Empirical Study on REST APIs usage in Android Mobile Applications

Mohamed A. Oumaziz[1], Abdelkarim Belkhir[2], Tristan Vacher[2], Eric Beaudry[2], Xavier Blanc[1], Jean-Rémy Falleri[1], and Naouel Moha[2]

[1] Univ. Bordeaux - LaBRI - UMR CNRS 5800, Talence, France
[2] LATECE, Département d'informatique, Université du Québec à Montréal, Canada
{moumaziz, xblanc, falleri}@labri.fr
{belkhir.abdelkarim, vacher.tristan}@courrier.uqam.ca
{beaudry.eric, moha.naouel}@uqam.ca

Abstract. A large set of mobile applications (apps) heavily rely on services accessible through the Web via REST APIs. The mobile apps market is therefore becoming a key market for service providers. However, the way in which mobile apps use services in practice has never been studied. In this paper, we perform an empirical study in the Android ecosystem in which we analyze 500 popular applications and 15 popular services. We also conducted an online survey to identify which are the best practices for Android developers. Our results show that Android developers generally favor invoking services by using an official service library instead of directly invoking services with a generic HTTP client. Our study shows that Android developers prefer to use a dedicated library developed by the service provider if it exists. The main advantages of such a library are that it simplifies the calls of the REST services but also provides added-value such as Android Views for example.

Keywords: Empirical Study, Mobile Applications, REST API, REST Services, Android

1 Introduction

Following the REST principles [6], server side applications are nowadays composed of several stateless independent micro-services [13]. They therefore make client side applications consuming more and more REST services [5]. Such evolution then brings new challenges especially for the design of Android Applications[3] that now have to handle lots of calls to REST services.

However, little is known on how Android apps use REST services in practice. Such knowledge is of high importance for the service providers since it would help them provide facilities to Android developers and hence improve the usability of their REST services. For instance, do the developers prefer to handle JSON

[3] There is now more than 3 million Android Applications: https://www.appbrain.com/stats

documents or Java objects? Do they want dedicated service libraries or do they want to perform the calls by using a HTTP client library?

In this article, we provide answers to these questions by proceeding to an empirical study in the famous Android ecosystem. Our study focuses on two research questions. Our first research question: "***As service users, how Android developers access popular REST services/APIs in their applications?***", aims at identifying the developers' habits for accessing REST services. Our study shows that Android developers prefer to use a dedicated service library developed by the service provider if it exists.

Our second question: "***As service providers, how to design client helper libraries to be popular among mobile applications?***", aims at identifying which features of service libraries are considered important by the Android developers. For instance, our study shows that the essential features for the developers are the existence of a complete documentation for the library, the library's vocabulary consistency with the service's one, the use of raw JSON to exchange data, the handling of authentication, and the ability to fine-tune the HTTP requests issued by the library.

This paper is structured as follows. Sections 2 and 3 respectively describe the study setup and results for research questions RQ1 and RQ2. Section 4 presents the related work. Section 5 concludes and presents future works.

2 Research Question 1: *As service users, how Android developers access popular REST services/APIs in their applications?*

In this section, we investigate our first research question. Based upon an informal look at the source code of several apps, we noticed that there are two main ways to access services from Android apps: directly by using an HTTP client, or by using a library developed by the provider of the services (official) or by its users (third-party). To assess which method is the most popular, we analyze how 15 popular services are used in a corpus of 500 popular apps. Section 2.1 explain in details how we build the corpus of services and apps. The Section 2.2 explains how apps invoke services. Results and observations are then presented in Section 2.3.

2.1 Corpus

Our corpus consists of two sets: a set of popular apps and a set of popular services that are called by the popular apps. Additionally, we also gather the list of libraries that allows Android apps to interact with the services of our corpus.

We started by gathering a set of popular apps. To that extent, we crawled the top 500 most popular apps list provided by the Google Play store[4]. We then download the APK (application packages) of each app by using the AndroZoo dataset[5] maintained by our colleagues from the University of Luxembourg [1].

[4] `https://play.google.com/store/apps/collection/topselling_free`
[5] `https://androzoo.uni.lu/`

During this step, we were only able to download 487 APK files. Our corpus then contains those 487 APK files.

