Benchmarking Web API Quality

David Bermbach[1] and Erik Wittern[2(✉)]

[1] ISE Research Group, TU Berlin, Berlin, Germany
db@ise.tu-berlin.de
[2] IBM T.J. Watson Research Center, Yorktown Heights, NY, USA
witternj@us.ibm.com

Abstract. Web APIs are increasingly becoming an integral part of web or mobile applications. As a consequence, performance characteristics and availability of the APIs used directly impact the user experience of end users. Still, quality of web APIs is largely ignored and simply assumed to be sufficiently good and stable. Especially considering geo-mobility of today's client devices, this can lead to negative surprises at runtime.

In this work, we present an approach and toolkit for benchmarking the quality of web APIs considering geo-mobility of clients. Using our benchmarking tool, we then present the surprising results of a geo-distributed 3-month benchmark run for 15 web APIs and discuss how application developers can deal with volatile quality both from an architectural and engineering point of view.

1 Introduction

Nowadays, mobile and web applications regularly include third-party data or functionality through web APIs; often, the application's own back end systems are accessed in a comparable way. Building on technologies like AJAX, runtime environments like the Play Framework[1], and research results, e.g., from service-oriented computing, cloud computing, or mash-ups, this no longer poses a technological challenge. Therefore, we now see thousands of public APIs as well as applications using them [33].

In consequence, though, application developers now heavily rely on third-party entities beyond their control sphere for core functionality of their applications. This can have impacts on applications' user experience. For example, the latency of web API requests may impact application response times. Response times above 1 or 10 seconds have been shown to disrupt users' flow of thought or even cause loss of attention, respectively [25]. Or, a long-term experiment performed by Google showed that increasing response times artificially from 100ms to 400ms did measurably decrease the average amount of searches performed by users [10]. User experience and, hence, application reputation is thus directly affected by actions and non-actions of the API providers. As another example,

Author names are in alphabetical order as both authors have contributed equally.
[1] https://www.playframework.com/.

A. Bozzon et al. (Eds.): ICWE 2016, LNCS 9671, pp. 188–206, 2016.
DOI: 10.1007/978-3-319-38791-8_11

APIs may be discontinued or changed without notice, thus disabling applications. A recent analysis of a set of mobile applications finds that they silently fail and in cases even crash when confronted with mutated (e.g., adapted or faulty) web API responses [13]. Another aspect largely ignored yet is quality of web APIs: Due to the black-box nature of web API endpoints[2], applications are directly affected by volatile latencies, throughput limitations, and intermittent availability without having any influence or forewarning. Furthermore, quality of web APIs may vary depending on the geo-origin of requests.

In this work, we aim to shed some light on this issue and propose a number of strategies for dealing with poor quality. For this purpose, we present the following contributions:

1. A measurement approach and its prototypical proof-of-concept implementation for geo-distributed benchmarking of performance and availability of web APIs.
2. The results of a geo-distributed 3-month experiment with 15 web APIs.
3. A number of strategies on the implementation and architecture level for dealing with select observations from our experiments.

This paper is structured as follows: Initially, we give an overview of the request flow for web API calls and discuss how different qualities can be affected at various points in that request flow (Sect. 2). We also describe how we propose to measure select qualities. Next, we describe our experiment design (Sect. 3) and our observations (Sect. 4). Finally, we sketch-out how application developers can deal with lack of quality (Sect. 5), and discuss related work (Sect. 6) before coming to a conclusion (Sect. 7).

2 Quality of Web APIs

In this section, we give an overview of select qualities in web APIs and discuss how they can be measured. For this purpose, we start with a description of individual steps in performing web API requests (Sect. 2.1) and potential root causes of failures (Sect. 2.2). Afterwards, we characterize the qualities which we have studied for this paper (Sect. 2.3) and describe our measurement approach and the corresponding prototypical implementation of our toolkit (Sect. 2.4).

2.1 Interaction with Web APIs

Web APIs expose *data*, e.g., a user profile or an image file, and *functionalities*, e.g., a payment process or the management of a virtual machine through a resource abstraction. This abstraction enables users to manipulate these resources without requiring insight into the underlying implementation.

[2] We denote an endpoint to be the combination of a resource, identified by a URL, and an HTTP *method* as proposed in [30].

Developers can access Web APIs through the *Hypertext Transfer Protocol* (HTTP), which again uses the *Transmission Control Protocol* (TCP) for error-free, complete, and ordered data transmission on the transport layer, and the *Internet Protocol* (IP) at the network layer. Figure 1 illustrates the steps involved in a typical HTTP request[3].

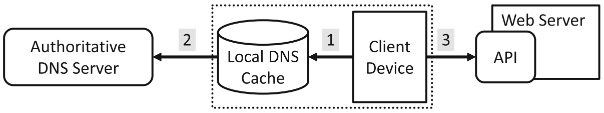

Fig. 1. Overview of the Steps Involved in Sending an HTTP Request

The resources exposed by an API are identified by *unified resource locators* (URLs), describing the scheme to be used for interaction, the server Internet address, and the specific resource identifier. The semantics of interactions with a resource depend upon the HTTP *method*, e.g., GET, POST, or DELETE. Before a client can send a request to the server that offers the web API, client and server need to establish a TCP connection. For this purpose, the client first sends a lookup request for the URL of the server to a *Domain Name Service* (DNS) server which returns the Internet protocol (IP) address and port number of the target host. If available, IP address and port may be returned from a local cache (step 1); otherwise, an external DNS authority is consulted (step 2). Afterwards, the client opens a socket connection to the server, i.e., it initiates TCP's three-way handshake, thus, establishing a TCP connection (step 3). Based on this connection, multiple HTTP requests with application data can be sent to the server.

