TU Delft

Delft University of Technology

Web API growing pains
Loosely coupled yet strongly tied

Espinha, Tiago; Zaidman, Andy; Gross, Gerd

DOI
10.1016/j.jss.2014.10.014

Publication date
2015

Document Version
Submitted manuscript

Published in
Journal of Systems and Software

Citation (APA)
Espinha, T., Zaidman, A., & Gross, H. G. (2015). Web API growing pains: Loosely coupled yet strongly tied. Journal of Systems and Software, 100, 27-43. https://doi.org/10.1016/j.jss.2014.10.014

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Delft University of Technology
Software Engineering Research Group
Technical Report Series

Web API Growing Pains: Loosely Coupled yet Strongly Tied

Tiago Espinha, Andy Zaidman and Hans-Gerhard Gross

Report TUD-SERG-2014-017

TUDelft

SERG

TUD-SERG-2014-017

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
`http://www.se.ewi.tudelft.nl/techreports/`

For more information about the Software Engineering Research Group:
`http://www.se.ewi.tudelft.nl/`

© copyright 2014, by the authors of this report. Software Engineering Research Group, Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology. All rights reserved. No part of this series may be reproduced in any form or by any means without prior written permission of the authors.

Web API Growing Pains: Loosely Coupled yet Strongly Tied

Tiago Espinha[a], Andy Zaidman[a], Hans-Gerhard Gross[b]

[a] *Delft University of Technology, Faculty EEMCS, Mekelweg 4, 2628 CD Delft, The Netherlands*
[b] *Esslingen University, Faculty of Information Technology, Flandernstrasse 101, 73732 Esslingen, Germany*

Abstract

Web APIs provide a systematic and extensible approach for application-to-application interaction. Developers using web APIs are forced to accompany the API providers in their software evolution tasks. In order to understand the distress caused by this imposition on web API client developers we perform a semi-structured interview with six such developers. We also investigate how major web API providers organize their API evolution, and we explore how this affects source code changes of their clients. Our exploratory qualitative study of the Twitter, Google Maps, Facebook and Netflix web APIs analyzes the state of web API evolution practices and provides insight into the impact of service evolution on client software. In order to complement the picture and also understand how web API providers deal with evolution, we investigate the server-side and client-side evolution of two open-source web APIs, namely VirtualBox and XBMC. Our study is complemented with a set of observations regarding best practices for web API evolution.

Keywords: Web API, software evolution, breaking changes, Web Services

Email addresses: t.a.espinha@tudelft.nl (Tiago Espinha), a.e.zaidman@tudelft.nl (Andy Zaidman), Hans-Gerhard.Gross@hs-esslingen.de (Hans-Gerhard Gross)

Preprint submitted to Elsevier *October 8, 2014*

1. Introduction

Modern-day software development is inseparable from the use of Application Programming Interfaces (APIs) [1, 2]. Software developers access APIs as interfaces for code libraries, frameworks or sources of data, to free themselves from low-level programming tasks and/or speed up development [3]. In contrast to statically linked APIs, a new breed of APIs, so called web service APIs, offer a systematic and extensible approach to integrate services into (existing) applications [4, 5]. However, what happens when these web APIs start to evolve? Lehman and Belady emphasize the importance of evolution for software to stay successful [6], and updating software to the latest version of its components, accessed through APIs [7]. In the context of statically linked APIs, Dig and Johnson state that *breaking changes* to interfaces can be numerous [7], and Laitinen says that, unless there is a high return-on-investment, developers will not migrate to a newer version [8].

In the context of web APIs, developers can no longer afford the inertia that was noted by Laitinen, as it is the web API provider that sets the pace when it comes to migrating to a new version of the web API. In the statically linked API context, developers could choose to stay with an older version of e.g. libxml, which meets their needs, yet, with web service APIs the provider can at any time unplug a specific version (and functionality), thus forcing an upgrade. In 2011, a study by Lämmel et al. showed that among 1,476 Sourceforge projects the median number of statically linked APIs used is 4 [9]. Should developers have no control over the API evolution (as is the case with web APIs), this would represent a heavy burden for client developers as it causes an endless struggle to keep up with changes pushed by the web API providers.

Also in 2011, a survey among 130 web API client developers entitled "API Integration Pain" [10] revealed a large number of complaints about current API providers. The authors reported the following regarding web API providers: *"[...] There's bad documentation. [...] APIs randomly change without warning. And there's nothing even resembling industry standards, just best practices that*

everyone finds a way around. As developers, we build our livelihoods on these APIs, and we deserve better."

Pautasso and Wilde present different facets of *"loose coupling"* [11] on web services. Indeed, all web APIs which make use of REST interfaces can be easily integrated with through a single HTTP request. However, a facet not considered in Pautasso and Wilde's work is that of how clients end up tightly tied to the evolution policies of the web API providers. This motivated us to investigate how web service APIs evolve and to study the consequences for clients of these web APIs.

In this exploratory qualitative study, we start by investigating [**RQ1**] what some of the pains from client developers are when evolving their clients to make use of the newest version of a web API. We do this by interviewing six professional developers that work with changing web APIs. Subsequently, we investigate the guidelines provided by 4 well-known and frequently used web API providers to find out [**RQ2**] what are the commonalities in the evolution policies for web APIs? Ultimately, we turn our attention to the source code. We do so by analyzing the code of several web API clients to find out [**RQ3**] what the impact on the source code of the web API clients is when the APIs start to evolve. Extending our previous study [12], we are also turning our attention to the impact of evolution at the server-side of a web API, more precisely, we are asking ourselves [**RQ4**] whether web API providers take precautions in order to ease evolution pains of web APIs? We do so by analyzing the source code impact on two different case studies of web API provider and its respective client.

The remainder of this paper is structured as follows: in Section 2 we first explain some terminology regarding web APIs. Subsequently, in Section 3.1 we describe our experimental setup for the client-side study including how the projects were selected and how we calculate the impact on code. Section 4.1 describes the interviews with client developers and the lessons learned across different domains. Section 4.2 looks at the different web API policies from different providers, while Section 4.3 presents the impact web API evolution has on client code. Section 3.2 describes our experimental setup for the server-

side study including how the projects were selected and an introduction to the analyses we performed. Section 5 presents the results of the server-side analysis for the two case studies and Section 6 frames these results with our research questions and provides a list of recommendations for API providers. Lastly, we discuss related work in Section 7 and present our conclusions in Section 8.

2. Terminology

Throughout this paper we refer to different terms in the context of web APIs. Indeed, the concept itself of a web API is somewhat ambiguous and is, in our definition, no different from a web service. Already Alonso et al [13] report that *"the term Web services is used very often nowadays, although not always with the same meaning"*. The authors then resort to the W3C definition which states that a web service is a *"software application identified by a URI, whose interfaces and bindings are capable of being defined, described and discovered as XML artifacts"*. While this definition is for the most part correct, it restricts the technology to XML for which there are currently alternatives (such as JSON) as it is shown in this paper. In the context of web APIs, different technologies also translate into different challenges faced by the developers which should be considered. In our study we encountered three major implementation approaches which we will briefly describe in the sub-sections below.

2.1. SOAP

The invocation of SOAP[1], originally defined as Simple Object Access Protocol, web APIs is most commonly performed through the sending of an XML document (where the method name as well as arguments are defined) over an HTTP media to the server. Before the invocation occurs, clients request the WSDL file (Web Service Description Language) which defines both which methods are available for invocation as well as which data types the web API expects.

[1]SOAP —`http://www.w3.org/TR/soap12-part1/`

2.2. REST

Representational state transfer (REST)[2] is an architectural style originally defined by Roy Fielding [14]. Apigee's booklet on web API design [3] provides a clearer overview of what constitutes a REST web API as well as a set of practical guidelines.

Essentially, a REST web API relies on entities and basic CRUD (Create, Read, Update, Delete) actions on those entities. For example, when a web API client would like to get data on the 'Customer' entity with id '1337' it would send a HTTP GET request to the address /customer/1337.

Such an approach does not allow for method invocation and Apigee supports that in certain particular cases, the 'entity' can actually be an action (i.e. a method) to be invoked by the web API server. In such cases, the client can then send a payload (specifying which method to invoke and its arguments) typically using JSON (c.f. next subsection).

An important distinction between SOAP and REST is the fact that while a WSDL equivalent exists (the WADL file), it is seldom used in practice [15]. The more common alternative is human-readable documentation usually through means of a wiki. This lack (in comparison to SOAP) becomes evident by the existence of software which specializes in generating documentation for REST web APIs[4].

