 27% of APIs that are offering official SDKs (we only show languages that are part of Tiobe's top10 or are offered by at least 5% of the APIs).

As we can see in Fig. 10, PHP, Java, Python and Ruby are the most often supported languages, with support from about half of the APIs to almost two thirds (in the case of Java). Relatively popular options include C#, JavaScript, Objective-C, or Node.js. Notice also that, in some cases, the API documentation simply states .NET support and does not map it to specific .NET languages (e.g., C# or .NET Visual Basic), which may influence the final distribution in what concerns the .NET framework. The SDK results are, in general, aligned with observations from previous work

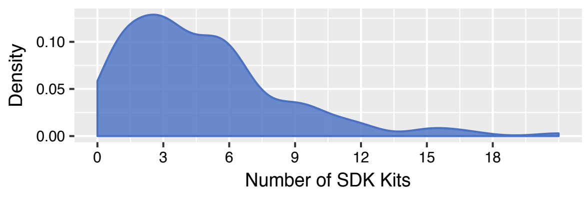

Fig. 9. Distribution of the number of supported SDKs.

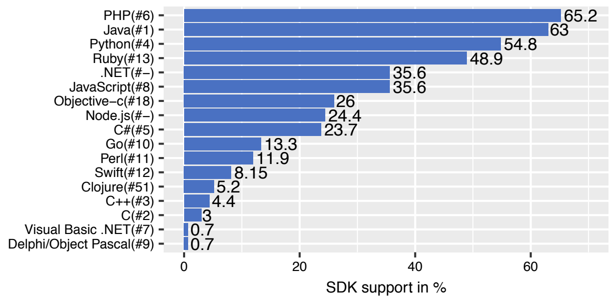

Fig. 10. SDK languages and Tiobe's language popularity.

[8], where Java, JavaScript, C#, Python and Ruby were also found to be often used, although the authors in [8] did not mention PHP.

4.4 Security Mechanisms

In this section, we present the results regarding the adoption of basic security mechanisms by the web APIs (and supporting infrastructure), namely user authentication mechanisms and SSL/TLS encryption support.

Most APIs (86%) in the set being analyzed require **user authentication** for API calls. Depending on the sensitivity of the data that is exchanged or on the purpose of the operation, some APIs require that the user is authenticated and authorized to execute for all operations, whereas others require authentication only for certain operations (e.g., operations that exchange user-specific data).

From the 500 analyzed APIs, 1.4% do not provide information about authentication mechanisms, 12.6% do not require any form of authentication, and 86% support at least one authentication mechanism with every 1 out of 10 supporting more than one authentication mechanism. Fig. 11 shows the distribution of the different authentication mechanisms. Excluded from the figure are: APIs that either do not require authentication or do not provide information about it; 7 APIs using an unknown authentication mechanism; and 25 APIs that are using 9 less common mechanisms, such as HMAC, 2FA, JWT, Session, Open ID, custom OAuth, Macaroon, Azure AD v2 and HTTP Cookie. Areas that intersect in Fig. 11 but have no associated percentage refer to no intersection at all.

In Fig. 11 we essentially have proprietary (API key and client credentials) and non-proprietary mechanisms (OAuth1.0, OAuth2.0, and HTTP Basic). The services using non-proprietary mechanisms (***Bp_StandardAuth*** [2]–[4], [11], [15]) add up to 42%, with the OAuth protocol (v1 and v2) being used by about one third of the APIs (32.6%), being the most commonly supported mechanism and we argue that it has the potential for becoming the most widely

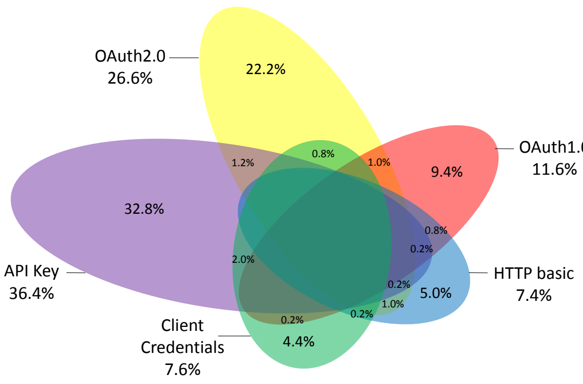

Fig. 11. Distribution of the different authentication mechanisms.

adopted standard for authentication. OAuth2.0 is currently used in 26.6% of the services and HTTP Basic, which is a standard that may suffice for some services, is used by 7.4% of the services. However, also 42% of the services are using proprietary authentication mechanisms (API key or client credentials), despite the open and standard specifications for authentication available, which are preferable options [2]–[4], [11], [15].