To build the set of popular services, we analysed the 487 APK files to identify which popular services are called. We then used the AndroGuard tool[6] to extract all the strings contained in the DEX bytecode files of each of our apps. Among these strings, we extracted the ones corresponding to URLs (i.e. starting with `http://` or `https://`). We then ranked these URLs by their number of occurrences and filtered out the ones that do not correspond to a service (this step was done by manually browsing the URLs). Finally, we manually selected 15 services among the 50 most popular ones, avoiding selecting too many services from a same provider (such as Google for instance). Our final list of services is shown in Table 1.

To identify all the libraries that are dedicated to the services of our corpus, we then use the Google search engine with the following query "[service name] android library"; where "[service name]" corresponds to the root URL of the service. Then we manually look at the answers returned by Google to assess whether it describes an official library (developed by the service provider) or a third-party one.

2.2 Experimental setup

To answer our first research question, we check how apps call the services, and in particular if they directly use the service by making HTTP requests or if they use a dedicated library. To that extent, we first compute the services used by each app in our corpus. Secondly, we compute for each app if it uses a library or not to access the services. Using this data, for each service, we classify the apps into three categories: apps using the service without library, apps using the service with an official library, and apps using a third-party library.

To find out which applications are using a given service we sought additional information from the services. First we manually read the documentation of each service to find out their API URL. Secondly, we manually browsed the code of all libraries to find out the list of all the Java packages they contain. Finally, we used the AndroGuard tool again to extract all the strings contained in all applications from our corpus. When we are able to find a service's API URL, we consider that the application uses this service. In this case, we also look for the Java package names of this service's libraries in the strings of the application. When we are able to find a package name in the string list, we assumed that the application is using its corresponding library.

Finally, to analyze how developers of Android applications access services in practice, we perform the following process. For each service provider, we compute the set of all Android applications from our corpus that use it. Then, we partition this set into three subsets: the set of applications that use the official library, the set of applications that use a third-party library or both the official library and a third-party library, and the set of applications that do not use any library. To

[6] `https://github.com/androguard/androguard`

discuss the favourite way of developers to access the service, we then compare the size of these subsets, normalized by the size of all applications that use the services. Results of this experiment are discussed in Section 2.3.

2.3 Results

The results of the experiment described in Section 2.2 are shown in Table 1. In this table, we show for each service, the number of available official and third-party libraries (columns 2 and 3), the number of client applications (column 4) and the ratio of client applications using an official, a third-party or an official and no library (through an HTTP Client), respectively in columns 5, 6 and 7. Moreover, we have highlighted in bold the preferred way to use the service for each service.

We can first notice that only 5 out of 15 services are accessed with a HTTP client rather than a library. Moreover, 2 out of the 5 services provide no official library (Instagram and OpenStreetMap). Therefore, developers favour libraries to access services. Additionally, for the 10 services where a library is preferred, it is always the official library that is preferred, even if there are only 3 cases where no third-party library is available. In conclusion, official libraries are the favourite way of developers to access services.

Table 1. Extracted results about the services of our corpus: number of available official and third-party libraries, number of client applications, ratio of clients using an official, a third-party or both third-party and official, and no library.

	Available libraries		Client applications			
Service	Official	3rd-party	All	Official	3rd-party	None
AWS	1	2	73	36%	0%	**64%**
Crashlytics	1	0	84	**70%**	0%	30%
Digits	1	0	11	**100%**	0%	0 %
Dropbox	1	1	15	**53%**	0%	47%
Facebook	1	4	245	**98%**	0%	2%
Firebase	1	2	59	**95%**	0%	5%
Flurry	1	1	86	**68%**	30%	2%
GoogleMaps	2	4	20	**70%**	0%	30%
GoogleSignIn	1	0	394	**71%**	0%	29%
Instagram	0	2	6	0%	0%	**100%**
LinkedIn	1	4	6	0%	0%	**100%**
OpenStreetMap	0	3	1	0%	0%	**100%**
Paypal	1	2	7	**100%**	0%	0 %
Twitter	1	5	46	**39%**	28%	33%
YouTube	2	1	53	23%	0%	**77%**

Table 2 reports which HTTP clients are used by popular apps. Although there are many HTTP clients, developers still prefer standard ones that are embedded in the Android Framework (highlighted in bold), which represents the top four in our ranking. We also notice that developers tend to use more than just one HTTP client, this can be related to the features that each client offers depending on developer's needs. For instance, *HttpsUrlConnection* is able to deal with HTTPS requests while *HttpUrlConnection* only deals with HTTP requests.