If additional security is required, the client will typically use HTTPS which introduces the *Transport Level Security* (TLS) protocol[4] between HTTP and TCP/IP. TLS has two main phases: a negotiation phase and a bulk data transfer phase. In the negotiation phase, the server authenticates itself through his X.509 certificate. Afterwards, the client sends his list of supported cipher suites (a combination of symmetric encryption algorithm and a message authentication code (MAC)) to the server which then selects a cipher suite supported by both client and server and responds accordingly. Using asymmetric encryption (e.g., RSA) and key exchange protocols (e.g., DHE), client and server also agree on a symmetric key and other TLS sessions parameters.

After this TLS handshake has been completed, the server signals a change to the bulk data transfer phase. During that phase, each HTTP request is broken down into data packets which are – based on the agreed session parameters – encrypted and signed before transmission over the network. Cipher suite and protocol version determine whether encrypt-then-MAC or the reverse order is used.

[3] For simplicity's sake, we do not include possible complications like proxies, keep alive connections, caches, or gateways in this figure.

[4] TLS has largely replaced its predecessor SSL which is typically supported only for compatibility with old clients.

The recipient can then reassemble the original request and verify its integrity based on the received MAC.

2.2 Sources of Failures

Considering the typical HTTP request flow described in Sect. 2.1, a number of possible breakpoints emerge at which a request may fail. As we will see, while some of these are in control of a web API provider, others are not.

A **failed DNS lookup** is caused by attempting to look up a host for which no DNS entry exists or by a network partitioning which causes the lookup request to an authoritative DNS server to time out. The first error source is rather unlikely for web API requests with the correct URL, as it would imply the disappearance of the API's host altogether. Typically, a failed lookup results in a timeout error reported to the client. The second error source appears only in case that the network is not available and the DNS entry is not yet cached locally.

A **client connection error** appears if no TCP connection can be established between the client and the server hosting the web API. Reasons for this error are network partitioning or that the server is in a state where it cannot accept connections (for example, because it crashed).

In the case of HTTPS, a request can also fail if authentication of the server is not possible due to certificate issues or if there is no cipher suite supported by both client and server.

A **client error** appears if the request sent by the client cannot be processed by the server. One reason for client errors is that the requested resource cannot be found on the server. Furthermore, users may not be authorized to access the requested resource. The client may not have been aware of authentication mechanism like basic authentication or OAuth or may not own proper credentials. Furthermore, providers may deny authorization for specific clients if their usage of an API exceeds certain thresholds. A broad range of client errors are considered by HTTP and should result in the server sending 4xx status codes. While these errors are attested to the client, it is important to note that their appearance can be tightly related to actions of the web API provider. For example, many changes on the server, e.g., introducing authentication, removing or renaming resources, or changing request formatting, cause existing clients to malfunction, i.e., the client error is in fact caused by the web API provider.

A **server error** appears if the server fails to process an otherwise correct request. Reasons for server errors may include failed lookups for resources in databases or errors in the execution of functionalities. Server errors are, similar to client errors, considered by HTTP and should result in the server sending 5xx status codes.

2.3 Qualities

Systems have a number of properties. These can be functional, i.e., describe the abilities of said system, or non-functional, i.e., describe the quality of said system. Quality describes how "good" or "bad" a system fulfills its tasks along several dimensions[5] – the qualities.

There is a plethora of qualities that we can see in web APIs. Examples range from availability and performance, to security, reliability, scalability, cost, or correctness (of results). All these qualities are inherently connected through complex direct and indirect tradeoff relationships [3]. In this paper, we focus on two qualities: availability and performance.

Availability. Generally, availability describes the likelihood of a system – here, a web API – being able to respond to requests. Providing a concise definition of availability, though, is non-trivial: Does an API have to send correct responses or does it suffice if it is still able to tell about current problems? For this paper, we distinguish three different kinds of availability to consider these questions:

Pingability describes whether there is anything "alive" at the API provider's site. This may be a load balancer or even a fault endpoint. For a single machine deployment, pingability describes whether said machine is reachable at an operating system level. Pingability is fulfilled if, at the web API's URL, some entity responds to basic low level requests, e.g., ping requests (ICMP protocol).

Accessibility describes whether the resource represented by the web API is still accessible but not necessarily able to fulfill its task. For a single machine deployment, accessibility describes whether the web server component is reachable but does not require the hosted application logic to be accessible. A web API is accessible if it responds to HTTP requests using one of the predefined HTTP status codes.

Successability describes whether the web API is fully functional. For a single host deployment, it requires the application logic to be working[6]. Hence, we define successability to be fulfilled if a web API responds to requests using 2xx or 3xx status codes.

Performance. Performance has two dimensions: *latency* and *throughput*. Latency describes the amount of time between the start of a request at the client and the end of receiving a response, also at the client. Throughput, on the other hand, describes the number of requests a web API is handling at a given point in time. Typically, throughput measurements try to determine the maximum throughput, i.e., the maximum number of requests that a web API is able to handle without timeouts.

Usually, these two dimensions are interconnected: If the load increases towards maximum throughput, then latency will increase. If this is not the case, then the system behind the web API is typically referred to as elastically scalable [20,27].

[5] It depends on the respective quality what "good" or "bad" implies.

[6] Please, note, that successability does not say anything about correctness of results.

2.4 Implementation

We have built our measurement system around the quality definitions above: Pingability can be measured by sending a ping request to the respective endpoint; accessability and successability can be measured by sending HTTP or HTTPS requests. This also allows us to track request latencies. Originally, we also planned to measure throughput (and, thus, also scalability since we do not have any insight into the API provider's implementation) by sending large numbers of concurrent requests. For our experiments, we refrained from doing so since most provider's terms of use explicitly rule out any kind of usage which would resemble a DDoS attack. Therefore, we also recommend strongly that application developers who plan to roll out their application to large user groups should contact the respective API providers about throughput limits.

