REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices

Carlos Rodríguez[1](✉), Marcos Baez[1], Florian Daniel[2], Fabio Casati[1], Juan Carlos Trabucco[3], Luigi Canali[3], and Gianraffaele Percannella[3]

[1] University of Trento, Povo, TN, Italy
{crodriguez,baez,casati}@disi.unitn.it
[2] Politecnico di Milano, Milan, Italy
florian.daniel@polimi.it
[3] Telecom Italia, Trento, Italy
{juancarlos.trabucco,luigi.canali,
gianraffaele.percannella}@telecomitalia.it

Abstract. Quickly and dominantly, REST APIs have spread over the Web and percolated into modern software development practice, especially in the Mobile Internet where they conveniently enable offloading data and computations onto cloud services. We analyze more than 78 GB of HTTP traffic collected by Italy's biggest Mobile Internet provider over one full day and study how big the trend is in practice, how it changed the traffic that is generated by applications, and how REST APIs are implemented in practice. The analysis provides insight into the compliance of state-of-the-art APIs with theoretical Web engineering principles and guidelines, knowledge that affects how applications should be developed to be scalable and robust. The perspective is that of the Mobile Internet.

Keywords: REST · APIs · REST principles · Mobile internet

1 Introduction

By now, Web applications leveraging on remote APIs or services, service-oriented applications or service compositions [21], mashups [5], mobile applications built on top of cloud services and similar web technologies are state of the art. They all have in common the heavy use of functionality, application logic and/or data sourced from the own backend or third parties via Web services or APIs that provide added value and are accessible worldwide with only little development effort. The continuous and sustained growth of ProgrammableWeb's API directory (http://www.programmableweb.com/apis/directory) is only the most immediate evidence of the success that Web services and APIs have had and are having among developers. On the one hand, today it is hard to imagine a Web application or a mobile app that does not leverage on some kind of remote resource, be it a Google Map or some application-specific, proprietary functionality. On the other hand, to some companies today service/API calls represent the equivalent of page visits in terms of business value.

A. Bozzon et al. (Eds.): ICWE 2016, LNCS 9671, pp. 21–39, 2016.
DOI: 10.1007/978-3-319-38791-8_2

Two core types of remote programming resources have emerged over the years: SOAP/WSDL Web services [21] and REST APIs [6]. While the former can rely on a very rich set of standards and reference specifications, and developers know well how to use WSDL [4] to describe a service and SOAP [3] to exchange messages with clients, REST APIs do not have experienced this kind of standardization (we specifically refer to JSON/XML APIs for software agents and exclude web apps for human actors). Indeed, REST is an architectural style and a guideline of how to use HTTP [7] for the development of highly scalable and robust APIs. While the freedom left by this choice is one of the reasons for the fast uptake of REST, it is also a reasons why everybody interprets REST in an own way and follows guidelines and best practices only partially, if at all.

It goes without saying that even small differences in the interpretation of the principles and guidelines underlying REST APIs can turn into a tedious and intricate puzzle to the developer that has to integrate multiple APIs that each work differently, although expected to behave similarly. For instance, while one provider may accompany an own API with a suitable WADL [10] description, another provider may instead not provide any description at all and require interested clients to navigate through and explore autonomously the resources managed by the API. Of course, if instead all APIs consistently followed the same principles and guidelines, this would result in design features (e.g., decoupling, reusability, tolerance to evolution) that would directly translate into savings in development and maintainance costs and time [18,23].

With this paper, we provide up-to-date insight into how well or bad the principles and guidelines of the REST architectural style are followed by looking at the problem from the mobile perspective. We thus take an original point of view: we analyze more than 78 GB of plain HTTP traffic collected by Italy's biggest Mobile Internet (MI) provider, Telecom Italia, identify which of the individual HTTP calls are targeted at REST APIs, and characterize the usage patterns that emerge from the logged data so as to compare them with guidelines and principles. We further use the maturity model by Richardson [8], which offers an interesting way to look at REST in increasing levels of architectural gains, to distinguish different levels of compliance with the principles. The dataset we can rely on allows us, at the same time, to look at how conventional Web applications leverage on REST APIs as well as to bring in some insights regarding the use of APIs in the Mobile Internet. Concretely, the contributions of this paper are as follows:

- We descriptively characterize a *dataset* of more than 78 GB of HTTP requests corresponding to one full day of Mobile Internet traffic generated by almost 1 million subscribers.
- From the core principles and guidelines of REST and the structure of the dataset, we derive a set of *heuristics* and *metrics* that allow us to quantitatively describe the API ecosystem that emerges from the data.
- We *analyze* the results, study how well the data backs the principles and guidelines of REST, and discuss how the respective findings may impact API maintainability and development.

The paper is structured in line with these contributions. We first recap the theoretical principles and guidelines that we want to study in this paper (Sect. 2). Next, we introduce the dataset we analyzed and how we collected it (Sect. 3) and discuss its key features (Sect. 4). Then, we specifically focus on the REST APIs (Sect. 5) and conclude the paper with an overview of related works and our final considerations on the findings (Sects. 6 and 7).

2 REST APIs

The Representational State Transfer (REST) architectural style [6] defines a set of rules for the design of distributed hypermedia systems that have guided the design and development of the Web as we know it. Web services following the REST architectural style are referred to as *RESTful Web services*, and the programmatic interfaces of these services as *REST APIs*. The principles governing the design of REST APIs are in big part the result of architectural choices of the Web aimed at fostering scalability and robustness of networked, resource-oriented systems based on HTTP [7]. The core principles are [6,23]:

- *Resource addressability.* APIs manage and expose resources representing domain concepts; each resource is uniquely identified and addressable by a suitable Uniform Resource Identifier (URI).
- *Resource representations.* Clients do not directly know the internal format and state of resources; they work with resource representations (e.g., JSON or XML) that represent the current or intended state of a resource. The declaration of content-types in the headers of HTTP messages enables clients and servers to properly process representations.
- *Uniform interface.* Resources are accessed and manipulated using the standard methods defined by the HTTP protocol (`Get`, `Post`, `Put`, etc.). Each method has its own expected, standard behavior and standard status codes.
- *Statelessness.* Interactions between a client and an API are stateless, meaning that each request contains all the necessary information to be processed by the API; no interaction state is kept on the server.
- *Hypermedia as the engine of state.* Resources as domain concepts can be related to other resources. Links between resources (included in their representations) allow clients to discover and navigate relationships and to maintain interaction state.