Table 2. Extracted results from our corpus about the HTTP clients used in apps that are using services: number of apps using the client, percentage of apps using a service with the client.

HTTP Client	Count	% of all apps
HttpUrlConnection	204	**98.07**%
HttpsUrlConnection	178	**85.57**%
DefaultHttpClient	171	**82.21**%
AndroidHttpclient	134	**64.42**%
OkHttpClient	53	25.48%
Retrofit	35	16.82%
Volley	20	9.61%
AsyncHttpClient (loopj)	9	4.32%
GoogleHttpClient	6	2.88%
AsyncHttpClient (koushikdutta)	1	0.48%
RoboSpice (OkHttp)	1	0.48%
RoboSpice Basic	1	0.48%
Fuel	0	0.0%
RoboSpice (Retrofit)	0	0.0%
RoboSpice (GoogleHttpClient)	0	0.0%
AndroidAnnotations	0	0.0%

2.4 Threats to validity

We discuss here the threats to validity of our study following the guidelines provided by Wohlin et al. [22].

Internal validity threats concern the causal relationship between the treatment and the outcome. The techniques we used to detect client libraries and API's URLs are not infallible. We are aware about few limits of those techniques. For instance, if the binary code of an application is obfuscated, our techniques probably fail to identify URLs and used libraries. Another difficult case is the construction of URLs by string concatenations. Since we made a static analysis, we cannot catch all possible strings that could be built at runtime. Finally, we

had to manually look at all available libraries for each service in our dataset. We may have missed few of them.

External validity threats concern the possibility to generalize our findings. Further validation should be done with a bigger corpus. Our corpus only contains 15 services and about 500 apps. Therefore our results might be too specific to our corpus and not generalizable to all Android apps.

Reliability validity threats concern the possibility of replicating this study. We attempt to provide all the necessary details to replicate our study and our analysis. Furthermore, python scripts and the dataset used in this study are available on-line to leverage its reproduction[7].

3 Research Question 2: *As service providers, how to design client helper libraries to be popular among mobile applications?*

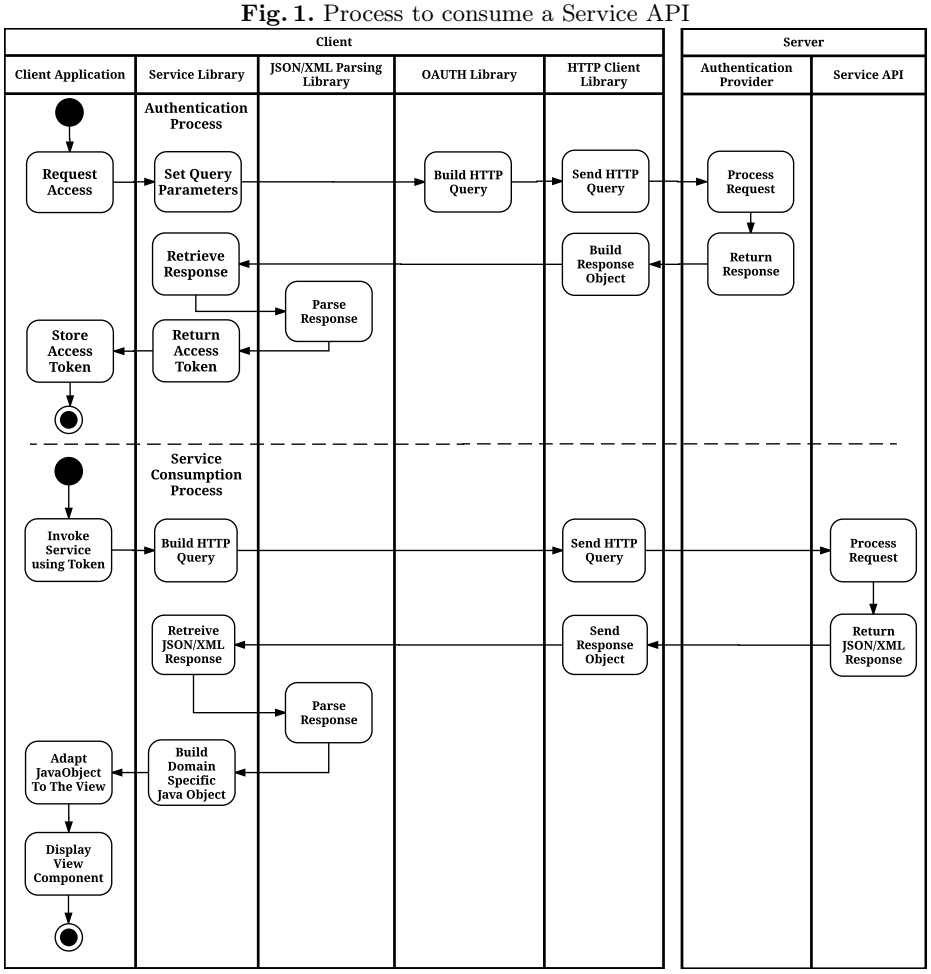
To answer the second research question, we proceeded as follows. As a first step, we studied the technical process that apps follow to call a service, and the different kinds of libraries used under the hood. This study allows us to have a good understanding of such a process with its different steps (such as creating and sending the requests). Then, as a second step, from this process we identified the good versus bad practices that should be followed when designing a service library to ease its interaction with the app. In the third step, we conducted an online survey that was sent to Android developers to validate these good and bad practices by experts. As a final step, we analysed the official service libraries provided by popular service to verify if the latter are conform to these practices. We detail in the following each of these steps.