Our measurement toolkit is parameterized with a list of API endpoints. Based on this parameter list, it periodically sends ping, HTTP, and HTTPS requests. The toolkit then logs detailed results which are analyzed when the benchmark run has been completed. Our open source prototype for benchmarking web APIs is publicly available.[7]

3 Experiment Design

In this section, we will give an overview of our experiment setup. We will start with a description of chosen API endpoints (Sect. 3.1) before continuing with the parameters of our experiment setup (Sect. 3.2).

3.1 Analysed Web API Endpoints

Typically, web API endpoints require clients to authenticate to interact with the resources they expose. Authentication mechanisms include *basic authentication*, where a username and password have to be sent with every request, or the more frequently used *OAuth*. OAuth allows clients to access web APIs without having to expose user's credentials. OAuth, however, requires eventual human interaction to authorize client applications for requests and involves additional requests to establish authentication. While this may be an option for actual applications, it is prohibitive for benchmarking purposes. One way to circumvent the need for human interaction is to "automate" it through the use of people services [4], but latency and availability measurements would then become entirely meaningless. Another possibility is to make requests using API-specific *software development kits* (SDKs), which hide such complexities from the client. However, SDKs are not readily available for all APIs. Furthermore, correctly interpreting benchmark results obtained using SDKs would require us to fully understand their inner workings (e.g., through code-reviews) and eventually to align them for comparability. All in all, we, thus, decided to perform our experiments with unauthenticated API endpoints only. Focusing on these endpoints allows us to

[7] https://github.com/dbermbach/web-api-bench.

control the parameters influencing our measurements, e.g., as we avoid multiple roundtrips (possibly to third-party entities) for authentication.

We identified unauthenticated API endpoints from a variety of different providers with regards to company sizes, country of origin, local or global target users, public or private sector. We specifically included some of the most well-known providers like Google, Apple, Amazon, and Twitter. Table 1 gives an overview of the web API endpoints which we have selected for our experiments and the respectively supported protocols.

Table 1. Benchmarked API Endpoints and Supported Protocols.

API Name	ICMP	HTTP	HTTPS	Request Meaning
Amazon S3	-	X	X	Get file list for the 1000 genomes public data set
Apple iTunes	X	X	X	Get links to resources on artists
BBC	-	X	-	Get the playlist for BBC Radio 1
ConsumerFinance	X	X	X	Retrieve data on consumer complaints on financial products in the US
Flickr	X	X	X	Get list of recent photo and video uploads
Google Books	X	-	X	Get book metadata by ISBN
Google Maps	X	X	X	Query location information by address
MusicBrainz	X	X	X	Retrieve information about artists and their music
OpenWeatherMap	X	X	-	Get weather data by address
Postcodes.io	X	X	X	Get location information based on UK zip codes
Police.uk.co	-	X	X	Retrieve street level crime data from the UK
Spotify	X	X	X	Get information on a given artist
Twitter	X	X	-	Get the number of mentions for a given URL
Wikipedia	X	X	X	Retrieve a Wikipedia article
Yahoo	X	X	X	Get weather data by address

Please, note: In this work, we want to highlight unexpected behavior and unpredictability of web API quality but we do not aim to discredit individual API providers. For this reason, we decided to anonymize our results and will, for the remainder of this paper, only refer to these API endpoints as API-1 to API-15. There is no correlation between the identifiers and the order of API endpoints in Table 1. However, we will reveal this mapping information upon request if we are convinced that the information will not be used to discredit individual providers.

3.2 Experiment Setup

To analyze whether API quality varies depending on the geo-origin of the request, we deployed our toolkit in several locations. For this purpose, we used one Amazon EC2 instance in each of the following Amazon regions: US East (Virginia), US West (Oregon), EU (Ireland), Asia (Singapore), Asia (Sydney), Asia (Tokyo), and South America (Sao Paulo).

We configured our toolkit to send ping, HTTP, and HTTPS requests every 5 min but each starting at a different timestamp for the first request to avoid interference. Ping was configured to use 5 packets per request and we disabled local caching for HTTP and HTTPS requests.

Our test started on August 20th, 2015 (16:00h CEST) and was kept running for exactly three months; due to detailed logging and extensive prior testing, we can rule out crashes and other issues happening in our prototype.

4 Observations

Within this section, we present select results of our quality benchmarks. First, we summarize findings regarding availability and latency (Sect. 4.1). The partially poor quality revealed by these findings motivates our discussion of mitigation strategies in Sect. 5. Second, we present select cases of observations that reveal interesting behavior and correlations in qualities (Sect. 4.2).

We have uploaded our collected raw data on GitHub so that other researchers can use it as well.[8]

4.1 General Observations

Figure 2 summarizes our findings with regard to collected **availability** measures. For every API endpoint, all measurements from all regions were aggregated to derive the illustrated successability values. Results regarding ping requests allow statements regarding the pingability of the web APIs (cf. Sect. 2.3). For ping requests, the results presented in Fig. 2 reflect the mean values for all ping requests performed (we performed 5 attempts to ping a server per measurement). In cases where no data is presented, the server did not allow ping requests with the ICMP protocol. Results regarding HTTP and HTTPS requests allow statements regarding the accessibility and successability of requests (cf., Sect. 2.3). The results presented in Fig. 2 focus on successability, that is, the success rates relate to the number of requests that return an HTTP status code between 200 and 399. On the other hand, we do not differentiate here between requests failing due to failed DNS lookups, client connection errors, client errors, or server errors. Thus, Fig. 2 reflects the perspective of an application user for whom it is more important *if* a request fails rather than *why* it does. In cases where no HTTPS data is available, the endpoint did not support TLS.