Using an encrypted connection is recommended for communications that exchange sensitive data [40]. While examining the APIs for **transport channel encryption** (HTTPS), we identified 3 cases: i) no support for HTTPS (9.2%); ii) mandatory HTTPS throughout all communications (61.2%); and iii) optional HTTPS (26.4%). For the remaining 3.2% we could not determine if SSL/TLS was supported. Offering (at least optionally) HTTPS is considered to be a best practice (***Bp_HTTPS*** [2], [15]).

When using the API involves authentication, the use of an encrypted connection is certainly a very important requirement, otherwise the authentication details can be easily harvested by attackers [41]. Thus, we analyzed if there are APIs that do not support HTTPS but still require user authentication to be performed. This resulted in a total of 5.0% (25 APIs), from which 4.0% use proprietary user authentication mechanisms based on the user's API Key, 0.6% identify the user by his account credentials, and 0.4% use an undetermined authentication mechanism. Of these services, a particular API even requires the user to pay a monthly fee for the API usage, starting at 8€. For the APIs using the non-proprietary authentication mechanisms, we found HTTPS to be either mandatory or optionally available.

4.5 Usage Policies

In this section, we present the results regarding the usage policies of the web APIs. This includes the form of registration (required to start using the API), if payment plans for the API usage are available or required, and if call limits are being enforced.

Most APIs (82.4% – 412 APIs) require some form of **service registration** for developers to start using the API. The distribution of the different types of usage registration is shown in Fig. 12. Registration enables the API provider to carry out several management functions, such as monitoring of API use, call limit enforcements or routing API requests to different servers [4]. Actually, 75.6% require developers to create a user account**,** which in some cases enables the generation of an API Key that needs to be passed with every API call (not represented in Fig. 12). In other cases, the account credentials are used by the authentication process to enable calls to the service. 6.8% of the analyzed APIs offer the creation of an API Key without requiring an account. 13.6% do not require any registration for using the API.

Regarding the **payment plans**, 76.4% offer a free-of-charge usage, even though most APIs require an account to make API calls. 7.4% offer optional payment plans that enable more features than the free-of-charge plans such as a higher call limit. 3.6% require a payment plan to get access to the API and the remaining 12.6% do not provide any further information regarding this matter. Authors in [21] analyzed 69 web services in 2017, finding out that nine out of ten services offered a free plan (six out of ten provided the ability to customize a plan). In our larger dataset, this free-service trend is also quite visible, despite slightly less accentuated (in 76.4% of the services).

Call limits enforce that a certain number of API calls are not exceeded during a defined period and allow the API provider to control the service usage, which is especially important for APIs that handle a huge number of calls [4]. Fig. 13 presents the results regarding the APIs call limits (normalized to daily limits).

More than half of the APIs (56.4%) provide information about the existence or absence of call limits (***Bp_RateLimit*** [2]–[4], [15]), either as part of the documentation or written down in the general terms and conditions, the remaining do not provide any information about applied call limits. 12.8% (64) state that they do not enforce any call limits, whereas 43.6% (218 APIs) state that they are enforcing call limits. 3.6% block user's API calls, if their algorithm detects some kind of API abuse. Their respective documentations do not advertise the exact call limits to discourage abuse. 11.8% enforce different call limits for different operations or provide call limits per user, device or application. As an example, the *Samsung Artik Cloud* web API has a daily call limit for GET requests of 1000 calls per user and per device, and 5000 calls per application. 2.2% of the documentations state that they are enforcing call limits, but do not describe how these limits are enforced. The remaining 26% enforce a fixed call limit.

Overall, the results show that the call limits encompass very different ranges. Authors in [21] also analyze the enforcement of call limits feature in services collected from *programmableweb.com* and *market.mashape.com*, to find respectively 59% and 88% of web services enforcing operation limitations, 21% and 4% enforcing functionality limitations, and 21% and 8% enforcing time limitations of the API usage. Thus, although there is some variability in the type of limitation, the trend is to actually place some kind of usage limit on the clients.