Step 1. Process to consume a service. Figure 1 illustrates the general process followed by any app to call a service. It presents all the actions, mandatory and optional, that have to be done. Furthermore, for each action, it identifies which library can be used to automate it. The process starts with a first optional authentication sub-process where the client asks for an access right. In this sub-process the app interacts with the service library that uses a parsing library (to read and write JSON or XML document), a OAUTH library (to ask for a permission) and a HTTP Client library (to deal with the HTTP protocol). The process then continues with the service consumption sub-process where the client interacts with the service. In that sub-process the app is also using the service, parser and HTTP libraries.

Step 2. List of good and bad practices when developing a client library. We identify here the good versus bad practices that must be followed when designing a service library, as reported in Table 4. The practices are sorted in three different categories : Query, Code, and Features.

[7] `http://se.labri.fr/a/ICSOC17-oumaziz`

Fig. 1. Process to consume a Service API



Step 3. Online survey to validate the good and bad practices. The goal of the survey is to confirm the best practices that must be followed by service providers in their libraries to ease the authentication and service consumption by Android developers. The survey is available online[8]. Based on the good and bad practices identified in *Step 2*, we build a survey on Google Forms and emailed it to 2000 Android developers that we randomly selected from the top 500 Android apps developers for each category in the Google Play. We also submitted the survey as Reddit Thread on the very active subreddit *Androiddev*[9], and advertised the survey through the social networks. 51 Android developers responded to our

[8] `http://bit.ly/clientpractices`
[9] `https://www.reddit.com/r/androiddev/`

survey and 83% of them are familiar with Android application development. The survey and its results are available on our website[10].

Step 4. Analysis of the official REST libraries. As a final step, we manually analysed the 11 libraries and 14 services from our corpus. We did not analyse the OpenStreetMap and Instagram services because no libraries are available for these services, and the Google API Client library groups the GoogleMaps, GoogleSignIn and YouTube services: . These libraries and services have been analysed by three experienced Android developers to verify their conformance with the practices identified and validated in the two previous steps. We performed this analysis using their documentation, their source code, their provided examples and/or typical client mobile apps. The latter were provided either by the libraries as examples or available on GitHub. The results of this analysis are reported in Table 3, and we discuss them in the following section.

3.1 Results of Research Question 2

For each good/bad practice listed in Table 4, we discuss the related results reported in Table 3 followed by a discussion, highlighted in bold, on the results of the survey.

① *JSON vs. XML.* As shown in Table 3, it seems that APIs favour the JSON format over others. Every library allows to return at least a response in the JSON format and provides sometimes at least another format (XML, CSV, etc.). Although JSON is the most popular format, it is not used by default by all libraries. For example, LinkedIn API uses XML by default over JSON. **This is also confirmed by the results of our survey, where 92.2% of the developers stated that JSON was preferred.** This result can be explained by the fact that the JSON format is easier to handle, but also the loading and reading of JSON files are especially faster on iOS and Android compared to XML files [2].