2.3. JSON-RPC & JSON

The JSON-RPC[5] (JavaScript Object Notation — Remote Procedure Call) approach shares a similarity with REST: it uses JSON as the data format for requests. In fact, oftentimes web APIs claim to be RESTful while in fact a flavor of JSON-RPC is used. JSON-RPC (where RPC stands for Remote Procedure Call) provides a mechanism for one software system to be able to invoke methods

[2]REST — `https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm`
[3]Web API Design — `http://bit.ly/apigee-web-api-design`
[4]MireDot — `http://www.miredot.com/`
[5]JSON-RPC — `http://www.jsonrpc.org/specification`

on another software system over the network. It is therefore, not an architectural pattern as REST and is bound to less restrictions.

The remote procedure calls are also made over HTTP and use JSON to specify which method should be invoked as well as the payload (arguments and responses). JSON in turn is used to dynamically construct types composed of key:value pairs, where the key is a string and the value is either an object or an array of objects. This contrasts with SOAP's XML-based invocations where types are defined statically and interactions are much more verbose.

3. Experimental Setup

Our exploratory study is comprised of two subsidiary studies. In the first part of our study we start by investigating the impact web API evolution has on clients for a group of high-profile web API providers. However, due to the closed nature of these web API providers, nothing can be learned from the potential pains web API providers also face when their web APIs must evolve. Therefore, with the second part of our study, we selected two open-source projects as well as a client for each of these projects as an attempt to shine a light on the web API providers' side of the story.

The two subsections below describe the experimental setup for each of these two parts of the study.

3.1. Experimental Setup for the Client-Side Investigation

In order to perform our client-side study which is exploratory in nature, we divided the study in three steps. We started by interviewing six developers (Table 1) who maintain clients for web APIs as to obtain anecdotal evidence of developers who had to undergo web API evolution in their clients. While we expected the developer interviews to provide fruitful insight into the evolution of web APIs, we also knew in advance that developer interviews are always subject to some degree of personal bias. The second step of our study therefore focused on analyzing objective software evolution metadata regarding the web APIs. Namely, we analyze the evolution policies (i.e. deprecation periods, breaking

change notifications, etc) from four major web API providers. This allows us to identify potential best practices. The last step of our study is to analyze the main artifact where the web API evolution task may cause more or less impact: the source code. We measure and interpret the impact web API evolution has on client code by analyzing code churn and identifying the commits related to web API evolution.

In this section we provide more insight on how we selected the developers to be interviewed, how we selected the projects under analysis as well as how we measure the impact on the client code.

3.1.1. Interviews With The Developers

Our experiment included interviews with several developers who have at some point dealt with evolving web APIs. In order to find suitable candidates we e-mailed the developers of all the clients under study (see Section 4.3) and sent out public calls for participation on social networks. Ultimately, due to a low response rate from the approached developers, we interviewed all developers who accepted to participate in the study.

Additionally we had the opportunity to interview the client developers of a multi-national payment aggregator company whose software system interacts with several financial institutions through web APIs.

Table 1 provides an overview of the web APIs each of the 6 interviewees

API	Developers	Observations
Google Maps	1 developer	Single developer. Creator and sole developer of the project since its creation in 2009.
Google Search & Bing	1 developer	Single developer. Creator and sole developer of the project since its creation in 2012.
Redmine API	1 developer	Developer is part of a larger team of 13 developers. Started in 2005.
Google Calendar	1 developer	Single developer. Creator and sole developer of the project since its creation in 2006.
Unnamed Payments Aggregator	2 developers	Two professional developers, part of a larger team.

Table 1: Interviewed Developers

developed clients for.

The interviews took on average thirty minutes per developer and were performed by the first author of the paper either face-to-face or via Skype in the format of a semi-structured interview [16]. The ten starting questions that we used during the interview are listed in Table 2 and cover several web API-related issues, such as: maintenance effort, frequency of version upgrades, security, developer communication and implementation technologies.

As for the analysis of the collected data, we followed the guidelines set forward by Creswell [17] for qualitative research. The interviews were recorded and further transcribed by the first author. Subsequently, the first and second authors read the transcriptions and established a basic coding in dialogue. This basic coding resulted in the identification of the categories or overall themes as they are reflected in the sections 4.1.x. Once we identified the categories, the first author did the actual coding.

3.1.2. Selecting Web APIs

In order to perform our code analysis we required web APIs with a large number of clients. To find such web APIs we resorted to ProgrammableWeb's [6] web services directory. From this list, sorted by popularity, we picked the top most popular web APIs and quickly verified which web APIs contained the largest number of references in GitHub. This led us to choose Twitter, Google Maps and Facebook. The projects using the Netflix web API were found while investigating projects on GitHub.

3.1.3. Selecting The Projects With Web API Evolution

Once we have selected a set of web APIs that are known to have evolved, we have to find candidate projects integrating with those web APIs. Candidate projects for our analysis need to meet the criteria of having had to perform maintenance due to the web API having changed. In order to have access to projects which contain this evolution step and thus shine a light on the amount

[6] Web Services Directory — `http://bit.ly/web-services-directory`, last visited October 3rd 2013

of changes involved in web API evolution we devised a mechanism to identify the evolution step.

This mechanism was then applied on GitHub as it contains a large collection of potentially suitable open-source projects.

For web API providers such as Twitter, Google Maps and Netflix, where an explicit versioning system is provided, the approach consists of two steps. They are: 1) compiling a list of all the projects on GitHub which contain references to the latest version of their specific web API, and 2) for each project found, filter the Git diffs which contain references to the old version of the web API.

Facebook required a different approach. Even though a booklet on web API design by Apigee[7] emphasizes the importance of versioning by dubbing it *"one of the most important considerations"* and advising developers to *"never release an API without a version"*, Facebook violates this principle. Because there is no version number involved in the requests, our search is done by querying the GitHub repositories for small pieces of code which were reported in Facebook Developer's blog[8] as having been changed.

3.1.4. Impact evaluation

The goal of the paper is to investigate how web service APIs evolve, and how this affects their clients. So, for each project, we looked at the commits right before and right after the first commit containing references to the new version of a web API. This was done to identify potential initial preparations prior to bringing a new API online, as well as to check for a potential fallout effect caused by switching to the new API.

In order to estimate the impact involved in maintaining the clients of a web service API, we start by using the code churn metric [18], which we define for each file as

$$FileCodeChurn = \frac{LOCAdded + LOCChanged}{TotalLOC} \quad (1)$$

The code churn we analyze and display in Table 4 (Avg. Churn) represents the average code churn for each commit. Of note is the fact that the churn presented does not count added files. Additionally, the *evolution churn* presented in the

[7]Web API Design — `http://bit.ly/apigee-web-api-design`
[8]Completed Changes — `http://bit.ly/fb-completedchanges`

table consists of the churn caused by the evolution-related code changes. This churn is determined manually and through visual inspection of the evolution-related commits. This is done manually to ensure that all the churn considered in the evolution commits is indeed related to the evolution task. The percentage presented is then how this evolution churn compares to the average. With the data we collected we are also able to plot graphs showing the code churn per commit. This way we can also visually identify abnormally high code churn peaks as well as churn peaks surrounding the evolution related commits. These peaks are potential candidates for web API-related maintenance and are then investigated in more detail by looking at the source code and commit messages.

While code churn provides a good starting point for assessing the impact of a maintenance task, it does not provide the whole picture: the nature of the code change, the number of files involved and their dispersion also play a role in determining the impact of a change. Hence, we also provide a more in-depth view of how the API migration affects a particular project. This is done by looking at the number of source code files changed, and analyzing the nature of the changes (e.g. file dispersion, actual code changes, whether the API-related files are changed again). This analysis also allows us to mitigate the code churn's indifference to the complexity of code changes.

3.2. Experimental Setup For The End-to-End Analysis

While it is important to consider the impact web API changes have on clients, this relationship is bidirectional and minimizing client-side impact can be achieved if the web API provider is mindful of changes to the web API.

This is particularly important due, again, to how in a web API relationship the provider and clients are tied to the web API provider's web API evolution policy.

To further our study, we perform an end-to-end analysis, meaning that we study how the web API changes from the provider's perspective and how this impacts the client. Ideally, we would have liked to analyze the server-side code of the web APIs mentioned before. However, due to the closed nature of such

software systems, we resort to case studies using open-source systems (where both the provider and clients are open-source) to study these added aspects of web API maintenance.