[8] https://github.com/ErikWittern/web-api-benchmarking-data.

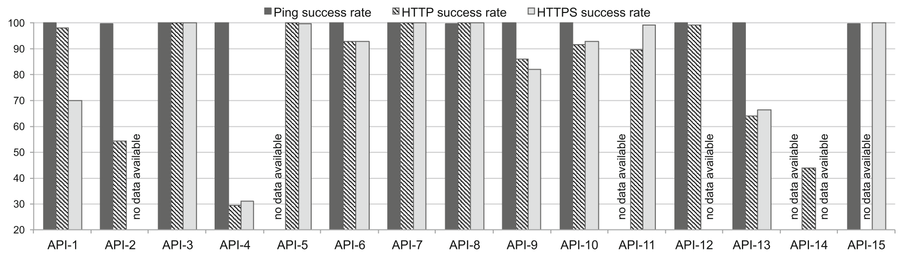

Fig. 2. Aggregated Availability Results.

Table 2. Availabilities of Benchmarked API Endpoints.

End-point	Successability [%]			Days with HTTP/HTTPS Successability <50 %							
	Ping	HTTP	HTTPS	Ireland	Oregon	Sao Paulo	Singa-pore	Sydney	Tokyo	US East	Sum
API-1	99.97	97.80	69.73	11/11	-/72	-/-	-/-	3/29	-/-	83/-	14/195
API-2	99.35	54.16	-	43/-	43/-	43/-	43/-	43/-	43/-	43/-	301/-
API-3	99.83	99.98	99.98	-/-	-/-	-/-	-/-	-/-	-/-	-/-	-/-
API-4	99.86	29.43	31.08	64/64	64/64	78/64	64/64	64/64	64/67	64/64	462/451
API-5	-	99.97	99.40	-/4	-/-	-/-	-/-	-/-	-/-	-/-	-/4
API-6	99.88	92.59	92.58	49/49	-/-	-/-	-/-	-/-	-/-	-/-	49/49
API-7	99.96	99.98	99.98	-/-	-/-	-/-	-/-	-/-	-/-	-/-	-/-
API-8	99.33	99.96	99.96	-/-	-/-	-/-	-/-	-/-	-/-	-/-	-/-
API-9	99.75	85.79	81.90	-/-	-/-	-/89	-/-	6/-	57/-	-/-	63/89
API-10	99.94	91.37	92.56	49/49	-/-	-/-	-/-	-/-	-/-	-/-	49/49
API-11	-	89.44	99.13	-/-	66/-	-/-	-/-	3/6	-/-	-/-	69/6
API-12	99.79	98.86	-	-/-	-/-	-/-	-/-	-/-	-/-	-/-	-/-
API-13	99.77	63.81	66.17	-/-	92/92	-/-	-/-	70/70	3/58	71/-	236/220
API-14	-	43.75	-	-/-	91/-	92/-	-/-	92/-	88/-	4/-	367/-
API-15	99.23	-	99.96	-/-	-/-	-/-	-/-	-/-	-/-	-/-	-/-
Sum				216/177	356/228	213/153	107/64	281/169	255/125	265/64	1610/1063

We furthermore depict corresponding figures in the columns "success rates" in Table 2. We find that availability rates, indicated by ping requests, are above 99 % for all APIs that support the ICMP protocol. With respect to successability, indicated by HTTP and HTTPS GET requests, a different picture emerges. Of the 14 endpoints tested with HTTP GET requests, 8 have a successability of 90 % or higher, 6 have a successability of 95 % or higher, and only 4 have a successability of 99 % or higher. On the bottom, 4 endpoints even show a successability below 65 %. On average, the 12 endpoints tested with HTTPS GET requests performed slightly better than their HTTP counterparts. 6 out of 12 endpoints have a successability of 99 %, and 8 have a successability of 90 % or higher.

To obtain a more precise picture about the distribution of the availability of API endpoints across regions, we determined for every API in every region the number of days in which successability is below 50 %. Note that overall, our experiments ran for 92 days. The results of this analysis are presented in the columns "Days with HTTP/HTTPS Successability <50 %" in Table 2.

The results show that even if overall successability is above 90 %, there can be considerable outages across a day in a subset of regions. For example, tested endpoints of API-6 and API-10 are not available for 49 days via HTTP from region Ireland. Furthermore, differences in the overall successability between different regions become visible. Oregon has overall the most and Singapore the least days in which web API calls fail for over 50 % of all requests. Interestingly, considering the successability observations of the 11 API endpoints that accept requests both via HTTP and HTTPS, we find that on average in 73.53 % of all situations where either HTTP or HTTPS requests fail, the other request does succeed during the same time frame.

Next, we also assess **latency** figures for the tested API endpoints. In Fig. 3, we present box plots of the measured latencies for HTTPS requests across regions. We decided to present HTTPS requests only since HTTPS seems to be becoming the default in the web.

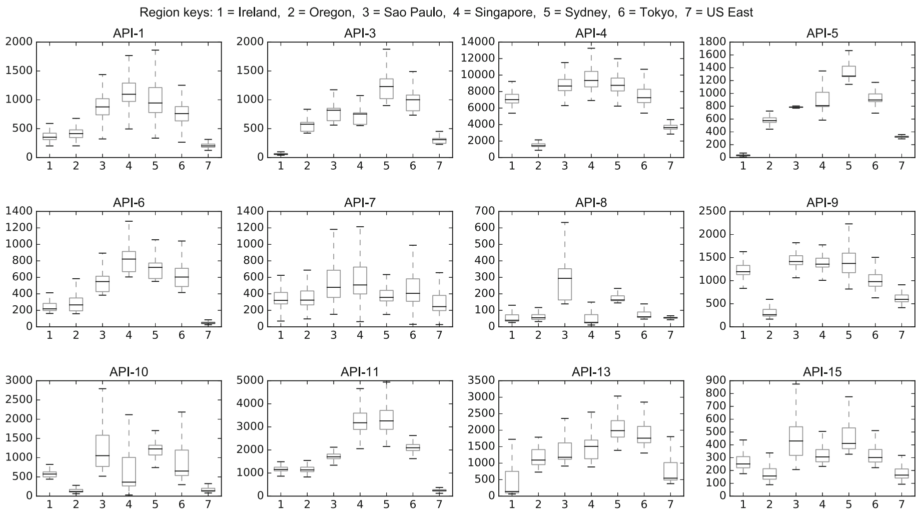

Fig. 3. Summary of HTTPS Request Latencies Across Regions in Milliseconds.