② *Typed Response vs. Non-typed Response.* As shown in Table 3, over the 11 libraries we studied, six return a domain-specific object representing an entity of the API (e.g. a File in the DropBox Library). It means users do not need to parse the response sent by the API. They have already their responses typed and formatted. On the other hand, some providers such as AWS, Facebook, and LinkedIn return an object containing data. For example, Facebook returns a GraphResponse object that contains the response of a request, which can be either a JSONObject, a JSONArray, or a Java String. The user has then the choice of the format that is the most convenient for his use. **In contrast, the results of our survey indicate that more than 70.6% of Android developers actually prefer to have responses as Raw service data (Java Strings). Hence, they can use their own parser and control the way they handle their responses.**

[10] `http://se.labri.fr/a/ICSOC17-oumaziz/`

Table 3. Results of the manual analysis of service libraries practices

	Query			Code			Features	
Service Library	① JSON vs. XML	② Typed Response	③ Enc. HTTP Query	④ Full Supp	⑤ Consistent Voc.	⑥ Documented	⑦ Auth	⑧ Android Funcs
AWS SDK	✓	Parser-spec Obj	✓	✓	✓	✓	✓	
Crashlytics		No resp.	✓	✓	✓	✓	✓	
Digits		Domain-spec Obj	✓	✓	✓	✓	✓	✓
Dropbox Java SDK	✓	Domain-spec Obj	✓	✓	✓	✓	✓	✓
Facebook SDK	✓	Parser-spec Obj	✓	✓	✓	✓	✓	✓
Firebase SDK for Android	✓	Domain-spec Obj	✓	✓	✓	✓	✓	
Flurry	✓	Domain-spec Obj	✓	✓	✓	✓	✓	
Google API Client for Java	✓	Domain-spec Obj	✓	✓	✓	✓	✓	
LinkedIn SDK		Parser-spec Obj		✓	✓	✓	✓	
Paypal Android SDK	✓	No resp.	✓	✓	✓	✓	✓	✓
Twitter Android SDK	✓	Domain-spec Obj	✓	✓	✓	✓	✓	✓

③ *Encapsulated HTTP Queries vs. Non-encapsulated HTTP Queries.* Almost all (10 out of 11) libraries except LinkedIn encapsulate HTTP queries. Users do not have to build their own requests, they can use predefined methods. However, some libraries such as Facebook allow users to build their own requests while providing encapsulated queries. Therefore, this eases the building of "generic" requests where data to retrieve are quite similar. **In the survey, 37.3% of developers consider that even if queries are encapsulated, having the possibility modify them is mandatory, and 47.1% of them think that it would be appreciated. Therefore, although the majority of libraries encapsulate queries, developers still prefer to have access and control the queries.**

④ *Full API support vs. Non-exhaustive API support.* All libraries provide the objects and methods to cover all services declared in the API. **The survey indicates that 58.8% of developers would appreciate to have a complete library, and 27.5% of them consider that it should be mandatory.**

⑤ *Consistent vs. Inconsistent vocabulary with documentation.* All libraries without exception have a consistent vocabulary with the documentation provided in the API. **This observation is confirmed in the survey, where 60.8% of developers answered that consistency is mandatory and 35.3% that it would be appreciated.**

⑥ *Documented vs. Non-documented Library.* All libraries studied provide a documentation on how they should be used. The documentation is presented in different forms: details about specific objects or methods, examples, application samples, etc. **In the survey, 70.6% of developers judge that such documentation is mandatory, while only 3.9% judge that it is not important.**

⑦ *Allowing Authentication vs. Third-party Authentication.* All services analysed require authentication to be used. Therefore, the provider can restrict the data used by a developer or an end-user. Authentication is a means to secure which data are reachable to someone, but also to control the request flow for avoiding overloading servers. One important point to emphasis is that all libraries implement the entire service authentication protocol, namely OAuth2. **In the survey, developers confirm the necessity to implement the whole service authentication protocol with 58.8% who appreciate a library providing authentication and 29.4% who request it to be mandatory.**

⑧ *Providing Android Specific Functionalities vs. Providing Only General Functionalities.* Libraries seem to have different points of view for providing Android specific functionalities such as Widgets, Views, or Activities. Almost half of the libraries (5 out of 11) provide at least one Android specific functionality such as Widgets, Activities, or Views. Providing such functionalities to developers could help them to focus on their own applications instead of trying to integrate

logic from a third-party environment. **However, the majority of developers (86.3%) in the study consider that providing such functionalities is not important for a library.**

3.2 Threats to Validity

In this section, we discuss the threats to validity using the same method as in Section 2.4. The terminology used in the survey might have been misunderstood by the responders. However we wrote definitions and examples to mitigate these threats. Our survey was answered by only 51 Android developers. Therefore, our findings might not be generalizable to all Android developers.