3.2.1. Project Selection

For this part of our experiment we chose Oracle Virtualbox[9] and XBMC[10] since both are large projects (∼4 million and ∼2.2 million SLOC respectively) and for both there is a client available that is actively maintained and used. Additionally these two projects use different web API technologies (VirtualBox uses SOAP whereas XBMC uses JSON-RPC) which, on both the server and client side, also plays a role in how much code needs to be changed and under what circumstances.

Together with these two projects we analyzed a web API client for each case. For VirtualBox we studied phpVirtualBox[11] which is a feature-complete web-based GUI for VirtualBox and is endorsed by Oracle as a *"hot pick"* on the main page (even though it is developed by a third party developer). For XBMC we analyzed the Android-XBMC client as it is the official client for this web API (developed by a subset of the XBMC developers).

3.2.2. Source Code Analysis

Our source code analysis is aimed at better understanding how the code of both the servers and clients is organized, how it evolves and how the web API is implemented. We look at technologies used for the implementation (as we have observed these have an impact on maintenance) as well as how the web API source code is organized. Specifically we look at the encapsulation of the code (e.g. is the whole source code tied to the web API or is the functionality abstracted into a translation layer), at its size[12] and at the structure of the web API itself (e.g. is the different business logic also well encapsulated).

[9] Oracle VirtualBox — `https://www.virtualbox.org/`
[10] `http://xbmc.org/`
[11] phpVirtualBox — `http://sourceforge.net/projects/phpvirtualbox/`
[12] SLOCCount — `http://www.dwheeler.com/sloccount/`

3.2.3. Co-change analysis

With our co-change analysis we identify which files consistently change with the web API-related files. Our initial goal with this analysis is to identify to what extent the web API-related code is self-contained. If we can establish that the web API-related code is (relatively) self-contained, we expect the web API to be more stable, which in turn would be helpful for the web API client developers.

Continuing our reasoning: in most cases, changes to a web API involve changing more than just the web API interface. For instance, if new methods are added, then the types used as method parameters will also have to be added elsewhere. Similarly, when a web API is changed due to a change in method parameters, these changes are often a result of deeper changes in the business logic of the software system.

These deeper changes to files which co-change with the web API interface also provide an interesting view of how a system evolves. If a non-web API file (e.g. in a different package) consistently changes whenever the web API changes, it provides a hint that such file contains web API-related functionality. This might be an indication for a refactoring opportunity.

For our analysis we make use of association rule mining to identify co-evolving entities, similar to how Zimmermann et al. have applied it previously [19]. We make use of the Apriori algorithm as implemented in the *Sequential Pattern Mining Framework*) (SPMF) tool[13]. Because this tool requires the input to be numeric, we mapped each filename to a number and considered each commit as a transaction where the files (i.e. the numbers) are the items of the transaction. In addition, because we are particularly interested in finding association rules that indicate a change to the web API, we add one item at the end of each line/transaction: 1 if the commit contains changes to the web API files, 0 if it does not. The parameters (support and confidence) used for the Apriori algorithm are explained in the analysis section.

[13]SPMF — `http://www.philippe-fournier-viger.com/spmf/`

4. Client-side Analysis

In the sub-sections below we present the results regarding the first part of our study. Namely the results of the interviews with client developers, our findings regarding web APIs evolution policies and lastly the source code impact on clients caused by web API evolution.

4.1. Interviews With Client Developers

This study aims at understanding how web API evolution impacts client developers through the forced nature of the web API changes. To do so we first performed interviews with client developers for well known web APIs (Table 1).

The most interesting findings obtained through the interviews are presented in the subsections below. These subsections represent the major themes (or codes) which the participants had experience and commented on. As additional remarks we present the results which do not fit in the predefined questions.

4.1.1. Web API Stability

We asked the client developers *"how does the effort of initial integration with a web API compare with the effort of maintaining this integration over time"* (Q1). Two of the interviewed developers (one for Google Maps and one

Q1	How does the effort of initial integration with a web API compare with the effort of maintaining this integration over time?
Q2	How often does your web API provider push changes?
Q3	How dependent is your client on the 3rd party web APIs you are currently using?
Q4	Does your project also make use of statically linked libraries and do you feel there is a difference on how its evolution compares with web APIs'?
Q5	How do you usually learn about new changes being pushed to the web API your client is making use of?
Q6	Do implementation technologies make a difference to you?
Q7	How do you learn how to use an API? (Documentation? Examples? Do errors play a role in this learning?)
Q8	Is having different versions of a web API useful when integrating with your client?
Q9	When using 3rd party APIs, did you ever find that particular thought was put into an API behavior?
Q10	As a web API client developer, given your development life cycle, how many versions should the API provider maintain? And for how long?

Table 2: Questions Asked During the Developer Interviews

for Google Calendar) were very peremptory and claimed that it takes them far more time maintaining the integration than it does integrating with a web API in the beginning.

The developer behind the integration with Redmine web API claimed that the effort involved in these two tasks is divided *"at least 50% into each task, with possibly even more time going into maintaining the integration"*.

What also came to light from all the participating client developers was the fact that in the beginning, the web APIs are very unstable and generally prone to changes.

This results in two-fold advice for web API providers and client developers alike when it comes to web API stability:

- From a provider's point of view, more thought should be put towards the early versions of the web API. In the event the web API requires some instability, then an approach as suggested by one of the interviewed client developers is recommended: the Redmine API developers clearly mark which features are prototype/alpha/beta (i.e. features which are very likely to change).
- As for web API client developers, because of this inherent instability in the early versions of web APIs, the need for separation of concerns and good architectural design becomes more urgent than ever. Integration with static libraries can be maintained for as long as the client developer wishes but since a third party is now in charge of pushing changes, making sure the changes are contained to a small set of files should become a top priority.

4.1.2. Evolution Policies

When asked about evolution policies (Q2, Q3), the participants presented us with different insights.

While different web API providers establish different timelines for deprecation of older versions of their web API, the client developer using Google Calendar's APIs was generally happy with the two year window provided by

Google and in fact favored longer periods for this evolution. This developer claimed that *"we got now two years for updating to the [new] Google Calendar API, I think it should be even longer because a year is nothing anymore"*. Of consideration is the fact that this developer works on his project as a hobby (even though he is a professional developer) and therefore favors having a longer time to migrate to newer versions of the API. As he himself states *"if you have other projects, if you have to make money on other projects, even in two years it is difficult to find time to implement [the changes]"*.

The developer interviewed in the context of the Redmine API claimed that *"[while] it is quite a difficult question which depends on many factors in the project, four months time before deprecating would be fine"*. Because the Redmine API is still under development, he would rather have shorter cycles with functionality added more often.

Despite this developer's preference for shorter cycles, the nature of the changes should also be considered. In the case of the Redmine API, the evolution process consists mostly of feature addition and the features of the web API which are likely to change are clearly marked accordingly. However, looking at the comments in the 2011 survey [10] regarding Facebook's similar four-month deprecation policy, developers complained about how *"Facebook continually alters stuff thus rapidly outdating my apps"* and *"as I only use Facebook[...], [the biggest headache] is the never ending changes to the API"*. This is an indicator that more than just the frequency of the changes, web API providers should take also into consideration how invasive are the changes being pushed.

Also interviewed were two client developers for web APIs provided by financial institutions. An important distinction in this context is the fact that the web APIs being used are not available for free, as opposed to the others under study. Perhaps for this reason and according to the interviewed developers because *"the stakes are too high in the financial context"*, the web API providers maintained all the older versions of the web API indefinitely. This allows for client developers to never have to make any changes unless they require the features made available in the new web API version. While this is the ideal sce-

nario from a web API client developer's point of view, whether this is feasible for all web API providers and the effort it takes to maintain several versions simultaneously is still something we would like to investigate in future research.

4.1.3. Static Libraries versus Web APIs

We asked all the interviewed developers how does, in their experience, the evolution of static libraries compare with the evolution of web APIs (Q4). While only one of the developers was simultaneously using static libraries as well as web APIs, his experience was that the static library he used had always maintained backwards compatible methods even after adding new features.

The developer interviewed in the context of Google Maps also mentioned that while his projects do not resort to statically linked libraries, he is using Drupal (a content management system) as the basis for his Google Maps integration and admitted that with Drupal and PHP he was in control of when to migrate to newer versions in contrast with those pushed by Google Maps. This is particularly relevant seeing as PHP itself introduced breaking changes in versions 5.3 and 5.4.