An often-used technique (which is present in 22% of the 218 APIs that enforce call limits) is the return of HTTP headers carrying information about the status of current call limits (***Bp_RateHeader*** [3], [15]) to allow the user or application to assess where it stands in terms of call limits. From what we observed, these headers typically provide the maximum number of allowed requests in the current period, the number of remaining requests in the current period and a timestamp that marks when the call limit counter will be reset. This kind of information enables dynamic adjustments of the API usage, which may be used to prevent the API user from being blocked.

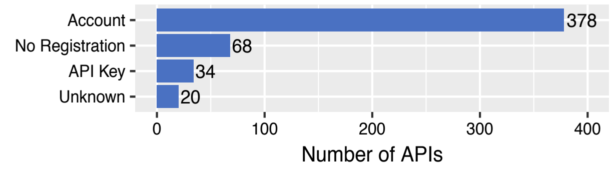

Fig. 12. Distribution of the different required registration forms.

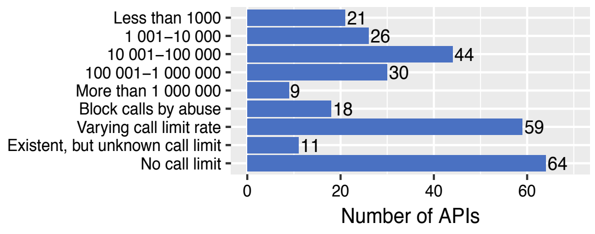

Fig. 13. Call limit enforcements.

4.6 Documentation and Application Use

We begin by reviewing the API documentation dates, to have an overall idea about the activity behind APIs. This section has then two main parts, in the first part we detail how developers are creating documentation for their services (i.e., by analyzing the use of documentation generation software) and in the second part we analyze the known participation of APIs in applications.

Considering that many times users can choose among alternative APIs, a user may prefer an API that is regularly maintained over an API whose documentation has been last updated several years ago, if both allow using the same service. We tried to analyze this in the documentation of the 500 APIs, to find that 73% of the APIs do not provide this information. The remaining 27% break down to 12% updated during 2017, 8.6% during 2016, and 6.4% between 2010 and 2015. So, within the 27% about half is providing documentation that has been modified during 2017, which suggests activity behind the API. Still, the dataset is very poor in this feature and cannot provide rich data for further discussion.

Documentation can be manually created or **generated by software**. There are several API development tools on the market, which help not only designing and testing an API, but also documenting it. This software can automatically generate documentation based on comments in the source code (e.g., Apidoc), based on API specifications, such as the OpenAPI specification (formerly Swagger [28]), or on API description languages such as API Blueprint (e.g., Apiary). Generated documentation can be also seen as an alternative to interface description documents, such as WSDL for SOAP web services.

Generated documentation (***Bp_GeneratedDoc*** [2], [15]) is being used in about half (45%) of the 500 web services, with documents generated by Swagger UI taking the lead. This assists typical web development tasks by, for instance, creating **interactive API test consoles** to enable testing API calls within the browser, which we found present in one third (33.6%) of the 500 APIs and in almost all of the Swagger UI and Apigee SmartDocs generated documentation. The distribution of the different software tools that generate documentation (and that generate test consoles) is shown in Fig. 14.

Although the growing trend of using documentation tools is relatively well-known [42], still 55% do not use software generated documentation and document their API in some textual form as part of a web page, which results in rather diverse descriptions of the API, in terms of structure, content, and level of detail and creating huge obstacles for automated analysis of the API documentation. Also, in 14.6% of the cases, we classified the documentation as a software generated documentation ("Unknown Software" in Fig. 14), because although they featured certain recognizable structures, we were unable to determine the exact software used.

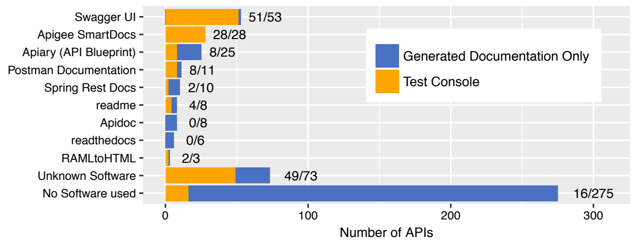

Fig. 14. Documentation tools used by the APIs.