4 Related Work

In the following, we discuss some relevant research done on assessing bad and good practices in REST APIs as well as research on libraries identification.

4.1 Bad and good REST practices

In [15,16,17], we evaluated the design of several REST APIs based on good and bad REST practices, also called REST patterns and antipatterns. We proposed automatic approaches for the detection of these practices. However, we specifically evaluated APIs without considering any interaction with clients, and in particular mobile clients, as we do here. Other works have followed proposing similar (anti-)patterns detection approaches in service applications, but implementing other techniques such as bi-level optimisation problems [20] or ontologies [4].

In [18], Rodriguez et al. evaluated the conformance of design best practices in REST APIs from the perspective of mobile applications. They analysed these practices on a large dataset of HTTP requests collected from a Mobile Internet traffic. This work is the first that has studied the traffic of HTTP requests from the mobile perspective. However, the best practices analysed are rather common to any kinds of REST APIs, and they focus specifically on HTTP requests.

In contrast, in this paper, we consider practices that may apply on mobile apps. We take also into account the interaction between clients and REST APIs by analysing all the process from the authentication to the service consumption, and thus while considering all kinds of message exchanges (requests, responses). We study also how REST APIs are implemented and documented.

4.2 Libraries identification

There are several works that have been done in the past for identifying advertisement libraries in Android mobile apps. Book et al. [3] and Grace et al. [7] used a method based on whitelists to identify advertising libraries. To do this, they

Table 4. List of good and bad practices when developing a service library

Query Related Practices
① ✓JSON vs. ✗XML
Description: Always choose JSON over XML when both are proposed by the API provider.
Example: ✓*"students" : ["firstName" : "John","lastName" : "Doe"]* is a JSON response returned from the REST API. In opposite, ✗*< students >< student >< firstName > John < /firstName><lastName> Doe </lastName></student></students>* is the same response returned in an XML format.
Consequences: XML is a more human readable format than JSON, but the problem is that all the opening/closing tags in XML files make them heavier than JSON files. Not only is JSON format more compact, it is also easier and more CPU-friendly to parse [14].
② ✓Typed vs. ✗Non-typed Response
Description: The response returned from the library for a given query should be a Java Object. In contrast, a *Non-typed Response* is a response returned as a JSON or XML format.
Example: ✓**ArrayList<Post>** response is a ✓*Typed Response* returned by the method *GetTimeline*. this response can be directly used by the client developer. ✗**JSONObject** response is a ✗*Non-typed Response* returned by the method *GetTimeline*. This response should first be parsed into local usable objects.
Consequences: When working with *Non-Typed Responses*, an additional effort is needed to parse them depending on their format into usable Java Objects.
③ ✓Encapsulated vs. ✗Non-encapsulated HTTP Queries
Description: The HTTP query should be encapsulated in a method proposed by the interface of your library. A *Non-encapsulated HTTP Query* has to be manually built by the developer with all the needed parameters.
Example: The Method ✓**SearchUser(Text, AccessToken)** proposed by the library ✓*encapsulate the HTTP POST Query* : **GET https://www.example.com/v1/users/search?q={Text}&access_token={AccessToken}** In opposite, when using the method ✓**SearchUser(URL, AccessToken)** the HTTP URL query is visible in the code, this means that it is a ✗*Non-encapsulated HTTP Query*.
Consequences: Client developers may prefer to consume method-encapsulated HTTP queries than handling them directly. This eases the interaction with the API resources, by making its access points practical, understandable and their invocations more readable within the consuming code.
Code Related Practices
④ ✓Full vs. ✗Non-exhaustive API support
Description: The Service Library should cover all the services proposed by the REST API.
Consequences: It is less convenient to work with a non-exhaustive library. This will require client developers to either compensate the lack of coverage by handling the queries manually, or choosing a more exhaustive third-party library.
⑤ ✓Consistent vs. ✗ Inconsistent vocabulary with documentation
Description: The vocabulary used in the code when naming classes, methods and attributes should correspond to the one used in the documentation of the REST API.
Example: For Twitter, the concept of tweet is named as a ✓**Tweet** in both its REST API documentation and the code the client library. Opposite a bad practice would be to name the concept Tweet as ✗**Post** in the code of the library, because it is ✗*Not vocabulary-consistent* with the term *Tweet* used in the documentation.
Consequences: Using a different vocabulary in your code might be confusing and misleading to client developers. This will surely affect the understandability of your library. Having a consistent vocabulary eases the development task of developers since each entity in the API and the library has the same name.
⑥ ✓Documented vs. ✗Non-documented Library
Description: The library should be well documented, the user should be able to understand how to access the REST API endpoints preferably with code samples.
Consequences: As for any other software product, a well documented library will facilitate its use.