Together, these principles explain the name "representational state transfer": interaction state is not stored on the server side; it is carried (transferred) by each request from the client to the server and encoded inside the representation of the resource the request refers to.

2.1 Best Practices for Development

Along with the general principles introduced above, a set of implementation best practices have emerged to guide the design of quality APIs [16,19,22,23].

These best practices address the main design aspects in REST APIs: (i) the modeling of resources, (ii) the identification of resources and the design of resource identifiers (URIs), (iii) the representation of resources, (iv) the definition of (HTTP) operations on resources, and (v) the interlinking of resources. We overview these best practices in the following; a summary with examples is shown in Table 1.

Resource modeling. REST APIs can manage different types of resources: *documents* for single instances of resources, *collections* for groups of resources, and *controllers* for actions that cannot logically be mapped to the standard HTTP methods [16]. While modeling resources for REST APIs is not fundamentally different from modeling classes in OO programming or entities in data modeling, there are a couple of recommended naming practices that are typical of REST APIs: singular nouns for documents, plural nouns for collections, and verbs only for controllers [16], no CRUD names in URLs [16,22], no transparency of server-side implementation technologies (e.g., PHP, JSP) (http://www.ibm.com/developerworks/library/ws-restful/).

Resource identification. Resource identifiers should conform with the URI format, consisting of a scheme, authority, path, query, and fragment [2]. In the case of Web-accessible REST APIs, the URIs are typically URLs (Uniform Resource Locators) that tell clients how to locate the APIs. In order to improve the readability of URLs, it is recommended to use hyphens instead of underscores, lowercase letters in paths, "api" as part of the domain, and avoid the trailing forward slash [16]. In addition, in its purest form, REST services should avoid declaring API versions in the URL [16].

Resource representation. Resources can support alternative representations (e.g., XML, JSON) and serve different clients with different formats. Which representation to serve should be negotiated at runtime, with the client expressing its desired representation using the HTTP `Accept` header instruction. This fosters reusability, interoperability and loose-coupling [22]. APIs should therefore use content negotiation instead of file extensions to specify formats (e.g., `.json` or `.xml`). In addition, it is recommended that APIs support (valid) JSON among their representation alternatives [16,22].

Operations. To manage resources, REST APIs should rely on the uniform set of operations (`Post`, `Get`, `Put`, `Delete`, `Options`, `Head`) defined by the HTTP standard [7] and comply with their standardized semantics:

– `Post` should be used to create new resources within a collection.
– `Get` should be used to retrieve a representation of a resource.
– `Put` should be used to update or create resources.
– `Delete` should be used to remove a resource from its parent.
– `Options` should be used to retrieve the available interactions of a resource.
– `Head` should be used to retrieve metadata of the current state of a resource.

REST APIs should thus never tunnel requests through `Get` or `Post`, e.g., by specifying the actual operation as a parameter or as part of the resource name.

Table 1. REST API design best practices with compliance (✔) and violations (✖)

Resource modeling

Singular noun for documents, plural noun for collections, verb for controllers, avoid CRUD names in URIs, and hide technology:

✔ http://api.test.org/universities

✖ http://api.test.org/university/deleteCenter?id=1

Resource identification

Use hyphens instead of underscores, lowercase letters in paths, and avoid the trailing forward slash:

✔ http://api.test.org/universities/12/faculty-centers?page=1

✖ http://api.test.org/universities/12/Faculty_centers/

Resource representation

Content negotiation instead of file extensions to specify desired formats, support (valid) JSON format among the representation alternatives:

✔ `GET` http://api.test.org/universities
`Accept: application/json`

✖ `GET` http://api.test.org/universities.json

Operations

Avoid tunneling requests through `Get` and `Post` and instead make standard use of the methods:

✔ `DELETE` http://api.test.org/universities/1
`Status 204`

✖ `GET` http://api.test.org/api?action=delete\&target=university\&id=1

Hyperlinks

Links should not be constructed by clients but obtained from the resource representation, they should follow a consistent structure and be sensitive to the current state of the resource:

✔ `GET` http://api.test.org/universities/1

```
  Accept: application/json
< {"name" : "UniTN",
<   "links" : { "faculty-centers" : "/universities/1/faculty-centers" }
  }
```

✖ `GET` http://api.test.org/universities/1

```
  Accept: application/json
< { "name" : "UniTN" }
```

2.2 Assessing REST Compliance

Next to the lower-level development best practices, concrete APIs may follow the very principles underlying REST to different extents. The maturity model by Richardson [8] offers a way to explain the respective degree of compliance by means of different levels of maturity:

- *Level 0*: At this level, APIs work by *tunneling* requests through a single endpoint (URL) using one HTTP method. Examples of services working at this level are XML-RPC and those SOAP/WSDL services that transmit all communications as HTTP `Post` requests and use HTTP purely as transport protocol. Yet, also some REST APIs adopt this technique.
- *Level 1*: At this level, instead of using a single endpoint, functionality exposed by the API is split over *multiple resources*, which increases the addressability of the API and facilitates consumption. However, services at Level 1 still make use of payload data or the URL to identify operations.
- *Level 2*: APIs at this level make proper use of the *HTTP methods* and *status codes* for each resource and correctly follow the uniform interface principle.
- *Level 3*: APIs at this level embrace the notion of *hypermedia*. Thus, not only resources can be accessed through a uniform interface but their relationships can be discovered and explored via suitable links.

Each level of compliance comes with greater benefits in terms of quality and ease of use by the developer familiar with REST. We will come back to these levels when analyzing the adherence of APIs to the principles and best practices.

3 Mobile Telco Infrastructure and Dataset

In order to study how well the state-of-the-art landscape of REST APIs complies with the introduced principles and guidelines, in this paper we rely on a dataset of 78 GB of plain HTTP traffic collected by Italys biggest Mobile Internet (MI) provider, Telecom Italia. To understand the nature and provenance of the dataset, Fig. 1 provides a functional overview of the underlying cellular network architecture (upper part) and of how data was collected (lower part).