4.1.4. Communication Channels

Another issue touched upon in the interviews with the client developers has to do with how the web API providers notify their clients of upcoming changes (Q5). The client developers integrating with financial institutions' web APIs said that while it is a rare event, they will be notified by e-mail of any upcoming changes pushed by their web API providers. What was also mentioned was that while it ultimately does not affect them (because the web API providers do not force them to migrate to newer versions), it would be unfeasible to keep up with changes (should they be mandatory) from all providers due to the unreliable nature of e-mail (e.g. messages can be lost, automatically filtered as spam or simply missed altogether by the recipient).

Nonetheless, the web API providers under analysis have changed their communication channels over time. For instance, Google and Twitter nowadays force all client developers to request an API key and by doing so, they are

added to a mailing list on which the upcoming changes are announced.

While this is what is currently considered the state of the practice, client developers for these web APIs will still get e-mails even if their code is not affected by the changes.

Facebook goes further and dynamically determines what parts of the web API a specific client is using in order to send e-mails only when changes are planned for that particular functionality.

4.1.5. Implementation Technologies

Even though all the web APIs under study use JSON-based technologies, we asked the interviewed developers whether they believe that the choice of technology from the web API provider can have an impact on the effort it takes to both integrate and maintain the integration with a web API (Q6).

One of the developers integrating with financial institutions using both SOAP and REST interfaces claimed both come with advantages and disadvantages. For instance, while integrating with a SOAP interface there is generally a WSDL file available which gives an overview of which methods and types are available and how to invoke them. The downside is the extreme verbosity of such an interface which is hardly ever human-readable. On the other side, REST, while allowing for less wordy interactions lacks anything similar to the WSDL file and the client developer is left to rely solely on the documentation which is usually written manually by the web API providers (and is thus, not as reliable as an automatically generated WSDL file).

An interesting remark by the same developers was that while some web API providers claim to provide a REST interface, this is in fact not the case. In his experience the interface is simply an HTTP endpoint which outputs JSON content but which does not, for example, meet the criteria of being stateless.

The developers integrating with Google Calendar and Google Maps expressed negative opinions on XML as a language for message exchange (thus, SOAP). Specifically, the developer integrating with Google Calendar claimed that *"the simpler the [better]. I hate XML because XML is such an open stan-*

17

dard, it is very complex." whereas the developer integrating with Google Maps claimed that *"SOAP is gone and dead"*. The developer behind the integration with Google Calendar went further and commented on the effort involved in maintaining the two technologies, namely *"at the beginning [he] had a wall to climb [when switching from SOAP to JSON] but now because I have everything it is certainly much easier to switch another API from XML to JSON"*.

4.1.6. Additional Remarks

An interesting remark from the interview with the client developer for Google Maps was his concern for vendor lock-in. In fact, when dealing with web APIs, a client is *tightly coupled* with a particular web API provider. The same developer highlighted the dangers of such dependencies with the example of Google Translate which Google officially discontinued in December 2011 (even if later on the web API was made available once more).

Additionally, even though the feedback provided by the developers integrating with the financial web APIs was limited due to the providers maintaining all the old web API versions, these developers also contributed with an additional anecdotal story. During their integration with financial institutions worldwide, they are often faced with web API documentation in foreign languages. This causes great distress and requires the developers to resort to either unreliable machine translation or to eventual colleagues who happen to speak the language, both of which come with the cost of time.

4.2. Web API Characteristics

In the aforementioned survey performed in 2011, the authors claimed that in the web API world, *"there's nothing even resembling industry standards"* [10]. We also found this to be the case amongst the chosen web API providers.

In fact, each of the four web API providers under study in this paper adhere to different policies on what concerns web API evolution. These are explored in detail in the following sub sections.

4.2.1. Google Maps

The Google Maps API allows client developers to, amongst other things, display maps for specific regions, calculate directions and distances between two locations.

This API[14] falls under the global Google deprecation policy, i.e., whenever products are discontinued or backwards-incompatible changes are to be made, Google will announce this at least one year in advance. Exceptions to this rule regard whenever it is required by law to make such changes or whenever there is a security risk or *"substantial economic or material technical burden"*. To summarize, save for security-related bugs, Google claims to provide a 1-year window for the transition to a new API.

In practice, however, Google is much more lenient, e.g. analyzing the migration of Google Maps version 2 to version 3, Google provided a 3-year period for this transition rather than the announced 1-year deadline. Additionally, before the deadline arrived for version 2 going offline, Google prolonged this period for another 6 months, effectively offering a 3.5-year period for the transition. Why they offered such long period is not certain. However, anecdotal evidence from Google's user forums shows that many developers waited until the last moment to upgrade. In March 2013, an unnamed developer asked *"I'm working on upgrading to v3 but I'm expecting to finish 2 or 3 weeks after 19 May [initial deprecation date], so I was wondering if we can get an official answer about this"*. Similarly, when earlier in 2013 Google experienced an outage in all its Maps APIs' versions, several developers also asked whether v2 had already been taken offline, thus revealing that a number of developers were still using it. Google's provision of a very long transition period may have led the developers to be too relaxed about the deprecation, leading them to migrate at the latest moment.

[14] Google Maps Terms — https://developers.google.com/maps/terms

4.2.2. Twitter

The Twitter API allows client developers to manage a user's tweets as well as the timeline. While this web API[15] has no official deprecation policy, the announcement for the current API version set a 6-month period to adjust to the change. Since it implies a different endpoint URI, both versions could in fact be maintained in parallel indefinitely, and it means that once the old endpoint is disabled, all applications using it will break. Despite the 6-month period, Twitter did not follow the original plan. The new API version, announced in September 2012, was intended to fully replace the old version by March 2013. However, rather than fully take it offline, they decided to approach the problem by starting to perform "blackout tests"[16], both on the date the API was supposed to be taken offline and twice again two weeks apart after the original deadline. These blackout tests last for a period of one hour and can occur at random during the days they are announced. They act as an indicator for unsuspecting users, that they should migrate.

This approach contrasts that of Google and Facebook but gathers appreciation in its own right. The blackout tests have been very well received, with developers claiming *"These blackout tests will be super helpful in the transition. Thanks for setting those up!"*[17]

4.2.3. Facebook

The Facebook API is extensive and allows for client developers to access many data related to users' posts and connections. Facebook's[18] approach to web API evolution is substantially different as it does not use an explicit versioning system. Instead, the introduction of new features is done by an approach referred to as "migrations," which consists of small changes to the API that each developer can enable/disable at will during the roll-in period. After this period the changes become permanently enabled for all clients. The Facebook

[15]Twitter API v1.1 — https://dev.twitter.com/docs/api/1.1/overview
[16]https://dev.twitter.com/blog/planning-for-api-v1-retirement
[17]Discussion API v1's Retirement — http://bit.ly/apiv1-retirement
[18]Facebook Breaking Change Policy — `http://bit.ly/fb-changepolicy`

Developers website claims that Facebook provides a 90-day window for breaking changes. Like Google's, this policy also explicitly excludes security and privacy changes, which can come into effect at any time without notice. Unlike Google, however, Facebook has proven to not be lenient and the 90-day window is consistently enforced.

Facebook is also in the process of changing this policy. While so far there have been breaking changes put into place every month from January 2012 to May 2013 (with the exception of March 2012)[19], Facebook has announced that from April 2013, all the breaking changes will be bundled into quarterly update bulks (except security and privacy fixes).

In addition, Facebook has an automated alert system in place, which sends e-mails[20] to developers whenever the features they use are affected by a change. Dynamically determining which developers are relying on which features of the API goes along our previous line of work [20] where we investigated to which extent such a mapping affects system maintenance.

4.2.4. Netflix

The Netflix API is a public web API which allows client developers to access text catalogues of movies and tv shows available in the Netflix collection. Netflix, much like Twitter, has no official deprecation policy. Additionally, to date only two versions have been released and both versions do still work. What makes this web API stand out is the fact that all the versions released to date still work and have no planned deprecation date. This highlights that, albeit at a potential cost to the web API provider, it is possible to simultaneously maintain several versions of the same web API.

It should be noted, however, that over time Netflix has released breaking changes across all versions of its web API. For example, on June 2012 Netflix announced a new web API endpoint to which all clients had to migrate within three months. Additionally, in March 2013 Netflix announced that it *"is not*

[19] Completed changes — `http://bit.ly/fb-completedchanges`
[20] Example e-mail: `http://bit.ly/so-migrationemail`

accepting new developers into its public API program". This suggests that the public Netflix API is on a path to discontinuation.