These observations agree, in general, with the work in [4] (2014), where the authors emphasize the industry interest in Swagger. In our work, we found 10.6% of the analyzed APIs using Swagger UI and nearly half using software to generate the documentation (notice that, currently, one fourth of the tool generated documentations are created by Swagger UI), which are, for most cases, also compatible with the OpenAPI specification. This suggests some tendency to make the API documentation machine-readable and also understandable, which will also allow easier composition of different services as well as other automated processes (e.g., service recommendation) [10]. Still, for such kind of approach to work the documentation must be of good quality (even if generated). Previous research has discussed there are actually differences in the quality of Swagger specifications [43], which suggests the need for further standardization efforts.

Of the 500 API documentations, 90.4% provide **example API requests and responses**. These examples give the developer a quick insight on the use of the API, including URI structure, parameter selection, response structure, which simplifies development. Also, a total of 70.2% describe and **explain possible error messages** arising from the use of the API, whether these are simple text messages in the response body or HTTP status codes. The remaining 29.8% do not provide any information about error responses. When these error messages are not self-explanatory, the developer may not be able to know what went wrong or why exactly the error occurred and may not be able to trigger the correct error handling code.

Finally, we retrieved the **reported number of applications that use the APIs** as set in *programmableweb.com*, which is a repository that has been used in previous work [1]–[3], with this same purpose. About one third of the APIs (30.2%) are not listed in *programmableweb.com* and 41.8% of the APIs are associated with no applications. 16.4% are associated with 1 to 5 applications, 8.8% fall in the 6-50 range and 2.8% are used in more than 50 applications, with *Twitter.com* counting 830 applications, the highest number observed.

The abovementioned distribution shows that only a few APIs are often used in applications, with the majority hardly being used. As these values provided by *programmableweb.com* are based on user input, the true exact numbers might be different, as services are added to this kind

TABLE 1. APIs REPORTEDLY USED BY APPLICATIONS AND MAIN FEATURES

Range	Total APIs	URI scheme		Documentation Tool		Output Format			Authentication mechanism		
		REST	RPC	Yes	No	JSON	XML	Both / Other	OAut h	Prop.	Both / Other
1 to 5	82	84%	16%	50%	50%	55%	4%	41%	41%	38%	21%
6-50	44	86%	14%	68%	32%	59%	5%	36%	52%	30%	18%
>50	14	86%	14%	71%	29%	57%	7%	36%	86%	7%	7%

of repositories but rarely, or never, removed. However, the values are still indicators of some level of reusability of the APIs or at least of their popularity. We further analyzed this data against some of the main features provided by the APIs. Table 1 summarizes the results.

As we can see in Table 1, services using a REST-style URI scheme and producing output in JSON (marked in green in Table 1) are present in all three ranges and show little or no fluctuation. Moreover, their values are clearly dominant in each of the two respective features. We also see dominance in the use of documentation generation software and in the use of OAuth. In these cases (marked in orange in Table 1) the growing trend is also very clear, with services that are being used by more applications having a greater likelihood of also using a documentation generation software and OAuth. We also verified that none of the popular APIs were using HATEOAS, which further emphasizes the unpopularity of this REST principle.

5 Discussion

In this section, we show an overall view of the results, discussing: i) level of agreement of the services with the REST principles; ii) common decisions regarding the features discussed in this paper; iii) observed compliance with best practices.

In Fig. 15 we show the observed **compliance with the REST principles (RQ1)**. We excluded *Stateless*, *Layered System*, and the optional *Code on Demand* principle from the figure, due to the related inconclusive data found.

Overall, there is visible heterogeneity regarding the adoption of REST principles. Some are highly adopted (but also easy to comply with, such as *Client-Server*) while others are barely adopted (e.g., *Uniform Interface*). The low adoption of certain principles or properties, such as *Cacheable* or *HATEOAS* is in line with previous work, in which similar findings were discussed [2], [3], [9], [10]. The *Uniform Interface* principle gathers a number of properties, which show high variability (e.g., *Manipulation of Resources through Representations* and *HATEOAS)*.

Overall, we found out that only four (0.8%) of the analyzed services fully comply with the REST principles. These services are provided by *pearson.com*, *github.com*, *reverb.com*, and *familysearch.org*, with the adoption of *HATEOAS* being highly responsible for such a low number. The Richardson Maturity Model [18] is also a way of understanding principle compliance. In this model, Level 0 services just use a single endpoint (e.g., SOAP services); Level 1 distribute functionality across several resources, but services still need to use message data or the URL to know which operation is to be invoked; Level 2 services use HTTP methods and status codes adequately; and Level 3 apply HATEOAS. We identified no Level 0 services; 12.2% Level 1; 60.2% Level 2; and 4% Level 3, which is, in general, aligned with the results presented in [9].