The cellular network uses 2G (GSM/GPRS), 3G (UMTS) and 4G (LTE) base stations (Node B) for the connection of mobile devices. The Radio Network Controllers (RNCs) control the base stations and connect to the Serving GPRS Support Nodes (SGSNs) that provide packet-switched access to the core network of the operator within their service areas. Via the core network, the SGSNs are connected with the Gateway GPRS Support Nodes (GGSNs) that mediate between the core network of the operator and external packet-switched networks, in our case the Internet. The GGSNs also assign the IP addresses to the devices connected to the Internet through the operator's own network.

If a mobile device issues an HTTP request to a server accessible over the Internet, the request traverses all the described components from left to right. Special hardware probes tap into the connection between the SGSN and the GGSN to intercept raw traffic. The probes forward the traffic to multiple, parallel data collectors that filter the intercepted data by purpose (we specifically focus on network usage and HTTP traffic) and produce purpose-specific log files as output; each file contains approximately 15 min of traffic. For our analysis,

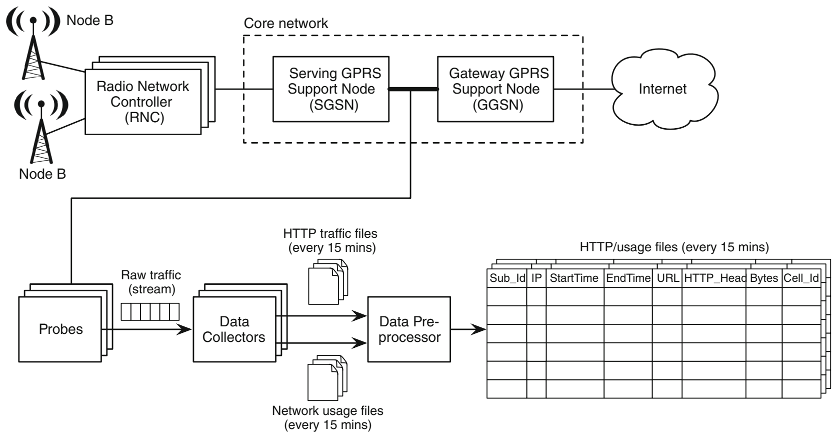

Node B
Node B
Radio Network Controller (RNC)
Core network
Serving GPRS Support Node (SGSN)
Gateway GPRS Support Node (GGSN)
Internet
Probes
Raw traffic (stream)
Data Collectors
HTTP traffic files (every 15 mins)
Network usage files (every 15 mins)
Data Pre-processor
HTTP/usage files (every 15 mins)
Sub_Id
IP
StartTime
EndTime
URL
HTTP_Head
Bytes
Cell_Id

Fig. 1. Cellular network architecture with probes for the collection of Mobile Internet usage data and an excerpt of the structure of the data studied in this article.

a pre-processing of the files is needed to join the HTTP traffic records with the network usage records, so as to be able to correlate traffic with network usage properties like cell IDs or data sizes.

The result is a set of joint, enriched HTTP traffic files of which Fig. 1 shows an excerpt of the data structure: Sub_Id and IP are the subscriber identifier and IP address (both fully anonymized), StartTime and EndTime delimit the HTTP transaction as registered by the cellular network, URL contains the complete URL requested by the mobile device, HTTP_Head contains the full header of the HTTP request, Bytes contains the size of the data uploaded/downloaded, and Cell_Id uniquely identifies the base station the device was connected to.

The available dataset was collected throughout the full day of 14 October (Wednesday) by one data collector located in the metropolitan area of Milan, Italy. The average amount of HTTP traffic recorded per day is about 150 GB (about 340 mln individual HTTP requests), the usage data is in the order of 200 GB/day; the enriched HTTP traffic files amount to approximately 180 GB/day. The pre-processor joining the HTTP traffic and network usage files is implemented by the TILab software group in Trento using RabbitMQ (https://www.rabbitmq.com) for the parallel processing of chunks of input data and Redis (http://redis.io) for in-memory data caching of joined tuples to be added to the enriched HTTP traffic files in output.

Please note that, in line with similar Internet usage studies [1], personal identifiers were anonymized prior to the study, and only aggregated values are reported. Data are stored on in-house servers and password protected. Before publication, the work was checked by Telecom for compliance with Italian Law D.Lgs 196/2003 (which implements the EU Directive on Privacy and Electronic Communications of 2002), Telecom's own policies, and the NDA signed between Telecom and University of Trento.

4 Mobile Internet Traffic Analysis

We start our analysis of the use of REST APIs with a set of descriptive statistics about the available dataset as a whole. We recall that the data contain all HTTP requests recorded by the data collector over one full day of usage, including regular Web browsing activities. The analysis of the dataset provides an up-to-date picture of the Mobile Internet and informs the design of heuristics for the identification of those calls that instead involve APIs only (next section).

It is important to note that our analysis is based on HTTP traffic only and, for instance, does not take into account HTTPS traffic, streaming of audio/video media, or other protocols. As for the quality of the data analyzed, the data preprocessor's data joining logic has proven to have an approximate success rate of 90 % (due to diverse imprecisions in the input data); we could however not identify any systematic bias in the dataset due to failed joins.

4.1 HTTP Requests and Responses

Figure 2 summarizes the key characteristics of the dataset we leverage on in this paper. Figure 2(a) reports on the different *HTTP methods* (also called "verbs") used by the recorded HTTP requests, along with the respective count. We can see that the two most commonly used methods (`Get` and `Post`) dominate the traffic in today's Mobile Internet, followed by other methods such as `Connect`, `Head`, `Put`, `Options` and `Delete`. The less common methods `Propfind` and `Proppatch` are used by Web Distributed Authoring and Versioning (WebDAV), an extension of HTTP for web content authoring operations (see RFC 2518 [9]). `Source` (used by the Icecast multimedia streaming protocol), `Dvrget` and `Dvrpost` (used for multimedia/multipart content and streaming over HTTP), and `List` are other non-standard HTTP methods.

The identified usage of HTTP methods provide a first indication of the potential compliance of the RESTful APIs with the REST architectural style guidelines [6], which, as we have seen earlier, advocate the use not only of `Get` and `Post`, but also of `Put`, `Delete`, `Options`, `Head`, etc. for the implementation of what is called the "uniform interface" of REST APIs. Our dataset shows that by now these request methods are not only being used by some APIs, but have turned into state of the art.