Because no deprecation of older versions exists and the migration to the newer version is fueled by the client developers' will to access the latest features, very little can be said regarding the amount of time given to client developers to migrate. Nonetheless, the migration witnessed in both clients under study stems from the web API endpoint change for which three months were given to migrate.

4.3. Impact on Client Code

Web	Project	First Commit	Number of developers
Twitter	rsstwi2url	March 2012	1
	TwiProwl	August 2009	3
	netputweets	January 2011	6
	sixohsix/twitter	April 2008	46
Google Maps	hobobiker	November 2009	1
	cartographer	May 2008	5
	wohnungssucherportal	January 2010	1
Facebook	spring-social-facebook	August 2010	9
Netflix	pyflix2	June 2012	3
	Netflix.bundle	July 2010	8

Table 3: Project Metadata

When analyzing the impact web API evolution has on different clients, several considerations must be made regarding each project's code base. In Table 3 we present each of the projects under analysis for each web API along with its age and number of developers per their GitHub history. In Table 4 we present different metrics for the projects under analysis for each web API. All projects are different in nature and the number of lines of code (LOC) for the projects under consideration varies from 1.2KLOC to 479KLOC. This, coupled with the file count for each project, has an influence on both the average code churn for each project as well as the code churn required to implement evolution-related changes.

In our study we use code churn, a measure of how much code has been changed, as a first indicator of commits which should be further investigated manually. The raw data used for this study, namely a spreadsheet per project

including a list of commits and commits which are tagged as being related to the web API evolution task, is also available online [21].

In the following subsections we analyze the data per web API provider and present our findings.

4.3.1. Twitter

The web API evolution step under analysis for Twitter consists of a minor version upgrade. It is, nonetheless, described by Twitter as "the first major update of the API since its launch". This new version does indeed bring many changes. For instance, clients are now forced to authenticate, XML support was discontinued in favor of JSON (until then, developers were given the option for either XML or JSON) and changes have been made to rate limiting (which can penalize clients who query the web API too often).

netputweets — The netputweets project is an alternative web interface for Twitter on mobile phones. Because it implements a wide range of features from

[21] Raw commit data — http://figshare.com/articles/JSS_Web_API_Data/1192860

Web API	Project	LOC	Commits	Avg. Churn	Evol. Churn (% of Avg.)		File dispersion	Evol. commits
Twitter	rsstwi2url	2101	366	0.008203	0.008251	0.59	3	1
	TwiProwl	1199	156	0.007530	0.030629	306.75	1	1
	netputweets	8853	218	0.001679	0.005521	228.80	15	3
	sixohsix/twitter	3866	375	0.00871	0.004509	-48.21	11	7
Google Maps	hobobiker	478994	37	0.0000607	0.000147	142.27	4	2
	cartographer	1895	36	0.009328	0.115652	1139.84	17	2
	wohnungssucherportal	35119	208	0.00026	0.000127	-51.34	4	1
Facebook	spring-social-facebook	30362	1042	0.001222	0.000277	-77.33	14	1
Netflix	pyflix2	3433	49	0.008032	0.008409	4.70	8	2
	Netflix.bundle	1724	80	0.007530	0.002115	-71.91	2	1

Notes: *Avg. Churn* defines the average churn of modified files per commit (for all the commits). *Evol. Churn* defines the churn caused by the evolutionary step (manually inspected within the commits which set forward changes for the evolutionary step). *File dispersion* consists of how many files are changed in the context of the evolutionary step. *Evol. commits* consists of how many commits were involved in implementing the changes for the evolutionary step.

Table 4: Statistics Per Project

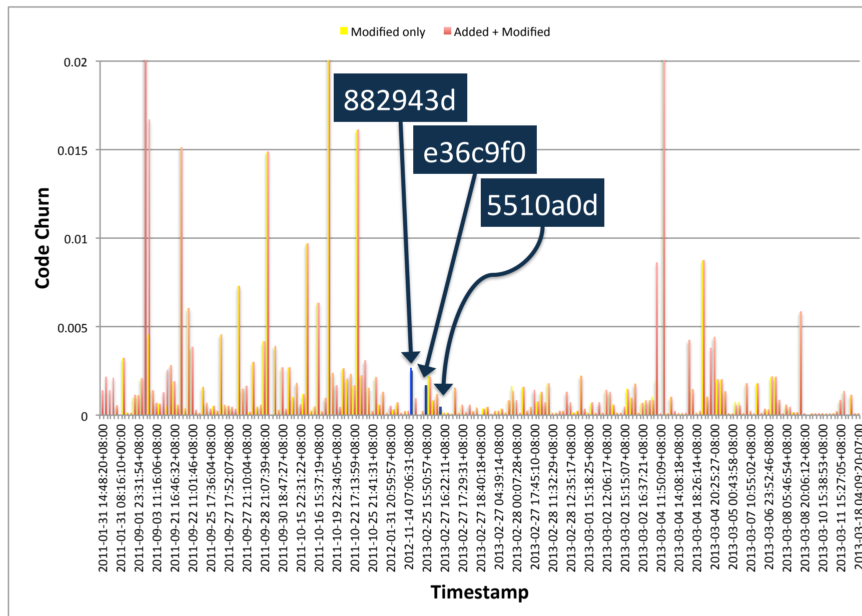

Figure 1: Code churn per commit netputweets

the Twitter web API, it is also the Twitter project with the highest LOC.

In Figure 1 we present the code churn data compiled for the netputweets project. The figure shown concerns only the netputweets project although we did compile the data for all the projects. Doing so helped us in identifying potentially interesting hotspots in the projects' commits. In this figure we highlighted the commits involved in the web API evolution task. Here it is possible to visually assert that these commits are not exceptional in terms of code churn when compared to the remaining commits. Table 4 does show, however, that the evolution-related code churn is approximately 228% higher than average and the changes span across several files.

From the same table we also learn that netputweets is the Twitter project with the largest codebase, yet, its evolution-related churn is lower than TwiProwl (the smallest Twitter project by LOC). The netputweets project contains approximately eight times more LOC than TwiProwl and it took three commits over 15 files to implement these changes. Such an increase in file dispersion may signal a tight coupling with the web API. Through manual inspection of the

netputweets project we confirmed our hypothesis. Many of the changed files contain on themselves a static reference to the Twitter endpoint which, should it change, requires these files to be also changed. Additionally, several changes are also made to the code handling the web API data. Because the data is used directly throughout the code (which implies a tight coupling with the specific data format), several changes are required throughout several files.

TwiProwl — As the client with the lowest LOC (compared to all analyzed Twitter projects), TwiProwl is also the one that implements the changes for the new API version by changing one file in a single commit. This project is a one file script which explains the file dispersion of 1. The 300% code churn compared to the average churn comes from implementing a new feature in the Twitter API (user lookup) and from adjusting several lines of code which directly iterate through the data provided by the web API (which was changed in this version).

sixohsix/twitter — Another project which has an elevated file dispersion is a Twitter library for Python (sixohsix/twitter). Manual inspection resulted in a different finding from that of netputweets. This client tucks away all the web API-specific integration into one file and even after the changes for the newest version had been implemented, the project was still using the older version. This is possible because of how the developer implemented a mechanism to allow him to choose the version of the web API by changing an argument in the method calls. This also justifies the file dispersion. While normally having to change a several number of files would be a task developers wanted to avoid, the only change to be done in this case is an argument that specifies which web API version to use.

rss_twi2url — The rss_twi2url project is a small script which provides tweets as an RSS feed. Because of its limited focus on a small subset of Twitter's web API, we expected the changes caused by web API evolution to be small. This was confirmed through the low code churn (approximately the same as the average). The changes span across three files, although manual code inspection revealed one of the files is a configuration file (changed due to Twitter's rate limit) and the other two files were directly impacted by Twitter's change on the

returned data.

What We Learn

While in a general way Twitter pushed extensive changes with its latest web API version, our findings are that when dealing with web APIs a good architecture matters more than ever. This finding is supported by two projects: sixohsix/twitter and netputweets. While both integrate extensively with the Twitter web API, netputweets, by virtue of a poorer architecture, contains larger evolution-related churn. For instance, the Python library presents an evolution related churn that is 48% smaller than the average code churn which supports a more carefully thought architecture, versus the 228% higher than average churn seen on netputweets.