We consider two observations especially significant from the presented figures. First, for a single web API endpoint, latencies can vary tremendously depending on the geographic region from which requests originate. On average per API, the highest mean latency in a region is approximately 9 times higher than the lowest mean latency in another region. Even for the endpoint of API-7, whose performance is the most consistent one, the average latency in Singapore is 1.64 times higher than in Ireland. For 10 out of the 12 presented web API endpoints, the difference between the lowest and highest average latency is above 300 %, for 7 it is above 500 %, and for 4 endpoints, it is even above 1000 %. At the top end, for API-5, the highest average latency is over 27 times higher than the lowest average latency in another region. Second, even within individual regions, latencies can vary tremendously. For example, calls to the endpoint of API-1 from region Sydney have a standard deviation of 6943.63 around a mean value of 1411.61ms. Or, also from region Sydney, calls to the endpoint of API-7 have a standard deviation of 2410.95 around a mean value of 487.39ms. Both cases feature significant outliers (up to 498, 246ms respectively 120, 035ms). So while these endpoints may technically be available, their response times render them unusable in some cases from a practical point of view.

4.2 Select Examples

The first example of interesting behavior is that two API endpoints, the ones of API-2 and API-4, became unavailable in all regions during our experiments and remained that way until we concluded our tests. Figure 4 shows observed status codes of HTTP requests to the endpoint of API-2 over time. For the chart, we assigned a status code of 600 to requests for which the server never returned a response. Possible error sources are failures in the DNS lookup, client connection errors, or network partitionings (cf. Sect. 2.2), which typically result in a timeout of the request. The figure shows that requests predominantly succeed in the first half of our experiments, except for eventual server errors or requests for which we received no answer. At a certain point, however, requests consistently start to return a status code of 401, indicating that the client failed to authenticate, and some infrequent lost requests with status code 600. In this case, it seemed the provider turned the originally open endpoint into an authenticated one, requiring clients to adapt correspondingly to continue working.

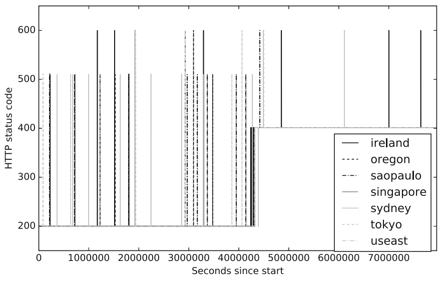

Fig. 4. Availability of API-2 over Time.

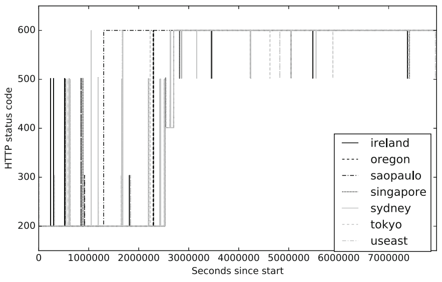

Fig. 5. Availability of API-4 over Time.

Figure 5 shows the status codes for HTTP requests to the endpoint of API-4 over time. Here, requests start to fail from one region, Oregon, at first. Then, as in the case of the endpoint of API-2, all regions receive status codes of 401 indicating unauthorized calls, however, shortly after turning into status codes of 600 throughout. This behavior is at odds with the fact that pingability of API-2 succeeded throughout the experiment.

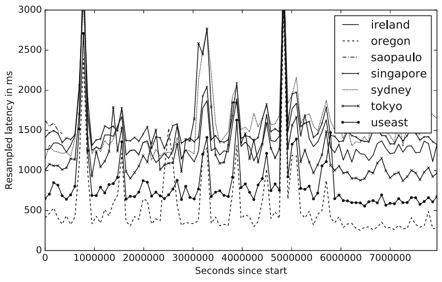

Fig. 6. Daily Latencies by Region (API-9).

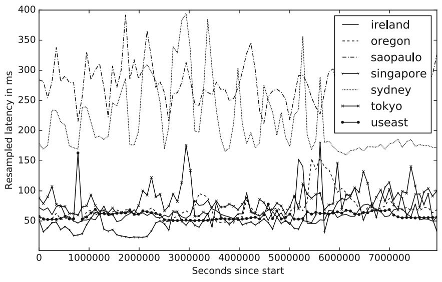

Fig. 7. Daily Latencies by Region (API-8).

Another interesting observation from our experiments is the possibility to infer data center locations. Figures 6 and 7 plot the average daily latency for the endpoints of API-9 and API-8 over the period of our experiment. Figure 6 is an example for a case where latency values across regions follow one pattern. The region from which requests feature the consistently lowest latencies is Oregon, indicating that API-9's endpoints are served from close to that region. There are still differences in latency due to the distance from that region, but the uniformity in the behavior of latencies indicates that all global requests are served from a single geographical location. On the other hand, in Fig. 7, the latencies from the different regions are much less clearly related. Notably, latencies for the regions Sao Paulo and Sydney are significantly higher than in the other five regions. Apart from that, the API seems to serve requests in different regions from different data centers, as there are little notable similarities between latencies.