In this respect, it is good to keep in mind that the mobile app market is largely characterized by applications that heavily leverage on Web APIs to provide their users with mobile access to large content repositories and highly scalable computing power, two resources that are typically limited on mobile devices. Since our dataset captures Mobile Internet usage, there may be a bias toward a more rich use of HTTP methods. On the other hand, it is important to note that the `Connect` methods are used to establish HTTPS connections, that is to switch from plain HTTP to its encrypted counterpart HTTPS. Once a communication switches from HTTP to HTTPS (e.g., when a user logs in to Facebook) we are no longer able to intercept tunneled HTTP requests and, hence, to follow the conversation. The estimation of the telco operator is that, of all the mobile internet traffic, around 25–30% corresponds to HTTPS traffic. We acknowledge the lack of such type of traffic as a limitation of our dataset.

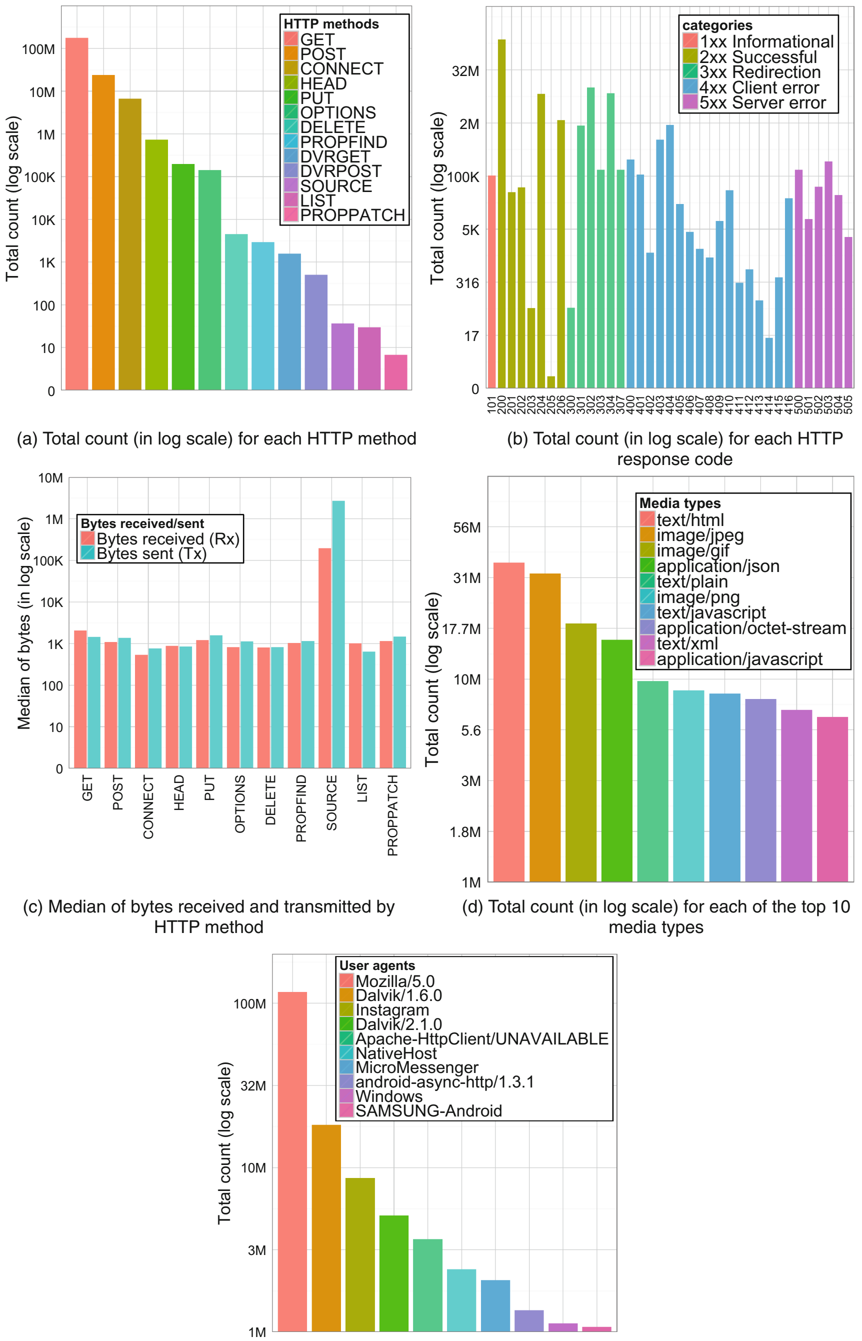
Figure 2(b) illustrates the counts of the *HTTP response codes* corresponding to the requests in Fig. 2(a). According to the figure, the recorded requests feature a rich and varied usage of HTTP response codes. Responses are dominated by successful and redirection operations (2xx and 3xx codes), and errors (4xx and 5xx) are mainly due to clients requesting resources not found on the server (404) or forbidden to the client (403). In 2005, Bhole and Popescu [24] did a similar analysis of HTTP response codes and identified only 5 different codes in their dataset, with status code 200 representing 88 % of the analyzed traffic – despite the HTTP protocol specification (version 1.1) dating back to 1999 [7]. In other words, after approximately one decade HTTP responses are characterized today by a much richer use of response codes and APIs that effectively work with the standard semantics of both request methods and response codes.

Figure 2(c) looks more detailedly into the different HTTP request methods and shows how much data is transmitted/received per method. Overall, the median of transmitted data is 1463 bytes, while the median of received data is 1643 bytes. The same numbers approximately hold for all methods, except for the `Source` method, which presents significantly higher values; we recall that the method is used by Icecast to stream multimedia content.

In 1995, Mah [13] showed that the median HTTP response length was about 2 KB. Pang et al. [20] registered a similar response length in 2005, and Maier et al. [15] approximately confirm analogous numbers in 2010. In the end of 2015, our dataset too confirms a similar median response length. This almost stable picture is somehow surprising, as over the last years we all have witnessed a Web that has grown more complex, in terms of both content and functionality. On the other hand, Mah also showed that in 1995 the median HTTP request length was about 240 bytes [13], while our dataset presents a median request length of about 1.5 KB. This change of the length of the requests must be explained by a different use of the Internet in upload between the two dates. In fact, from 1995 to today, the Web has evolved from Web 1.0 to Web 2.0, that is, from mono-directional content consumption to fully bidirectional content co-creation. The increase of request lengths provides evidences of this paradigm shift. A confirmation of this, however, would require an own, purposely designed study.