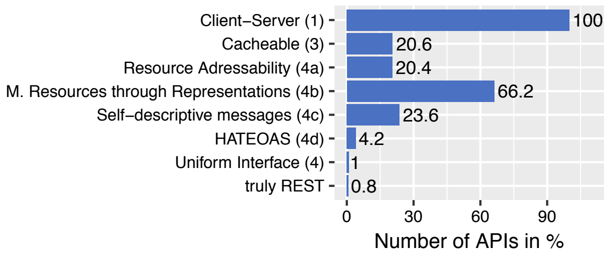

Fig. 15. Overall compliance with the REST principles.

Fig. 16 presents the most **common decisions (RQ2)** per feature. As we can see, in what concerns this aspect, developers lean quite strongly towards a decision in about one third (8 features) where we observed the same feature value in more than 75% of the services. If we lower this threshold to 50%, then we find about two thirds of the features holding the same value. A few features are scarcely adopted (e.g., HATEOAS and URI Path Method Override), with some being close to the 50% threshold, such as "No Format Selection Necessary" or "Deployment of First Major Version".

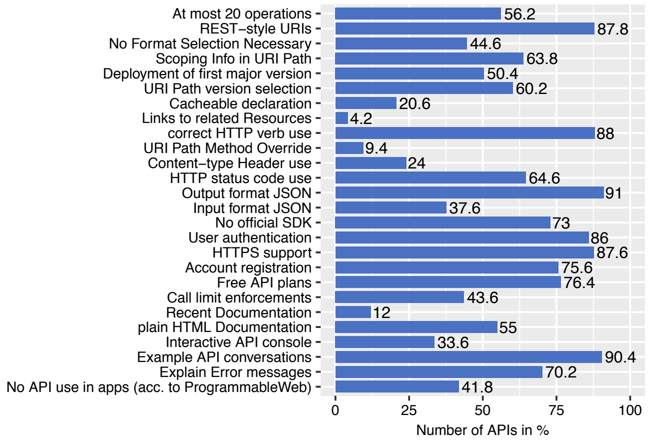

Fig. 16. Most commonly observed features.

If we look more closely to the second choices (not visible in Fig. 16, as they are not top results) and drill down to the details, we find a few interesting common values. In particular, the growing use of software generated documentation, in line with what was mentioned in [42], mostly refers to the use of Swagger UI. Service output format selection, when available, is done mostly using HTTP headers. Among the APIs offering official SDKs, PHP is taking the lead as the top language supported and API key is the most popular authentication mechanism. Finally, when the service requires registration, the most often scheme used is the creation of an account. This global image, discussed here, allows us to describe what currently a REST service tends to be, and, overall, we see strong agreement in the application of two thirds of the analyzed features.

Fig. 17 presents the **adherence to REST service development best practices (RQ3)** observed in the 500 services**.** We found almost half (7) of the best practices being followed by more than half of the services. The remaining are split in a group of 3 practices, followed by almost half of the services, and in another group followed by at most one fourth of the services, with *Bp_Hateoas* showing residual numbers. In the end, there is no single service following all

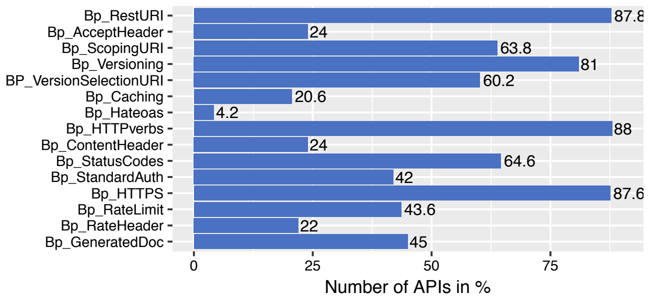

Fig. 17. Adherence to REST service development best practices.

best practices. Previous work stresses the fact that guidelines for building web services can be very fine-grained, while others rather simple [2], [13], [14]. This may explain the heterogeneity observed (including the very low numbers for some practices), but especially stresses the diversity present in services in the field.