Features Related Practices
⑦ ✓**Allowing Authentication vs.** ✗**Third-party Authentication**
Description: When an authentication is required to consume the offered services by the REST API. It is preferable that your Service Library allows authentication.
Example: ✗**Apache Oltu**, ✗**Spring Security OAuth** are among the most common used third-parties client libraries that allows authentication.
Consequences: The lack of means of authentication will force the developers to use another third-party library.
⑧ ✓**Android Specific Functionalities vs.** ✗ **Only General Functionalities**
Description: A good practice is to provide some Android specific functionalities such as widgets, views and fragments instead of providing only general functionalities
Example: The method *GetPostView* returns an object of type ✓**PostView** witch could be directly added in the view of the client Android application.
Consequences: It is highly appreciable when developing Android clients to have ready-for-use components, this will make the client application more lightweight.

compared the packages included in each application with the ones they collected in their lists.

There are also other tools such as AdDetect [12] and PEDAL [10] that applied machine learning techniques (SVM classification) to identify advertising libraries even if applications are obfuscated.

Mileva et al. [11] perform an analysis of the Maven configuration files of 250 Apache projects to mine usage of libraries and their versions. Such an approach cannot be used in the context of Android applications as the configuration files are not available.

Teyton et al. [19] used a corpus composed of 8795 libraries where they applied static analysis on the source code of each library to automatically extract Java package names. They were able to identify 1185 different libraries which they then used to automatically identify Java libraries dependencies. Still in the same idea, Li et al. [8] identified the use of over 1,113 android libraries for common functionalities and 240 android ad libraries. To identify these libraries they used their package names. They also did a pairwise similarity analysis to determine what are the common libraries packages in Android applications.

Wang et al. [21] proposed a novel clustering-based technique to automatically identify android third-party libraries. Their experiments showed that their technique was able to identify more than 600 different android libraries in a corpus of 100,000 android applications.

Li et al. [9] proposed a novel approach for identifying third-party libraries from Android applications. Rather than using code similarity to identify the libraries, they proposed to use code dependencies. Using this approach, they were able to identify 19,540 obfuscated libraries.

In this paper we used the Java package names as a way to identify libraries as it was done by Teyton et al. [19] and Li et al. [8]. However, we had to identify not just libraries but service libraries, to do so, we used API URLs as a way to determine if an app was using a service and then we applied this library identification technique to look if it was through a service library.

5 Conclusion and Future Work

While nowadays Android apps rely more than ever on REST services, no study has been performed on how Android apps invoke services.

We alleviated this situation by performing an empirical study of 15 popular web services on a dataset of almost 500 popular Android apps. We show that developers prefer to use web service official libraries rather than using third-party ones. We also show that developers that prefer to use HTTP clients rather than libraries prefer the default clients provided in the Android Framework.

Second, we propose a list of good and bad practices, identified through an analysis of the practices of the popular services and an online survey involving 51 Android developers. We show that the important features for libraries are: the use of raw JSON to exchange data, the availability of a complete documentation, the use of a consistent vocabulary with the service, the handling authentication and the possibility for developers to fine-tune HTTP requests.

As a further work, we plan to extend our list of practices and to largely extend the size of our dataset of services and apps in order to have results that can be more generalisable.

Acknowledgement. The authors thank the Android developers for answering the survey. This study is supported by NSERC (Natural Sciences and Engineering Research Council of Canada) and FRQNT, Canada research grants.