4.3.2. Google Maps

The changes put forth by Google in version 3 of its Maps web API are extensive. Google says so itself in its thorough upgrading Google Maps guide: "as you start working with the new API, you will quickly find that this is not simply an incremental upgrade"[22]. In our study we noticed that simple activities such as creating an instance of the web API are now done using entirely different constructors.

hobobiker — It took the hobobiker project changes in 4 files to implement the integration with the new web API. Through manual inspection we concluded that despite the low file dispersion, all the components of this project which require Google Maps integration are tightly tied with its web API. This tight connection is observed as each component of this project which requires web API integration establishes its own direct dependency to the web API. This, coupled with the 142.27% size of the evolution commits compared to the average churn may indicate poor architectural design.

wohnungssucherportal — The wohnungssucherportal, by comparison with hobobiker, despite requiring the same number of files to be changed, it took

[22]Upgrading Your Google Maps JavaScript Application To v3 — https://developers.google.com/maps/articles/v2tov3

-51% of the average churn to implement the same changes. The LOC of both these projects is rather high and is justified by both being web applications and containing a large amount of boilerplate code (hobobiker relies on Drupal whereas wohnungssucherportal relies on Ruby on Rails). While the elevated LOC influences the absolute average churn (0.00026 compared to 0.009328 for cartographer with 1895 LOC) it should not interfere with the code churn required by the web API evolution task.

cartographer — The cartographer project stands out in the elevated file dispersion it presents (17 files changed). As a library which allows other projects to integrate with Google Maps, this project also maintains backwards compatibility with the previous version of the Google Maps web API. Because of this and because this project's architecture clearly separates the connection to the two API versions, several files were touched to provide support to the newer version. Namely, 8 files were copied and became the basis for the older version 2 support and the same files were copied and modified to enable the integration with the latest version. It is then not surprising that the code churn involved in the evolution task represents a 1139% increase over the average code churn spread out across 17 files.

What We Learn

The three projects under study present different lessons learned for client developers. While the change introduced by Google with version 3 of Maps is overarching and requires substantial changes (as stated by Google and proven by the creation of an extensive migration guide), projects like hobobiker and wohnungssucherportal suffered the sharpest pains (hobobiker in code churn and wohnungssucherportal in file dispersion). This is so as these two projects reveal poor design choices where every reference to the Google Maps web API was hardcoded.

The cartographer project on the other hand continuously implements support for both the old and new versions of Google Maps and despite the higher code churn, tucks away the web API concerns in a way that the core library does not require changes as extensive as the other projects.

4.3.3. Facebook

The only Facebook project available (as more could not be found using our approach) is a fairly large plugin for the Spring Framework which provides Facebook integration. What can be learned from this project is that even though Facebook's migrations are said to be smaller (and happen more frequently than in other web API providers), in fact the changes cause many files to require maintenance.

Considering the changes analyzed are relative to a migration and therefore not a major version change, and considering the churn percentage of this evolution task compared to the average is lower by 50%, we expected to encounter an underlying architecture with a good separation of concerns. This was confirmed through manual inspection. The web API-related code is encapsulated in Java classes specifically built for the web API communication, which were also the only files that required changes. By analyzing the changes we also realized the changes concern two major modifications in the Facebook web API. Namely, Facebook changed the way it handles images and simultaneously changed the way it refers to "check-ins" and the way to retrieve them. What also contributes to the high file dispersion of these changes is the existence of an extensive test suite. In fact, for this specific commit there are six changed files (out of the 13) which are test-related. For this particular project we conclude that despite the changes pushed by Facebook being actually intrusive and require change, the way these particular client developers designed their architecture by isolating web API access into single-feature classes mitigates this problem. The changes span across several files but are generally small and confined to the web API-specific files.

4.3.4. Netflix

The Netflix web API pushed extensive changes with version 2. Amongst them are refactorings in the returned data, changes in API conventions and addition of new features.

pyflix2 — The pyflix2 project presents a rather high file dispersion of eight files.

While the file dispersion is the first indicator of a potential poor architectural design, the meagre 4% increase in code churn versus the average suggests the changes are not very extensive. Nonetheless, manual investigation shows that part of the changes consist of unit tests and database migrations which are stored in text files. The majority of the remaining changes transform the hardcoded Python strings to Unicode, as presumably Unicode became mandatory on the new Netflix API. However, there are no mentions to this in the Netflix web API documentation.

Netflix.bundle — The second project which makes use of the Netflix API is a bundle for Plex (a media center) which contains all its web API references in the same two files. This justifies how all the evolution related changes are contained in two files as only these two particular files have the necessity to make web API calls.

What We Learn

Both projects analyzed in the context of the Netflix web API are small and relatively young. This somewhat justifies the small number of commits. The code churn caused by the web API evolution is rather small (with the projects staying around or below the code churn average). The pyflix2 project is a Python library which requires a more extensive integration than the one provided by Netflix.bundle, hence the larger file dispersion and evolution-related churn.

5. End-to-End Analysis

In this section we analyze how the web API is implemented and whether there is evidence of special design and implementation considerations on the web API source code.

5.1. VirtualBox

Our analysis of the VirtualBox case study is split into two: the server-side and client-side analysis. In the server-side we cover the implementation, versioning and source code analysis of the web API as well as a co-change analysis of the web API-related files. On the client side we strictly look at the source and how it integrates with the VirtualBox web API.

5.1.1. Server-side

Oracle VirtualBox is a software system which allows the deployment of virtualized operating systems. Through its web API it allows full control over the features of each virtual machine deployed within an instance of VirtualBox.

Implementation In the VirtualBox web API all the methods are available under one single endpoint. This results in a single “class” which contains 1888 methods (one single file with approximately 54 kLOC) from different business entities (e.g. INATNetwork, IDHCPServer, IAppliance). This is symptomatic of a web API suffering from the multi-service anti-pattern as described by Dudney et al. [21] where one single web API provides functionality pertaining to multiple disjoint business entities.

The web API is implemented using SOAP and thus, its interface is available as a WSDL file. This file is generated based off a XIDL (Extended Interface Definition Language) file which is also used to define the interface for all the APIs VirtualBox provides (webservice bindings for Java, Perl, PHP and Python as well as static library bindings for C, Microsoft COM and XPCOM interfaces) in all the different languages. The fact alone that the SOAP interface is treated as any other (non-web) API raises a warning that VirtualBox developers take no special precautions when pushing changes to the web API.

Versioning As far as versioning is concerned, VirtualBox’s web API is assigned the same version of the application itself. The implication is that regardless of whether a major, minor or patch release has pushed breaking changes to the web API, the web API will also have its version number increased. While VirtualBox does follow the semantic versioning[23] approach, with a major.minor.patch versioning scheme, in fact, it applies a twist on this approach. With semantic versioning only a major version change is allowed to introduce breaking changes to the (web) API. In VirtualBox, the documentation states that both major and minor releases may introduce breaking changes, thus departing from the semantic versioning approach. In order to clarify the reason

[23]SemVer — `http://semver.org/spec/v2.0.0.html`

why semantic versioning was not being strictly followed, we asked the developers in the official VirtualBox mailing list for clarification. One of the lead developers stated it was simply a naming convention and that they *"consider the MAJOR.MINOR.0 releases as major"*. In a different post on the mailing list, a developer also stated that *"when [they] do a major update (first and/or second digit change), [they] may change APIs in incompatible ways. This is a burden on the API consumer but it allows [them] to keep a clean design and not accumulate legacy. Within one major version (e.g. 3.1.x), [they] guarantee API stability."*

Source code analysis In order to compile an overview of the breaking changes to the VirtualBox web API we analyzed all the provided SDK versions on the VirtualBox website. Each SDK downloadable is a ZIP file containing (amongst other files) a WSDL file. The WSDL file from each SDK was extracted in order to then extract WSDL changes from subsequent version pairs of the file. While the SDK downloadables contain both a version number and a SVN revision number, the SVN revision number is from an internal SVN server which means it does not match that of Oracle's public SVN repository. This means that we have a WSDL file, its external version number (in the Major.Minor.Revision format obtained from the ZIP), the internal SVN revision but do not know what *public* revision of the whole VirtualBox source code the SDK belongs to. Because we are interested in source code changes, it is a requirement to know which commit was used to compile a specific SDK (and thus a specific WSDL). In order to obtain this information, we had to devise a matching mechanism to map WSDL files from the SDK downloadables to SVN revisions of the public repository. Because the WSDL is not versioned but rather generated, for each commit in SVN we compile the XIDL file into a WSDL file and compare to each of the WSDL files in the SDK.