Future research could, based on the state of the practice discussed in this paper, focus on several areas. These range from coarse grained research to fine grained research activities, of which we emphasize: i) evolving the set of REST principles towards a second generation; ii) proposing best practices based on new principles and on common design/implementation decisions found in the field; iii) understanding how REST service development tools (e.g., the Swagger toolset) affect compliance with principles or best practices (and the real level of developer adoption regarding provided SDKs); iv) quantifying the practical effect of not following certain practices on the dependability properties of services (e.g., robustness, maintainability). Although REST services are, by now, a very popular way of deploying services, there are still numerous challenges that relate to the diversity observed in the deployments, leaving space for researchers and practitioners to improve the state of the practice.

6 Threats to validity

In this section, we present threats to the validity of this work and discuss mitigation strategies. Regarding external validity, it is important to mention that the analysis of REST services performed in this work cannot be generalized to all available web services, as it is based on a *limited set of services*. This threat cannot be avoided, but we tried to mitigate it by carrying out a large-scale study of 500 service APIs. *The set of APIs was defined based on popularity of the hosting web site*, which may exclude some popular APIs associated with less popular web sites. Also, the random *selection of a single web service* when a provider offered more than one could add some bias to the results. At the same time, knowing that some providers offer huge amounts of services (e.g., Google offers 100 different web services) would add bias towards the design of Google web services, which we decided to avoid in favor of a more general view on current public web services.

Regarding internal validity, it is relevant to point out that we *verified a subset of the available endpoints* in the case of APIs providing a large number of endpoints, as it was unfeasible to verify all. This may add error to our results, but we did however verify half of the endpoints (for APIs with 20 to 50 endpoints) and one quarter (for APIs with more than 50 endpoints). The verified subset showed strong consistency, which suggests that the non-verified endpoints will not change the results significantly.

In this work, we *only analyzed the services documentation* and did not perform actual invocations to prove it was correct. Thus, the documentation might be outdated, incorrect, or incomplete (e.g., not stating that a service supports caching), but using service invocations would go beyond the scope of the work. Being highly popular sites, we believe that the interest in keeping the API documentation correct should be quite high. *The work performed here was entirely manual*, due to the lack of structure of the APIs documentation. Due to the huge amount of effort involved, we verified just part of the results, which we estimate reached one fourth of the APIs, to find only residual mistakes. Still, we acknowledge that the results might hold some residual error, due to the human intervention in the process.

7 Conclusion

In this work, we went through the *alexa.com* top 4000 sites to find precisely 500 sites claiming to provide at least one REST web service API, and 22 services built based on other technologies (e.g., SOAP). We went through the REST services, analyzing a total of 26 different features, mostly gathered in the literature, to understand the level of compliance with the REST principles, common developer decisions, and adherence to REST service development best practices.

The knowledge brought in by this work, namely understanding the state of the practice, can provide useful research directions for the academy (e.g., understanding how REST development tools affect compliance with principles or best practices) and useful guidelines for practitioners (e.g., commonly used user authentication mechanisms). As future work, we intend to focus on the challenges identified in Section 5, starting with the empirical analysis of REST development tools.

Acknowledgements

This work has been partially supported by the project ATMOSPHERE, funded by the European Commission under the Cooperation Programme, Horizon 2020 grant agreement no 777154.

References

[1] M. Maleshkova, C. Pedrinaci, and J. Domingue, "Investigating Web APIs on the World Wide Web," in 2010 Eighth IEEE European Conference on Web Services, 2010, pp. 107–114.

[2] D. Renzel, P. Schlebusch, and R. Klamma, "Today's Top 'RESTful' Services and Why They Are Not RESTful," in Web Information Systems Engineering - WISE 2012, 2012, pp. 354–367.

[3] F. Bülthoff and M. Maleshkova, "RESTful or RESTless – Current State of Today's Top Web APIs," in The Semantic Web: ESWC 2014 Satellite Events, 2014, pp. 64–74.

[4] J. Kopecký, P. Fremantle, and R. Boakes, "A history and future of Web APIs," It - Inf. Technol., vol. 56, 2014.

[5] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Univ. of California, Irvine, 2000.

[6] R. Battle and E. Benson, "Bridging the semantic Web and Web 2.0 with Representational State Transfer (REST)," Web Semant. Sci. Serv. Agents World Wide Web, vol. 6, no. 1, pp. 61–69, Feb. 2008.

[7] B. Costa, P. F. Pires, F. C. Delicato, and P. Merson, "Evaluating a Representational State Transfer (REST) Architecture: What is the Impact of REST in My Architecture?," in 2014 IEEE/IFIP Conference on Software Architecture, 2014, pp. 105–114.