4.2 Media Type Usage

"Media types" are the generic Web synonym of "representations" in REST. Studying the media types returned by the HTTP requests allows one therefore to obtain a first indication of which representations state-of-the-art APIs use. Figure 2(d) shows the ten most used media types in our dataset. Keeping in mind that the dataset contains generic Web traffic (not only API traffic), it is of no surprise to find `text/html` on the first place, followed by `image/jpeg` and `image/gif`. More surprising is that the data format `application/json` is already on the forth position, while `text/xml` is only on the ninth position. As can further be seen in the figure, both `text/javascript`



(a) Total count (in log scale) for each HTTP method

(b) Total count (in log scale) for each HTTP response code

(c) Median of bytes received and transmitted by HTTP method

(d) Total count (in log scale) for each of the top 10 media types

(e) Total count (in log scale) for each of the top 10 user agents

Fig. 2. Descriptive statistics of the available dataset characterizing state-of-the-art Mobile Internet traffic as of October 2015.

and `application/javascript` refer to the same media type and, hence, the naming of the media types is not consistent throughout the different applications and/or APIs. In fact, it is good to note that the figure only shows the ten most used media type declarations; overall, the dataset contains 1134 different media type declarations.

The two media types that are of particular interest in this paper are of course JSON and XML, as these are meant for machine consumption and therefore refer to the invocation of an API or service. We exploit this property later on to identify calls to APIs among the huge amount of calls in the dataset. SOAP web services [21] too transfer XML-encoded data, yet the respective XML-encoded SOAP envelope is always associated with the media type `application/soap+xml` and transmitted via HTTP `Post` requests. It is therefore easy to distinguish calls to SOAP web services from potential calls to REST APIs.

4.3 User Agents

Finally, with Fig. 2(e) we would like to shed some light on the user agents used to issue the requests logged in our dataset. The figure again shows the ten most used user agents from a total of 57571 different user agent declarations. On the first position, we find Mozilla/5.0 with an extraordinary predominance. To understand this result, it is important to notice that the user agent string in the header of HTTP requests can be assigned arbitrarily by the user agents themselves. And this is what happens in practice, as nicely explained by Aaron Andersen in his blog http://webaim.org/blog/user-agent-string-history/: in order to prevent user agent sniffing and being discriminated, most modern Web browsers declare to be compatible with Mozilla/5.0 (even Internet Explorer, Edge, Safari and similar). More interesting to our own analysis are the user agents Dalvik (Android virtual machine), Android and Windows that testify the presence of mobile devices, while the user agents Instagram and MicroMessenger represent native mobile apps able to issue HTTP requests. Indeed, a closer inspection of our dataset revealed that 40.8 % of the traffic corresponds to native apps, while the rest 59.2 % is traffic generated from mobile, web browsers. As a follow up, future work we would like explore these two worlds with an own, dedicated study to understand whether and how they differ from each other.

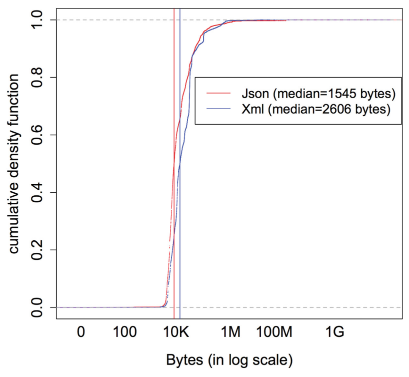
5 REST API Analysis

Given our dataset, which can be seen as a generic dump of HTTP requests that interleaves requests directed toward APIs for machine consumption with requests directed toward Web applications for human consumption, the first problem to solve is identifying which *requests* actually refer to the former. This is necessary to be able to effectively focus the analysis on APIs for software agents (from now on simply APIs) and not to be distracted by regular Web navigation activities. Given the limited amount of information available about the recorded HTTP requests, the problem is not trivial and requires the application of API-specific heuristics.

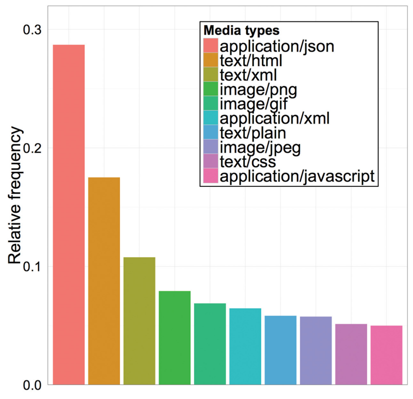
Recalling Fig. 2(d), we remember that among the top-10 media types used in our dataset we have JSON and XML, which are typical data formats for the exchange of data between software agents. It is thus reasonable to assume that requests returning any of these two media types are directed toward APIs. In order to identify such requests, we considered only those requests that contain the strings 'json' and 'xml' in their media type declaration. Examples of these include the common media types `application/xml` and `application/json`, but also less common media types such as `application/vnd.nokia.ent.events+json` and `application/vnd.wap.xhtml+xml`. The total number of such requests in our dataset is 18.2 million, 9.3 million for JSON and 8.9 million for XML.

In order to assure that these requests really return JSON and XML and to characterize the typical responses, we sampled all JSON and XML requests independently and representatively for the whole dataset using a 95 % confidence level and a confidence interval of 3. This corresponds to 1067 requests to the corresponding, presumed APIs randomly picked for both media types to obtain their payloads. Figure 3(a) shows the cumulative density function of the payload sizes. The medians are 1545 and 2606 bytes, respectively, for JSON and XML.

We also checked the formal validity of the payloads. Checks were performed using Python's internal libraries, which reported that 75 % and 76 % of the requests contained valid JSON and XML, respectively. The main reasons for invalid payloads were either empty payloads or, in the case of declared JSON payloads, the presence of JSONP callbacks (JSON wrapped in Javascript code) instead. As for the empty payloads, an inspection of the respective HTTP status



(a) Cumulative density function for the size in bytes of JSON and XML payloads.



(b) Relative frequency of number of domains using various media types. The total number of domains considered is 1000, which were obtained from our sampled data.