By using a tool to compile the fine grained changes between two WSDL files (WSDLDiff[24]) we can then know what changes were made at the web API level

[24] `http://www.membrane-soa.org/soa-model-doc/1.4/cmd-tool/wsdldiff-tool.htm`

as well as connect these changes to other files which also had to be modified to make the web API changes possible. This tool not only mines the changes from the WSDL files, but also identifies for each change whether it breaks backwards compatibility for clients. In the scope of our study, we are only interested in changes that break backwards incompatibility as those are the only ones which cause clients to be forced to change.

With resort to this tool and through analyzing the 73 officially released versions following the Major.Minor.Patch semantics, we were able to gather that VirtualBox has pushed:

- 3 major releases, all of which contain breaking changes.
- 7 minor releases, all of which contain breaking changes.
- 63 patch releases, two of which contain breaking changes.

This data is not surprising except for the two patch releases which bear breaking changes (version 4.0.2 to 4.0.4 and 3.0.0 to 3.0.2). Developers claim that patch releases are free from backwards incompatible changes but this principle was violated twice. We tried to identify the nature of the changes introduced and the rationale behind this decision.

Our analysis has found that version 3.0.2 removed web API methods and changed a data type, whereas version 4.0.4 included a small change where a field was renamed.

As an attempt to clarify what was the reasoning behind this decision we reached out to the project's mailing list. Klaus Espenlaub, one of the VirtualBox developers, claimed that in 4.0.4 a field was renamed for the sake of clarity and in 3.0.2 there were *"more intrusive"* changes but that the removed methods were *"not really useful"* and because this functionality had been made available for a 10-day period (between version 3.0.0 and 3.0.2) they found *"the probability of breaking existing third party code was negligible"* and thus *"decided it was worth it"*.

Co-change analysis In order to identify co-changes, we applied association rule mining techniques. To do so, we first identified the file which, should it be changed, would cause a change to the web API. While Virtu-

alBox makes use of WSDL for its web services, the WSDL file is generated and not initially available under versioning. For this reason, we perform our analysis on the file which is used to generate the WSDL interface, that is "src/VBox/Main/idl/VirtualBox.xidl". We then grab all the commits where this file is changed and treat each commit as a *"transaction"* for the Apriori algorithm. By doing so, we obtain a list of which files are associated with (and thus, tend to co-evolve) which other source files.

In Table 5 we present the results for a minimum support of 10% (lowest was 24 occurrences) and a confidence of 10%. The results of this table have been filtered to include only association rules where the web API is involved. The values for minimum support and confidence were chosen by experimentation. We started with significantly higher values (80% confidence and support) and gradually lowered until the occurrences were in the range of 20.

For VirtualBox we see then that 4 out of the 5 files which contain an association rule with the web API interface file are in the "Main" package and are all core implementation files (e.g. HostImpl.cpp deals with the implementation of host-related functionality, MachineImpl.cpp contains implementation regarding the virtual machines).

5.1.2. Client-side

As a client we study phpVirtualBox, a feature-complete web client for the VirtualBox software system. This project is developed by one single developer who is not affiliated with the development of VirtualBox itself.

This client links the WSDL file statically into their own source code. For the particular situation of phpVirtualBox this static approach works well, as each

Rule	Sup.	Conf.
src/VBox/Main/MachineImpl.cpp ⇒ VirtualBox.xidl	30	14.8%
include/VBox/settings.h ⇒ VirtualBox.xidl	21	25.3%
src/VBox/Main/ConsoleImpl.cpp ⇒ VirtualBox.xidl	29	20.6%
src/VBox/Main/HostImpl.cpp ⇒ VirtualBox.xidl	14	23.3%
src/VBox/Main/include/VirtualBoxImpl.h ⇒ VirtualBox.xidl	15	15.6%
src/VBox/Main/include/HostImpl.h ⇒ VirtualBox.xidl	12	16.9%

Table 5: Association Rules VirtualBox

Rule	Support	Confidence (%)
{./interfaces/json-rpc/VideoLibrary.cpp,./interfaces/json-rpc/methods.json} ⇒ ./interfaces/json-rpc/ServiceDescription.h	24	100
{./interfaces/json-rpc/JSONServiceDescription.cpp, ./interfaces/json-rpc/methods.json} ⇒ ./interfaces/json-rpc/ServiceDescription.h	40	100
{./interfaces/json-rpc/JSONServiceDescription.cpp, ./interfaces/json-rpc/ServiceDescription.h} ⇒ ./interfaces/json-rpc/methods.json	40	88.9
{./interfaces/json-rpc/ServiceDescription.h, ./interfaces/json-rpc/JSONServiceDescription.cpp} ⇒ ./interfaces/json-rpc/methods.json	24	82.8
./interfaces/json-rpc/JSONServiceDescription.cpp ⇒ ./interfaces/json-rpc/ServiceDescription.h	24	77.4
./interfaces/json-rpc/JSONServiceDescription.cpp ⇒ ./interfaces/json-rpc/methods.json	24	77.4
./interfaces/json-rpc/methods.json ⇒ ./interfaces/json-rpc/ServiceDescription.h	91	75.8
./interfaces/json-rpc/JSONServiceDescription.cpp ⇒ ./interfaces/json-rpc/methods.json	40	66.7
./interfaces/json-rpc/JSONServiceDescription.cpp ⇒ {./interfaces/json-rpc/ServiceDescription.h,./interfaces/json-rpc/methods.json}	40	66.7
./interfaces/json-rpc/types.json ⇒ ./interfaces/json-rpc/ServiceDescription.h	63	58.3
./interfaces/json-rpc/ServiceDescription.h ⇒ ./interfaces/json-rpc/methods.json	91	51.7
{./interfaces/json-rpc/ServiceDescription.h,./interfaces/json-rpc/methods.json} ⇒ ./interfaces/json-rpc/JSONServiceDescription.cpp	40	44
./interfaces/json-rpc/ServiceDescription.h ⇒ ./interfaces/json-rpc/types.json	63	35.8
./interfaces/json-rpc/methods.json ⇒ ./interfaces/json-rpc/JSONServiceDescription.cpp	40	33.3
./interfaces/json-rpc/methods.json ⇒ {./interfaces/json-rpc/JSONServiceDescription.cpp, ./interfaces/json-rpc/ServiceDescription.h}	40	33.3
{./interfaces/json-rpc/ServiceDescription.h,./interfaces/json-rpc/methods.json} ⇒ ./interfaces/json-rpc/JSONServiceDescription.cpp	24	26.4
./interfaces/json-rpc/ServiceDescription.h ⇒ {./interfaces/json-rpc/JSONServiceDescription.cpp,./interfaces/json-rpc/methods.json}	40	22.7
./interfaces/json-rpc/methods.json ⇒ ./interfaces/json-rpc/JSONServiceDescription.cpp	24	20
./interfaces/json-rpc/methods.json ⇒ ./interfaces/json-rpc/ServiceDescription.h	24	20
./interfaces/json-rpc/ServiceDescription.h ⇒ ./interfaces/json-rpc/JSONServiceDescription.cpp	24	13.6

Table 6: Association Rules XBMC

version of phpVirtualBox targets one specific version of the web API. However, SOAP-based web APIs allow for the WSDL to be queried dynamically [13, p. 186] and in a dynamic scenario where the web API client provides support for web APIs with different versions, this approach is recommended as a way to

ensure the web API does indeed support the required version.

On the client side of VirtualBox we analyze the impact on phpVirtualBox, a popular web-based front-end application used to control VirtualBox instances through the web API.

This client maintains its versions synchronized with those of VirtualBox. The approach taken by phpVirtualBox means that with every release, it does not have to keep any code related to deprecated or modified features. Ultimately, the client developer claims that regardless of the *patch* version, an instance of phpVirtualBox from a specific minor release should work with all the patch releases under the same minor release (which is coherent with the evolution policy of VirtualBox itself).

Source code The phpVirtualBox client is written in PHP and Javascript. All the interactions with the VirtualBox web API happen in two files: one PHP file which is automatically generated based on the server's XIDL file and a second PHP file, this one manually created, which contains all the web API interaction logic. While at a first sight this is a sound design decision as it results in one single point which potentially needs to be changed should the server push breaking changes, our analysis revealed a different scenario. Indeed, a single point for change exists but all the source code directly relies on this file which is a 1-to-1 mapping of the server-side methods and data types. In other words, the client's source code is directly tied to specific methods of the server-side.