4.3 Discussion and Threats to Validity

We rule out network partitionings that would disconnect a region completely for a longer period of time. As we see from the results of very well performing APIs (e.g.,. API-8, API-3), successful requests were performed from all regions throughout the duration of our experiments.

Our experiments target select endpoints. The findings from our experiments can, thus, not necessarily be generalized to the overall API from which the endpoint stems. For example, providers may choose to remove an endpoint we benchmarked, while the majority of the API is still available. Another aspect is that we only used GET requests – due to caching, actual availabilities may be worse. Nonetheless, we deem our results valid examples of how web API qualities can impact applications, which ultimately rely on specific endpoints.

We, furthermore, limited our experiments to endpoints that do not require authentication. One might argue that these endpoints may be of less importance to their providers and may, thus, undergo less scrutiny than other endpoints. Nonetheless, these endpoints may well be used by applications and the here presented findings can, hence, be considered relevant to application developers.

5 Implications for Application Engineering

Based on the surprising measurement results we reported in Sect. 4, we now describe implications for application engineering. First, we outline how concrete observed behaviors impact applications and present possible measures to directly address them (Sect. 5.1). Second, we discuss how different architecture options are able to deal with fluctuations in web API quality (Sect. 5.2).

5.1 Observed Behaviors and Direct Measures

Highly volatile latencies and temporary unavailabilites: All of our benchmarked web APIs show highly volatile latencies and temporary unavailabilities, even for requests originating from the same region. These behaviors lead to equally volatile application non-functionalities, whose impact depends upon the role the web API requests play in the application. If web API requests are a central part of an application's functionality, e.g., showing the user's location based on a maps API, the whole user experience will suffer. On the other hand, some web API calls perform mere supportive functionalities, e.g., advertising APIs, in which cases the perceived experience does not suffer.

Without detailed monitoring, an application provider may not even become aware of these quality problems and will only notice decreasing usage numbers (cf. [10]). While some web APIs expose monitored past and current availability and response times[9], the usefulness of this information is limited because it only focuses on the availability from a API provider perspective. Similarly, web API monitoring services like Runscope[10] only measure the availability of the server. However, unavailability may also be an effect of network partitioning between client and server, which is especially relevant for availability across geographic regions and/or from mobile clients.

In cases where a small degree of data staleness is acceptable, client-side caching can be used to address volatile latencies or unavailabilities, e.g., standard HTTP client-side caching or HTML 5's offline web storage. Recently, Google proposed a service to queue web API requests in case of temporary unavailability of mobile devices [1]. While the device is unavailable, the service queues web API requests and executes them once availability is back. This pattern could be adopted to compensate for temporary unavailability of web APIs in cases where the user does not need the results of the API call, e.g., status posts on social networks will work but requests for coordinates based on a given address will not.

Differences in latency and availability based on geo-origin of requests: Another set of observed behaviors is that some APIs denote stark differences in latency and availability across regions. Thus, developers rolling out globally accessible applications should not expect that web API qualities observed locally are true for every user. One approach to address this issue is, as we do in this paper,

[9] For example, http://status.ideal-postcodes.co.uk/.
[10] https://www.runscope.com/.

to perform geographically distributed benchmarking of qualities as part of the API selection process. Existing performance monitoring tools targeting website or web API providers could be used for this purpose[11]. However, the resulting efforts and costs might still render this solution inadequate, especially, since observed behavior may change at any time and without warning. Interestingly, we find that in many cases HTTP endpoints were accessible while HTTPS ones were not and vice versa (cf. Sect. 4.1). If the nature of the resource to interact with permits it, one approach to increase availability is, thus, to simply switch protocol if an API becomes unavailable. An alternative is using another API with comparable functionality as backup, i.e., choosing a strategy comparable to horizontal SaaS federation as proposed in [22].

Long-Lasting Unavailabilities and Disappearance of Endpoints: Finally, we have observed long-lasting unavailabilities or even the disappearance of endpoints based on discontinued or changed APIs in our experiments (cf. Sect. 4.2). This behavior causes a serious risk for application developers as functionalities their application rely on might disappear entirely, potentially even only in regions that developers do not have direct access to. Again, developers can rely on continuous, distributed monitoring of APIs to detect such cases, if they can justify the efforts and cost. A recent service[12] addresses the issue of (parts of) web APIs disappearing by allowing developers to register for notifications in case of API changes.

All in all, developers should not assume that web APIs are a given in their (then) current form or that their performance remains anything close to stable. Sending API requests asynchronously is, from a user experience perspective, probably a good idea for most scenarios.

5.2 Considering Web API Quality at Architecture Level

Since quality of web APIs is highly volatile, it needs to be considered in engineering of web or mobile applications. We have already discussed some direct measures like client-side caching or geo-distributed benchmarking during the API selection process. However, web API quality can also be considered on an architectural level. Figure 8 gives an overview of three different options.

The first option, on the left, is probably the state of the art for most mobile applications but also for many web applications. Whenever the API is needed, the application directly invokes the API and is, thus, highly dependent on it and mirrors experienced quality to the end user. This architectural style, hence, does not account for variations in quality but is the easiest to implement.

The second option, in the middle, uses a backup API in case of problems (unavailability or high latency) with the original one. To our knowledge, this is not yet done in practice (at least not on a large scale) but may become more and more feasible as dynamic service substitution techniques which have been well

[11] For example, https://www.pingdom.com.
[12] https://www.apichangelog.com/.

researched in service-oriented computing (e.g., [14]) can leverage the increased use of machine-readable API descriptions (e.g., Swagger) and corresponding research efforts [30]. This architectural style offers some degree of resilience to API-based problems but obviously introduces additional complexity (and, thus, development cost). For mobile applications, it may also be difficult to account for permanent API changes as new versions will never reach all users and will, in any case, take some time.