References

1. Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y.: Androzoo: Collecting millions of android apps for the research community. In: Proceedings of the 13th International Conference on Mining Software Repositories. pp. 468–471. ACM (2016)
2. Betts, T.: Mobile performance testing - json vs xml. Blog (Accessed in June 20th, 2017), `https://www.infragistics.com/community/blogs/torrey-betts/archive/2016/04/19/mobile-performance-testing-json-vs-xml.aspx`
3. Book, T., Pridgen, A., Wallach, D.S.: Longitudinal analysis of android ad library permissions. arXiv preprint arXiv:1303.0857 (2013)
4. Brabra, H., Mtibaa, A., Sliman, L., Gaaloul, W., Benatallah, B., Gargouri, F.: Detecting cloud (anti)patterns: OCCI perspective. In: 14th International Conference on Service-Oriented Computing. Lecture Notes in Computer Science, vol. 9936, pp. 202–218. Springer (2016)
5. Danielsen, P.J., Jeffrey, A.: Validation and interactivity of web API documentation. In: 20th International Conference on Web Services. pp. 523–530 (2013)
6. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine (2000)
7. Grace, M.C., Zhou, W., Jiang, X., Sadeghi, A.R.: Unsafe exposure analysis of mobile in-app advertisements. In: Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks. pp. 101–112. ACM (2012)
8. Li, L., Bissyandé, T.F., Klein, J., Le Traon, Y.: An investigation into the use of common libraries in android apps. In: 23rd International Conference on Software Analysis, Evolution, and Reengineering. vol. 1, pp. 403–414. IEEE (2016)

9. Li, M., Wang, W., Wang, P., Wang, S., Wu, D., Liu, J., Xue, R., Huo, W.: Libd: Scalable and precise third-party library detection in android markets. In: 39th International Conference on Software Engineering. pp. 335–346. IEEE Press (2017)
10. Liu, B., Liu, B., Jin, H., Govindan, R.: Efficient privilege de-escalation for ad libraries in mobile apps. In: Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services. pp. 89–103. ACM (2015)
11. Mileva, Y.M., Dallmeier, V., Burger, M., Zeller, A.: Mining trends of library usage. In: Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops. pp. 57–62. ACM (2009)
12. Narayanan, A., Chen, L., Chan, C.K.: Addetect: Automated detection of android ad libraries using semantic analysis. In: Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on. pp. 1–6. IEEE (2014)
13. Newman, S.: Building microservices - designing fine-grained systems, 1st Edition. O'Reilly (2015)
14. Nurseitov, N., Paulson, M., Reynolds, R., Izurieta, C.: Comparison of JSON and XML data interchange formats: A case study. In: 22nd International Conference on Computer Applications in Industry and Engineering. pp. 157–162 (2009)
15. Palma, F., Dubois, J., Moha, N., Guéhéneuc, Y.: Detection of REST patterns and antipatterns: A heuristics-based approach. In: 12th International Conference on Service-Oriented Computing. Lecture Notes in Computer Science, vol. 8831, pp. 230–244. Springer (2014)
16. Palma, F., Gonzalez-Huerta, J., Moha, N., Guéhéneuc, Y., Tremblay, G.: Are restful apis well-designed? detection of their linguistic (anti)patterns. In: 13th International Conference on Service-Oriented Computing. Lecture Notes in Computer Science, vol. 9435, pp. 171–187. Springer (2015)
17. Petrillo, F., Merle, P., Moha, N., Guéhéneuc, Y.: Are REST apis for cloud computing well-designed? an exploratory study. In: 14th International Conference on Service-Oriented Computing. Lecture Notes in Computer Science, vol. 9936, pp. 157–170. Springer (2016)
18. Rodríguez, C., Báez, M., Daniel, F., Casati, F., Trabucco, J.C., Canali, L., Percannella, G.: REST apis: A large-scale analysis of compliance with principles and best practices. In: 16th International Conference Web Engineering. Lecture Notes in Computer Science, vol. 9671, pp. 21–39. Springer (2016)
19. Teyton, C., Falleri, J.R., Palyart, M., Blanc, X.: A study of library migrations in java. Journal of Software: Evolution and Process 26(11), 1030–1052 (2014)
20. Wang, H., Kessentini, M., Ouni, A.: Bi-level identification of web service defects. In: 14th International Conference on Service-Oriented Computing. Lecture Notes in Computer Science, vol. 9936, pp. 352–368. Springer (2016)
21. Wang, H., Guo, Y., Ma, Z., Chen, X.: Wukong: a scalable and accurate two-phase approach to android app clone detection. In: Proceedings of the 2015 International Symposium on Software Testing and Analysis. pp. 71–82. ACM (2015)
22. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in software engineering. Springer Science & Business Media (2012)