[8] G. Schermann, J. Cito, and P. Leitner, "All the Services Large and Micro: Revisiting Industrial Practice in Services Computing," in Service-Oriented Computing – ICSOC 2015 Workshops, 2015, pp. 36–47.

[9] C. Rodríguez et al., "REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices," in Web Engineering, vol. 9671, A. Bozzon, P. Cudre-Maroux, and C. Pautasso, Eds. Cham: Springer International Publishing, 2016, pp. 21–39.

[10] F. Haupt, F. Leymann, A. Scherer, and K. Vukojevic-Haupt, "A Framework for the Structural Analysis of REST APIs," in Software Architecture (ICSA), 2017 IEEE International Conference on, 2017, pp. 55–58.

[11] Mark Masse, REST API Design Rulebook. 2011.

[12] S. Jauker, "10 Best Practices for Better RESTful API | Thinking Mobile," 05-Jun-2014. [Online]. Available: https://blog.mwaysolutions.com/2014/06/05/10-best-practices-for-better-restful-api/. [Accessed: 27-Mar-2017].

[13] F. Petrillo, P. Merle, N. Moha, and Y.-G. Guéhéneuc, "Are REST APIs for Cloud Computing Well-Designed? An Exploratory Study," in Service-Oriented Computing, 2016, pp. 157–170.

[14] F. Palma, J. Gonzalez-Huerta, M. Founi, N. Moha, G. Tremblay, and Y.-G. Guéhéneuc, "Semantic Analysis of RESTful APIs for the Detection of Linguistic Patterns and Antipatterns," Int. J. Coop. Inf. Syst., vol. 26, no. 02, p. 1742001, May 2017.

[15] V. Sahni, "Best Practices for Designing a Pragmatic RESTful API," Vinay Sahni. [Online]. Available: http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api#hateoas. [Accessed: 22-Dec-2017].

[16] M. Haldar, "RESTful API Designing guidelines — The best practices," Hacker Noon, 03-Feb-2017. [Online]. Available: https://hackernoon.com/restful-api-designing-guidelines-the-best-practices-60e1d954e7c9. [Accessed: 24-Jun-2017].

[17] D. Bermbach and E. Wittern, "Benchmarking Web API Quality," in Web Engineering, 2016, pp. 188–206.

[18] M. Fowler, "Richardson Maturity Model," martinfowler.com. [Online]. Available: https://martinfowler.com/articles/richardsonMaturityModel.html. [Accessed: 27-Jun-2017].

[19] M. A. Oumaziz et al., "Empirical Study on REST APIs Usage in Android Mobile Applications," in Service-Oriented Computing, 2017, pp. 614–622.

[20] A. Ivanchikj, C. Pautasso, and S. Schreier, "Visual modeling of RESTful conversations with RESTalk," Softw. Syst. Model., pp. 1–21, May 2016.

[21] A. Gamez-Diaz, P. Fernandez, and A. Ruiz-Cortes, "An Analysis of RESTful APIs Offerings in the Industry," in Service-Oriented Computing, 2017, pp. 589–604.

[22] E. Wittern et al., "Opportunities in software engineering research for web API consumption," in Proceedings of the 1st International Workshop on API Usage and Evolution, 2017, pp. 7–10.

[23] M. Seraj, "We Create, We Connect, We Respect, Therefore We Are: Intellectual, Social, and Cultural Value in Online Communities," J. Interact. Mark., vol. 26, no. 4, pp. 209–222, Nov. 2012.

[24] J. W. Patchin and S. Hinduja, "Changes in adolescent online social networking behaviors from 2006 to 2009," Comput. Hum. Behav., vol. 26, no. 6, pp. 1818–1821, Nov. 2010.

[25] M. Bakaev, V. Khvorostov, S. Heil, and M. Gaedke, "Web Intelligence Linked Open Data for Website Design Reuse," in Web Engineering, 2017, pp. 370–377.

[26] H. Singal and S. Kohli, "Trust Necessitated through Metrics: Estimating the Trustworthiness of Websites," Procedia Comput. Sci., vol. 85, pp. 133–140, Jan. 2016.

[27] M. R. Fedorko, "Search Engine Marketing (SEM)–The current state of the issue," 2014.

[28] SmartBear Software, "Swagger Specification." [Online]. Available: http://swagger.io/specification/. [Accessed: 18-Apr-2017].