Fig. 3. Size in bytes for JSON and XML payloads, and media type distribution by host

codes reveals that most of them are explained by 4xx and 5xx error codes, that is, by resources that no longer exist or are not addressable on the server or because of session expiration. Overall, the counts of the status codes (in parenthesis) in the sample are: for JSON 1xx (0), 2xx (1204), 3xx (1), 4xx (243) and 5xx (53), and for XML 1xx (0), 2xx (1280), 3xx (0), 4xx (233) and 5xx (2).

The next step toward the identification of APIs would be deciding which concrete URLs serve as APIs *end/entry points* (e.g., api.server.org/universities), starting from where clients can start exploring the APIs. Doing so is however not feasible without inspecting each API individually. We thus limit our analysis in this section to individual HTTP requests, without trying to infer API endpoints.

Given an HTTP request, the options for end points may range from the plain host name (e.g., api.server.org) to the full URL at hand (e.g., api.server.org/universities/45/people/3). We discard this last option as too fine-grained, while, ideally, APIs should be accessible through a dedicated host name not used for other purposes. This would make the host name an identifier.

We tested this assumption: Using the same sample of 1067 requests as above, we identified the respective individual host names (incidentally precisely 1000) and went back to the full dataset recorded by the data collector to retrieve all media types that are accessible through these host names. If the host names were used only to provide API access, the media types would all be media types oriented toward software agents. In order to keep the computation manageable, we used a 15 min time slot of the full dataset collected during a high traffic hour. The slot contains a total of 3.2 million requests that, when joined with the 1000 different host names, corresponds to 3.2 billion comparisons. Figure 3(b) shows the relative frequency for the top-10 media types identified. The media type `aplication/json` has the highest frequency, followed by `text/html`, `text/xml` and others. The presence of `text/html`, `text/css` and `text/javascript` indicates that through the same host names also content oriented toward human agents (Web sites) is delivered, not only content oriented toward software agents. Hence, we conclude that host names are not good API identifiers in general.

5.1 Compliance with Design Best Practices

Next, we specifically focus on the set of 18.2 million API requests identified previously and study how well the designers of the respective APIs followed the design principles and best practices introduced in Sect. 2. We define a set of heuristics based on the request metadata available in the dataset as well as on the payloads we obtained from our representative sample of API invocations, in order to derive empirical evidence of compliance (or not).

Since some of the best practices as well as the maturity levels discussed in the next section do not apply to individual HTTP requests (which would be too fine-grained), we group requests by host names. This means that rather than studying the compliance of APIs or HTTP requests we study that of API providers, in that we look at the full traffic toward the APIs accessible under one and a same host name. Differently from the data underlying Fig. 3(b), here we only focus on requests targeted toward JSON and XML resources and, hence, we are sure we study API-related traffic only.

The heuristics are summarized in Table 2, implemented in JavaScript for node.js and Python, and the respective code is available on https://github.com/mbaezpy/api-analysis. For instance, `rUnderscore` uses a regular expression to tell whether an invoked URL contains underscores; we recall that the guideline is instead to use hyphens. `rLowercase` checks whether URLs comply with the guideline to use only lowercase letter, while `rSlash` checks that URLs don't end with a final slash character. We refer the interested reader to the online resource for the concrete implementation of all heuristics.

Table 2. Description of heuristics to identify compliance with design best practices.

Heuristics	Description
`rUndescore`	Number of URLs avoiding the use of underscores in URLs
`rLowercase`	Number of URLs using lowercase in paths
`rSlash`	Number of URLs avoiding the trailing forward slash
`rVersionInPath`	Number of URLs avoiding version number in the path
`rVersionInQuery`	Number of URLs avoiding version number in the query params
`rApiInDomain`	Number of URLs with API as part of the subdomain
`rApiInPath`	Number of URLs with API as part of the path
`rCrudResource`	Number of URLs avoiding CRUD operations as resource name
`rHideExtension`	Number of URLs hiding the implementation technology
`rFormatExtension`	Number of URLs avoiding media type as resource extension
`rQueryExtension`	Number of URLs avoiding media type as query param
`rCrudInParam`	Number of URLs avoiding CRUD actions in query params
`rActionInQuery`	Number of URLs using action params (to tunnel operations)
`rIdInQuery`	Number of URLs avoiding resource IDs as part of the query
`rResNameApi`	Number of URLs avoiding use of API as resource name
`rMatchMedia`	Number of URLs not violating the use of content type
`rCacheQuery`	Number of URLs avoiding the use of CACHE in query params
`rHypermedia`	Number of URLs containing hypermedia links for control

Figure 4(a) illustrates the mean, median and standard deviation of the compliance of the identified host names with each of the heuristics. For example, if we take heuristic `rUnderscore`, we can see that, on average, 75 % of the resources accessible through a host comply with this heuristic, with a median of 100 % and a standard deviation of 41 %, approximately. The figure shows that all except one heuristic (`rApiInDomain`) have a median of 100 %, and that they reached means higher than 95 %, the exceptions being `rUnderscore`, `rLowercase`, `rSlash` and `rApiInDomain` as well as `rHideExtension` and `rFormatExtension`.

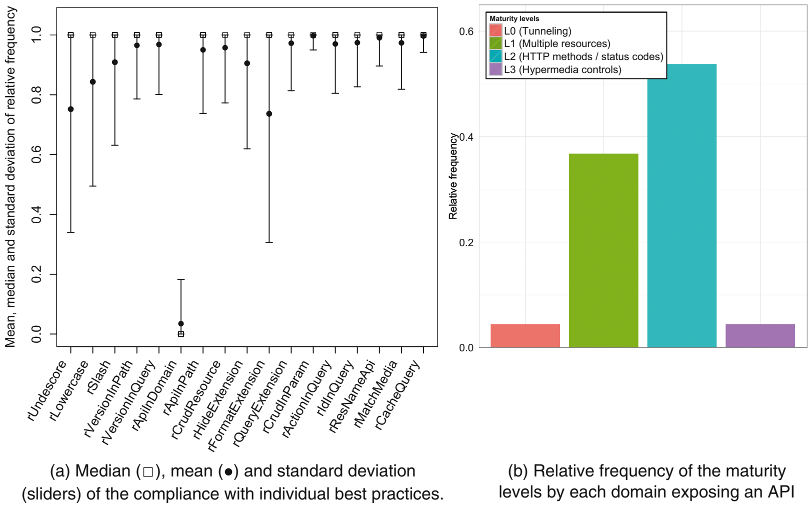


Fig. 4. Compliance of APIs with best practices and maturity levels of API providers.