When using this architectural style, we propose to add a monitoring component to the app which tracks latency and availability of the API calls based on the current location of the client and periodically forwards aggregations of this information to a back end component of the app developer. Otherwise, it may take really long for the developer to become aware of permanent API changes or long-lasting unavailabilities in some geographical regions.

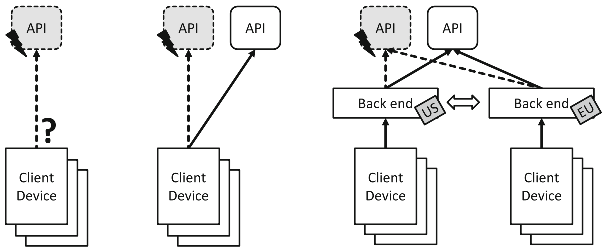

Fig. 8. High-Level Architecture Options for Web and Mobile Applications

The third option, on the right, uses a (geo-distributed) back end controlled by the application provider. Client devices do not invoke APIs directly. Instead, they direct their requests to the back end system which acts as a proxy, making actual API requests on behalf of the application. While this option certainly introduces an additional layer of complexity and cost (operating an additional back end), it also offers a set of unique benefits with regards to dealing with web API quality and resulting user experience:

1. Changing the API which is used becomes rather trivial and can be rolled out quickly on the back end. This is especially helpful as the back end will also be the entity to primarily detect API quality problems and is, through communication with other back end components, also able to interpret these problems (i.e., whether a problem is a temporary or regional issue).
2. Unavailabilities in some regions can be accounted for by tunneling API requests through another region. For instance, in case of problems with API availability for EU clients, the EU back end could send all its API requests through the US back end – at least if higher latencies are preferable over unavailability.
3. This architecture will typically improve user-perceived performance: Client-side caching can be used on both the client device and the back end, thus,

resulting in an additional speed-up and the back end may prefetch data by anticipating future API requests from the client device.
4. The API may not be a perfect fit for the requirements of the application, e.g., by returning excess amounts of data. At least for mobile applications data traffic is often expensive and, depending on the current location, slow. An additional back end can preprocess API call results (e.g., scale down images based on the needs of the client device) and can use custom, highly optimized protocols and data formats for communication with the client device.

All in all, the positive advantages outlined above should always be compared against the required efforts and cost both for building and for operating the back end. Still, the back end-based architecture will be a good solution for many application scenarios.

6 Related Work

To our knowledge, this is the first paper to address quality of web APIs through long-term benchmarking experiments, especially considering geo-distribution. There is, however, a lot of work quantifying quality in other application domains:

Beyond the classical TPC[13] benchmarks, a new set of benchmarks has recently been developed for various kinds of database and storage systems, e.g., [2,5–8,12,17,19,26,27,31,34]. There is also a number of dedicated cloud benchmarks that treat various (cloud) services as black boxes as we have done in our experiments, e.g., [9,15,18,21,23,35], or many SPEC[14] benchmarks, e.g., [28]. There are also a number of approaches trying to measure security or security overheads, e.g., TLSBench [24] for NoSQL databases, [16] for web services, and [11] for web servers.

Some works have studied how web APIs evolve [29] and how this evolution impacts client applications [32]. Others assess how well clients, e.g., mobile applications, are capable of dealing with web API changes [13]. We present specific cases in which web API endpoints changed to require authentication or eventually disappeared in Sect. 4.2. Our work, hence, provides an empirical motivation for these related works and more generally motivates a discussion about how clients can deal with web API imperfections as presented in Sect. 5 – no matter if these imperfections are caused by API evolution or other effects.

7 Conclusion

As the number of web APIs and their usage grows, their quality increasingly impacts application behavior and user experience. In this work, we presented the means to benchmark select qualities of web APIs in a geo-distributed way. Our 3-month study of 15 API endpoints reveals serious quality issues: Availability varies considerably between APIs, ranging from temporary outages to

[13] http://www.tpc.org.
[14] https://www.spec.org/.

the complete disappearance of tested endpoints. Furthermore, average latencies vary across regions by a factor of 9. In some cases, the observed latency of requests was so high that it virtually resembles unavailability. These findings show that application developers need to be aware of these issues and need to mitigate them if possible. For that reason, we presented ways for application developers to detect and handle web API unavailabilities and deal with volatile performance.