[29] A. Neumann, N. Laranjeiro, and J. Bernardino, "API Analysis Detailed Results," 06-Aug-2017. [Online]. Available: http://eden.dei.uc.pt/~cnl/papers/2017-tsc.zip.

[30] Y. Li, Y. Liu, L. Zhang, G. Li, B. Xie, and J. Sun, "An Exploratory Study of Web Services on the Internet," in IEEE International Conference on Web Services (ICWS 2007), 2007, pp. 380–387.

[31] N. Masashi, "API design guidance." [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design. [Accessed: 23-Jan-2018].

[32] J. Li, Y. Xiong, X. Liu, and L. Zhang, "How Does Web Service API Evolution Affect Clients?," in 2013 IEEE 20th International Conference on Web Services, 2013, pp. 300–307.

[33] M. Fokaefs and E. Stroulia, "WSDarwin: Studying the Evolution of Web Service Systems," in Advanced Web Services, A. Bouguettaya, Q. Z. Sheng, and F. Daniel, Eds. Springer New York, 2014, pp. 199–223.

[34] S. Wang, I. Keivanloo, and Y. Zou, "How Do Developers React to RESTful API Evolution?," in Service-Oriented Computing, 2014, pp. 245–259.

[35] G. Levin, "RESTFul API Versioning Insights," REST API and Beyond, 21-Aug-2016. [Online]. Available: http://blog.restcase.com/restful-api-versioning-insights/. [Accessed: 24-Jun-2017].

[36] K. Lange, "Boost Your REST API with HTTP Caching," Kenneth Lange. [Online]. Available: http://www.kennethlange.com/posts/Boost-Your-REST-API-with-HTTP-Caching.html. [Accessed: 22-Dec-2017].

[37] O. Liskin, L. Singer, and K. Schneider, "Teaching Old Services New Tricks: Adding HATEOAS Support As an Afterthought," in Proceedings of the Second International Workshop on RESTful Design, New York, NY, USA, 2011, pp. 3–10.

[38] R. Zazueta, "API Data Exchange: XML vs. JSON | Mashery," 23-Jan-2014. [Online]. Available: https://www.mashery.com/blog/api-data-exchange-xml-vs-json. [Accessed: 30-Mar-2017].

[39] B. Mulloy, "RESTful API Design: complement with an SDK | Apigee." [Online]. Available: https://apigee.com/about/blog/developer/restful-api-design-complement-sdk. [Accessed: 18-Jan-2018].

[40] M. B. Juric, I. Rozman, B. Brumen, M. Colnaric, and M. Hericko, "Comparison of performance of Web services, WS-Security, RMI, and RMI–SSL," J. Syst. Softw., vol. 79, no. 5, pp. 689–700, May 2006.

[41] D. Akhawe, A. Barth, P. E. Lam, J. Mitchell, and D. Song, "Towards a Formal Foundation of Web Security," in 2010 23rd IEEE Computer Security Foundations Symposium, 2010, pp. 290–304.

[42] R. Rodríguez, R. Espinosa, D. Bianchini, I. Garrigós, J.-N. Mazón, and J. J. Zubcoff, "Extracting Models from Web API Documentation," in Proc. of the 12th Int. Conference on Current Trends in Web Engineering, Berlin, Heidelberg, 2012, pp. 134–145.

[43] M. Vaziri, L. Mandel, A. Shinnar, J. Siméon, and M. Hirzel, "Generating Chat Bots from Web API Specifications," in Proc. ACM SIGPLAN Int. Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, 2017, pp. 44–57.

Andy Neumann is a European Computer Science student at ISEC - Polytechnic Institute of Coimbra, Portugal. His former two years of studies were completed at the HAW University, Hamburg. He is a researcher at CISUC - Centre for Informatics and Systems of the University of Coimbra. His interests include web services, programming and embedded systems.

Nuno Laranjeiro received the PhD degree from the University of Coimbra, where he currently is an Assistant Professor. His research interests include dependability of software services, web services interoperability, and services security. He has authored more than 50 papers in refereed conferences and journals in the dependability and services computing areas.

Jorge Bernardino is a Coordinator Professor at the Polytechnic of Coimbra - ISEC, Portugal. He received the PhD degree from the University of Coimbra. He has authored over 100 publications in refereed conferences and journals. He was President of ISEC from 2005 to 2010. During 2014 he was Visiting Professor at CMU. Currently, he is President of ISEC Scientific Council and IEEE member.