Overall, these results are better than we expected. The lower compliance with the former four heuristics is not major issue that affects the quality of the actual service provided through an API; they refer to naming conventions, which may or may not be shared by all developers. However, the still rather high compliance with the first three heuristics tells that most of the developers actually do follow the best practice, while they don't seem to like the use of "api" in the URL (consistently with the finding above that host names typically intermix content for human and software agents). The low compliance with the heuristics `rHideExtension` and `rFormatExtension`, instead, may have a negative effect on the maintainability and future evolvability of APIs. In fact, making implementation technologies explicit in the URL (e.g., the file suffix `.php`) hinders the switch from one server-side implementation technology to another (e.g., node.js). By the same token, showing resource extensions (e.g., `.json`) prevents content negotiation between client and server to agree on which representation format to exchange (e.g., XML instead of JSON). Of course, both cases can still be implemented (e.g., by using javascript inside an endpoint with suffix `.php` and by delivering XML through a resource with extension `.json`), but conventions have their meaning, and developers would be confused and software agents (e.g., Web servers) may not properly handle these mismatches.

5.2 API Maturity Levels

In order to estimate the compliance of the identified APIs with the maturity levels by Richardson, we leverage on some of the above heuristics to implement composite logics representing each of the four levels of maturity. Again, we study the dataset of 18.2 million API requests and group requests by host name to study API providers rather than individual requests or APIs. Starting from the heuristics introduced earlier, we assign maturity levels to hosts as follows:

- *Level 0, Tunneling.* As from a given URL is not possible to derive in practice whether the respective API consists of one endpoint only, we check whether the requests declare actions as query parameters (`rActionInQuery`), whether they pass resource identifiers as a parameters (`rIdInQuery`), or whether they have a resource name that suggests tunneling (`rResNameApi`).
- *Level 1, Resources.* Here we look for APIs that use multiple endpoints that however do not yet make proper use of the semantics of HTTP. The heuristics we use here are CRUD names as resources (`rCrudResource`), problems in content negotiation (`rFormatExtension`, `rQueryExtension`), self-descriptiveness (`rMatchMedia`), and use of headers (`rCacheQuery`).
- *Level 2, HTTP methods.* API providers that make use of resources and proper use of HTTP qualify for this level. However, at this level APIs don't make yet use of hypermedia links. The heuristics used for this level include the avoidance of CRUD operations in the query params (`rCrudResource`), media types as resource extension (`rFormatExtension`), media types as query parameters (`rQueryExtension`), "cache" in query params (`rCacheQuery`), as well as the matching of media types with actual content (`rMatchMedia`).
- *Level 3, Hypermedia.* Hypermedia means links inside resource representations to enable the client to navigate among resources. The `rHypermedia` heuristic helps us identify resources in this level by looking for hypermedia links inside the payload of HTTP responses.

The following pseudocode implements the logic for the identification of levels (`dNumResources` is the number of individual URLs accessed through a given host, `dNumMethods` is the number of different HTTP methods used by the requests):

```
function calculateLevel012(){
 if (dNumResources == 1 && dNumMethods == 1 && (rActionInQuery < dNumResources ||
    rIdInQuery < dNumResources || rResNameApi < dNumResources)) {
       level ="L0";
 } else if (rActionInQuery < dNumResources || rIdInQuery < dNumResources ||
    rCrudResource < dNumResources || rFormatExtension < dNumResources ||
    rQueryExtension < dNumResources || rMatchMedia < dNumResources ||
    rCacheQuery < dNumResources)) {
       level = "L1";
 } else {
       level = "L2";
 }
 return level;
}

function calculateLevel3(){
  resources = resourcesInL2.sample().getPayloads();
  numResources = count(resources);
  rHypermedia = 0;
  for resource in resources{
    if (resource contains hypermedia links){
      rHypermedia = rHypermedia + 1;
    }
  }
  estimatedLevel = rHypermedia / numResources;
  return estimatedLevel;
}
```

Compliance with Levels 0–2 is computed on the full dataset containing the 18.2 million requests, including both XML and JSON. Since the computation of Level 3 needs access to the actual payload of the requests, Level 3 is computed over a representative sample of the hosts complying with Level 2 (which is a prerequisite for Level 3) for which we were able to access the respective payloads. The sample consists of 1048 different requests with a confidence level of 95 % and a confidence interval of 3, along with the corresponding payloads.

The result of this analysis is illustrated in Fig. 4(b), which reports the fractions of the studied dataset that comply with the four maturity levels. Few hosts reach Level 0; note that we explicitly focus on requests toward REST APIs and therefore excluded invocations of SOAP or XML-RPC calls by discriminating the respective media types. A significant part of the dataset complies with Level 1, yet the respective APIs do not make proper use of HTTP. The biggest part of the dataset, however, does make good use of HTTP and complies with Level 2, while only few hosts qualify for Level 3. These data indicate that the current use of REST APIs is mostly targeted at providing CRUD access to individual resources (Level 1 and 2), while full-fledged APIs that properly interlink resources and use hypermedia as the engine of state are still rare (Level 3).

Despite big steps towards resource-oriented services, there is still a large percentage of services not taking full advantage of the HTTP protocol to provide true standard interfaces. Developers should be more aware of the benefits of standard interfaces, e.g., to be compliant with the increasing number of libraries and frameworks (e.g., backbone.js, ember.js) based on RESTful principles. The limited support of hypermedia, comes as no surprise as there is no agreement on (*de facto*) standards or formats, at least not in JSON, to make the required investment by both service providers and clients worthwhile.