References

1. Archibald, J.: Introducing Background Sync. https://developers.google.com/web/updates/2015/12/background-sync?hl=en. Accessed: 17 Dec 2015
2. Bermbach, D., Tai, S.: Benchmarking eventual consistency: lessons learned from long-term experimental studies. In: Proceeding of IC2E, pp. 47–56. IEEE (2014)
3. Bermbach, D.: Benchmarking Eventually Consistent Distributed Storage Systems. Ph.D. thesis, Karlsruhe Institute of Technology (2014)
4. Bermbach, D., Kern, R., Wichmann, P., Rath, S., Zirpins, C.: An extendable toolkit for managing quality of human-based electronic services. In: Proceedings of the 3rd Human Computation Workshop HCOMP (2011)
5. Bermbach, D., Kuhlenkamp, J.: Consistency in distributed storage systems. In: Gramoli, V., Guerraoui, R. (eds.) NETYS 2013. LNCS, vol. 7853, pp. 175–189. Springer, Heidelberg (2013)
6. Bermbach, D., Tai, S.: Eventual consistency: how soon is eventual? an evaluation of amazon s3's consistency behavior. In: Proceedings of MW4SOC, pp. 1–6. ACM (2011)
7. Bermbach, D., Zhao, L., Sakr, S.: Towards comprehensive measurement of consistency guarantees for cloud-hosted data storage services. In: Nambiar, R., Poess, M. (eds.) TPCTC 2013. LNCS, vol. 8391, pp. 32–47. Springer, Heidelberg (2014)
8. Binnig, C., Kossmann, D., Kraska, T., Loesing, S.: How is the weather tomorrow?: towards a benchmark for the cloud. In: Proceedings of DBTEST, pp. 1–6. ACM (2009)
9. Borhani, A.H., Leitner, P., Lee, B.S., Li, X., Hung, T.: WPress: an application-driven performance benchmark for cloud-based virtual machines. In: Proceedings of EDOC, pp. 101–109. IEEE (2014)
10. Brutlag, J.: Speed Matters for Google Web Search. Google, Inc, Technical report (2009)
11. Coarfa, C., Druschel, P., Wallach, D.S.: Performance analysis of TLS web servers. ACM Trans. Comput. Syst. (TOCS) **24**(1), 39–69 (2006)
12. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of SOCC, pp. 143–154. ACM (2010)
13. Espinha, T., Zaidman, A., Gross, H.G.: Web API Fragility: How robust is your mobile application? In: Proceedings of MOBILESoft, pp. 12–21. IEEE (2015)
14. Fredj, M., Georgantas, N., Issarny, V., Zarras, A.: Dynamic service substitution in service-oriented architectures. In: IEEE Congress on Services - Part I, pp. 101–104. IEEE, July 2008
15. Garfinkel, S.L.: An Evaluation of Amazon's Grid Computing Services: EC2, S3, and SQS. Harvard University, Technical report (2007)

16. Juric, M.B., Rozman, I., Brumen, B., Colnaric, M., Hericko, M.: Comparison of performance of web services, WS-security, RMI, and RMI-SSL. J. Syst. Softw. **79**(5), 689–700 (2006)
17. Klems, M., Bermbach, D., Weinert, R.: A runtime quality measurement framework for cloud database service systems. In: Proceedings of QUATIC. pp. 38–46 (2012)
18. Klems, M., Menzel, M., Fischer, R.: Consistency benchmarking: evaluating the consistency behavior of middleware services in the cloud. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 627–634. Springer, Heidelberg (2010)
19. Kossmann, D., Kraska, T., Loesing, S.: An evaluation of alternative architectures for transaction processing in the cloud. In: Proceedings of SIGMOD. pp. 579–590. ACM (2010)
20. Kuhlenkamp, J., Klems, M., Röss, O.: Benchmarking Scalability and Elasticity of Distributed Database Systems. pp. 1219–1230 (2014)
21. Kuhlenkamp, J., Rudolph, K., Bermbach, D.: AISLE: assessment of provisioned service levels in public IaaS-based database systems. In: Barros, A., et al. (eds.) ICSOC 2015. LNCS, vol. 9435, pp. 154–168. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48616-0_10
22. Kurze, T., Klems, M., Bermbach, D., Lenk, A., Tai, S., Kunze, M.: Cloud federation. Cloud Comput. **2011**, 32–38 (2011)
23. Lenk, A., Menzel, M., Lipsky, J., Tai, S., Offermann, P.: What are you paying for? performance benchmarking for infrastructure-as-a-service offerings. In: Proceedings of CLOUD, pp. 484–491. IEEE (2011)
24. Müller, S., Bermbach, D., Tai, S., Pallas, F.: Benchmarking the performance impact of transport layer security in cloud database systems. In: Proceedings of IC2E, pp. 27–36. IEEE (2014)
25. Nielsen, J.: Usability Engineering. Elsevier, 1st edn. (1994)
26. Patil, S., Polte, M., Ren, K., Tantisiriroj, W., Xiao, L.,López, J., Gibson, G., Fuchs, A., Rinaldi, B.: YCSB++: benchmarking and performance debugging advanced-features in scalable table stores. In: Proceedings of SOCC, pp. 1–14. ACM (2011)
27. Rabl, T., Gómez-Villamor, S., Sadoghi, M., Muntés-Mulero, V., Jacobsen, H.A., Mankovskii, S.: Solving Big Data Challenges for Enterprise Application Performance Management. pp. 1724–1735 (2012)
28. Sachs, K., Kounev, S., Bacon, J., Buchmann, A.: Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark. Perform. Eval. **66**(8), 410–434 (2009)
29. Sohan, S., Anslow, C., Maurer, F.: A case study of web API evolution. In: Proceedings of SERVICES, pp. 245–252. IEEE (2015)
30. Suter, P., Wittern, E.: Inferring web api descriptions from usage data. In: Proceedings of the 3rd IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb), pp. 7–12 (2015)
31. Wada, H., Fekete, A., Zhao, L., Lee, K., Liu, A.: Data consistency properties and the trade-offs in commercial cloud storages: the consumers' perspective. In: Proceedings of CIDR, pp. 134–143 (2011)
32. Wang, S., Keivanloo, I., Zou, Y.: How do developers react to RESTful API evolution? In: Franch, X., Ghose, A.K., Lewis, G.A., Bhiri, S. (eds.) ICSOC 2014. LNCS, vol. 8831, pp. 245–259. Springer, Heidelberg (2014)
33. Wittern, E., Laredo, J., Vukovic, M., Muthusamy, V., Slominski, A.: A graph-based data model for api ecosystem insights. In: Proceedings of ICWS, pp. 41–48. IEEE (2014)

34. Zellag, K., Kemme, B.: How consistent is your cloud application? In: Proceedings of SOCC. ACM (2012)
35. Zhao, L., Liu, A., Keung, J.: Evaluating cloud platform architecture with the CARE framework. In: Proceedings of APSEC, pp. 60–69 (2010)