6 Related Work

Large scale analyses of HTTP requests have been presented in several works, but focusing mainly on quality of service [11], user profiling [14] or the general understanding of Internet traffic [15]. Analyses of RESTful design patters and anti-patterns have been the subject of recent studies [18,19]. Palma et al. [18] presented a heuristic-based approach for automatically detecting anti-patterns in REST APIs, namely SODA-R, that relies on service interface definitions and service invocation. The authors analyzed 12 popular REST APIs, finding anti-patterns in all of them, with more anti-patterns than patterns in services like Dropbox and Twitter. As an extension, the same authors [19] also looked at linguistic properties in 15 widely-used APIs with similar results. These studies provide insight into design patterns and tell us that even popular REST APIs have their issues. However, these works focus more on the validation of the proposed frameworks rather than on a large scale analysis of API design practices.

In contrast, in this paper we perform a large-scale analysis of REST API design best practices and of the underlying principles by studying up-to-date Mobile Internet traffic traces. Although limited by the metadata available, the large scale of the analysis presented in this paper gives us insights into the current practice that was not present in the aforementioned studies.

7 Conclusion

The work described in this paper advances the state of the art in Web engineering with three core contributions: First, to the best of our knowledge this is the first work that empirically studies how well the developers of REST APIs follow the theoretical principles and guidelines that characterize the REST architectural style. Second, the work defines a set of heuristics and metrics that allow one to measure implementation anti-patterns and API maturity levels. Third, the respective findings clearly show that, while REST APIs have irreversibly percolated into modern Web engineering practice, the gap between theory and practice is still surprisingly wide, and only very few of the analyzed APIs reach the highest level of maturity.

These findings all point into one direction: The implementation and usage of REST APIs – as well as that of Web services more in general – is still far from being a stable and consolidated discipline. On the one hand, this asks for better, principled Resource-Oriented and, in general, Service-Oriented Computing (SOC) methodologies, tools and skills [12]; pure technologies are mature enough. On the other hand, keeping in mind the ever growing strategic importance of APIs to business, this asks for better and more targeted service/API quality and usage monitoring instruments, such as proper KPIs for APIs [17].

Acknowledgement. This research has received funding from the Provincia Autonoma di Trento under the project e2Call (Enhanced Emergency Call), grant agreement number 82/13. The authors thank all partners within e2Call for their contribution.

References

1. An, X., Kunzmann, G.: Understanding mobile internet usage behavior. In: Networking Conference, IFIP 2014, pp. 1–9. IEEE (2014)
2. Berners-Lee, T., Fielding, R., Masinter, L.: Uniform Resource Identifier (URI): Generic syntax. Technical report (2004)
3. Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H.F., Thatte, S., Winer, D.: SOAP Version 1.2. W3c recommendation, W3C. http://www.w3.org/TR/soap/
4. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Services Description Language (WSDL) 1.1. W3c note, W3C, March 2001
5. Daniel, F., Matera, M.: Mashups: Concepts, Models and Architectures. Data-Centric Systems and Applications. Springer, Heidelberg (2014)
6. Fielding, R.: Architectural styles and the design of network-based software architectures. Ph.D. dissertation, University of California, Irvine (2007)
7. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol - HTTP/1.1. Technical Report RFC 2616, The Internet Society, (1999). http://www.ietf.org/rfc/rfc2616.txt

8. Fowler, M.: Richardson maturity model: steps toward the glory of rest (2010). http://martinfowler.com/articles/richardsonMaturityModel.html
9. Goland, Y., Whitehead, E., Faizi, A., Carter, S., Jensen, D.: HTTP Extensions for Distributed Authoring - WEBDAV. Rfc 2518, The Internet Society (1999). https://tools.ietf.org/html/rfc2518
10. Hadley, M.: Web Application Description Language. W3C member submission, Sun Microsystems (2009). http://www.w3.org/Submission/wadl/
11. Khirman, S., Henriksen, P.: Relationship between quality-of-service and quality-of-experience for public internet service. In: Proceedings of the 3rd Workshop on Passive and Active Measurement (2002)
12. Lagares Lemos, A., Daniel, F., Benatallah, B.: Web service composition: a survey of techniques and tools. ACM Comput. Surv. **48**(33), 1–41 (2015)
13. Mah, B., et al.: An empirical model of http network traffic. In: INFOCOM 1997, vol. 2, pp. 592–600 (1997)
14. Mai, T., Ajwani, D., Sala, A.: Profiling user activities with minimal traffic traces. In: Cimiano, P., Frasincar, F., Houben, G.-J., Schwabe, D. (eds.) ICWE 2015. LNCS, vol. 9114, pp. 116–133. Springer, Heidelberg (2015)
15. Maier, G., Schneider, F., Feldmann, A.: A first look at mobile hand-held device traffic. In: Krishnamurthy, A., Plattner, B. (eds.) PAM 2010. LNCS, vol. 6032, pp. 161–170. Springer, Heidelberg (2010)
16. Masse, M.: REST API design rulebook. O'Reilly Media Inc, Sebastopol (2011)
17. Musser, J.: KPIs for APIs. The Business of APIs Conference (2014). http://www.slideshare.net/jmusser/kpis-for-apis
18. Palma, F., Dubois, J., Moha, N., Guéhéneuc, Y.-G.: Detection of REST patterns and antipatterns: a heuristics-based approach. In: Franch, X., Ghose, A.K., Lewis, G.A., Bhiri, S. (eds.) ICSOC 2014. LNCS, vol. 8831, pp. 230–244. Springer, Heidelberg (2014)
19. Palma, F., Gonzalez-Huerta, J., Moha, N., Guéhéneuc, Y.-G., Tremblay, G.: Are RESTful APIs well-designed? detection of their linguistic (Anti)Patterns. In: Barros, A., et al. (eds.) ICSOC 2015. LNCS, vol. 9435, pp. 171–187. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48616-0_11
20. Pang, R., Allman, M., Bennett, M., Lee, J., Paxson, V., Tierney, B.: A first look at modern enterprise traffic. In: Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement, pp. 2–2. USENIX Association (2005)
21. Papazoglou, M.P.: Web Services - Principles and Technology. Prentice Hall, Upper Saddle River (2008)
22. Pautasso, C.: Some rest design patterns (and anti-patterns) (2009)
23. Pautasso, C.: Restful web services: principles, patterns, emerging technologies. In: Bouguettaya, A., Sheng, Q.Z., Daniel, F. (eds.) Web Services Foundations, pp. 31–51. Springer, New York (2014)
24. Bhole, Y., Popescu, A.: Measurement and analysis of HTTP traffic. J. Netw. Syst. Manage. **13**(4), 357–371 (2005)