5.2. XBMC

5.2.1. Server-side

XBMC is a media player developed as open-source and has been under development since 2002. It provides a web API through a JSON-RPC implementation which allows for full control of the media player's operations.

Implementation Unlike the SOAP approach used by VirtualBox, it provides no schema similar to the WSDL. Instead, human-readable documentation is provided in the form of wiki pages on the website of the project which as of right now does provide information about the latest version of the web API.

Additionally, the source code of XBMC does have data types defined for server-side use. These types as well as their names are never provided to the web API clients. This in turn means that XBMC clients are not bound to type names or type references but rather the developers must compose JSON objects out of primitive types (strings, integers, etc).

Additionally, the 120 different methods are also packaged within 13 different classes. The encapsulation of the XBMC web API functionality is more carefully thought out than that of VirtualBox and it stands as a major difference between the two. The VirtualBox web API provides 1888 methods under one single class.

Versioning The API is implemented with its own separate versioning which, in its early stages, was represented by a single digit. Recently, at version 6 (also the current version), the semantic versioning approach was adopted. Additionally, the developers warn potential client developers that only even version numbers represent stable states of the web API. This versioning approach is similar to the one used in the Linux kernel where minor releases with an odd version number are not meant to be used in production[25].

Source code analysis On the server side of the XBMC scenario, the web API maintains its own separate version. To perform our analysis we must then first resort to the ServiceDescription.h file to identify which web API version corresponds to each snapshot of the source code.

While this analysis is possible on the server side, the Android XBMC client does not check which version of the web API is available on the server and it is therefore not trivial to identify which version is being targeted by the client.

On the client side the data sent through JSON-RPC as well as the responses are accessed through reflection. This makes it very difficult to identify which arguments of which data type are being used in each commit of the client's source code.

On the server side we attempted to use different approaches to identify which

[25] Linux Kernel Versioning — `http://www.tldp.org/FAQ/Linux-FAQ/kernel.html#linux-versioning`

methods exist in each commit. We attempted to do so by performing textual mining on the JSON file used as the definition of the web API as well as mining a C header file which also contains all the web API methods. Both approaches yielded no usable results. The JSON file is large (142 methods, 2221 LOC) and encompasses many different web API methods. It also relies on a separate file where the data types are declared (1610 LOC). This makes manual inspection of these files across the many thousands of commits infeasible. Automatic diffing of this file in pairs of consecutive commits also resulted in many false positives which would also be too time consuming to analyze manually.

Co-change analysis For the XBMC server we used a similar approach as the one used for VirtualBox. The JSON-RPC-based web API is declared as a JSON file which is then used to generate the actual interface. Therefore, changes applied to this file are a potential indicator of breaking changes. However, it is not trivial to identify what constitutes a breaking change for a JSON client. For instance, adding fields will not break backwards compatibility whereas removing fields depends on whether the client is actually using those fields.

For this reason, we considered all the commits where changes are applied on the interface definition files (methods.json and types.json) and use temporal references as a guideline to match changes on the server side to those on the XBMC Android client.

The association rules obtained from the XBMC project tell a different story from that of VirtualBox. All the rules involve three files only (VideoLibrary.cpp, JSONServiceDescription.cpp and ServiceDescription.h) in addition to the two web API interface files which are used to identify changes to the web api (methods.json and types.json). Furthermore, the files implicated as co-changing with the files used as reference for the web API are also all related to the web API, as can be seen by the package to which they belong (json-rpc). Table 6 shows us all the tight co-change relationship involving all the aforementioned files (web API interface plus the co-changed files). Nonetheless, the conclusion to be drawn is that changes to the web API are self-contained to the web API packaging.

In both the VirtualBox and XBMC case studies we also attempted to find

other possible links between the web API and the remaining source code. Specifically, we investigated whether there was any single package which was more likely to co-change with the web API. However, no such connection was found.

5.2.2. Client-side

The client application under study, the official XBMC client for Android, is developed by a subset of the XBMC developer community. This application actively keeps up with the latest version of the web API provided, thus breaking compatibility with previous versions of the server-side software. This is particularly troubling as the end-users who installs the Android XBMC client via the Google Play Store on an Android device cannot choose which version to install. This means that an end-user running an older version of the XBMC media player will have to either manually prevent the application from auto-updating or manually download the particular version through the XBMC client's previous releases website in order to have a functioning client. Looking at the comments left on the Play Store regarding the XBMC client, we found clear evidence of web API evolution causing trouble to end-users. Users of the client commented *"it worked until the last update with no warning. Apparently then it requires the latest version of XBMC. This is highly unfortunate, as it's hard to install older versions of software on Android devices. This basically means that if you want to use the Android remote you have to always use the last version of XBMC. In my opinion that's worse behavior towards your users than is usual amongst even open source, and I really hope this is not going to be repeated in the future."* and *"don't update the entire app without keeping backwards compatibility in mind.(...) I have 6 XBMCs running and have no time to update them all when a new release comes out of beta."*.

6. Discussion

In this section we use our findings to address our three research questions and present a list of nine do's and don'ts for developers of API web services.

6.1. Answering the Research Questions

We start by answering the research questions laid out in the introduction regarding the three different API providers.

6.1.1. RQ1

"What are some of the pains from client developers when evolving their clients to make use of the newest version of a web API?". Through our interviews, client developers highlighted how the early versions of web APIs are invariably unstable and change-prone. While some web API providers offer indicators of particularly unstable functionality in their web API, by default web API providers push breaking changes across the whole feature set. It also became clear that no standard policy exists on what concerns deprecation periods and that the ideal amount of time is dependent on the developer. Ideally longer periods would be provided but further study is required to establish what the cost would be for the web API provider to keep two versions of a web API active for a longer period of time. The technology being used also plays a role in the developers satisfaction with an observed preference for REST and JSON amongst the interviewed developers.

6.1.2. RQ2

"What are the commonalities in the evolution policies for web APIs?". In a survey on "Web API Evolution Pains" the authors concluded that "there's nothing even resembling industry standards, just best practices that everyone finds a way around". When it comes to evolution policies, this seems to be true as well. Google and Twitter make use of versioning and give ample periods of time (∼2 years and 6 months respectively) for the client developers to migrate. Facebook opts for not providing versioning altogether and pushes breaking changes every three months. Lastly, Netflix with already two existing versions continues to maintain both versions simultaneously. Twitter also stands out for the "blackout tests" which serve as warnings for developers that eventually the old web API version will be shutdown.

6.1.3. RQ3

"What is the impact on source code when web APIs start to evolve?". As expected, the impact on source code depends greatly on both the breadth of the changes pushed by the web API provider and on the quality of the clients' architectural design. An example of this is two projects which integrate with the Twitter web API. While sixohsix/twitter provides an extensive integration with the web API, the churn caused by the changes is much lower than that of TwiProwl which performs basic web API tasks. This same observation applies to the two Netflix projects. The code churn and file dispersion metrics have also had limited usefulness. For instance, the cartographer project contains changes in excess of 1000% of average churn and reports having 17 files changed, yet, the architectural design is robust as this project maintains support for multiple web API versions. The lesson learned is that the impact can be high (e.g. Google Maps pushed changes which affect the smallest of tasks) and that for this reason, developers should take caution and design for change. Lastly, our evidence also suggests that web APIs are significantly more change prone in their early versions.

6.1.4. RQ4

"Do web API providers take precautions in order to ease evolution pains of web APIs??". To answer this question we focus on the VirtualBox and XBMC projects since they are the only ones for which we have access to the source code.

VirtualBox In the case of VirtualBox the fact that the web API is generated from the same interface as other statically linked APIs provides an early indication that no special care is taken regarding the web API. This is further fueled by a web API which suffers from a severe case of the multi-service anti-pattern where the web API and all its 1888 methods (and respective request and response types) from different business entities are provided under one single class. Perhaps as a result of this, and to the fact that the backwards compatibility responsibility has been delegated fully to the client side, the main client for

this web API (phpVirtualBox) simply disregards the issue of backwards compatibility by releasing a new version for each of the VirtualBox releases. From the code analysis performed on VirtualBox it has also arisen that every single major and minor version has pushed breaking changes to the web API which reveals a great deal of external instability. Furthermore, the association rules mined out of the source code showed that multiple core implementation files consistently change together with the web API, revealing a potentially high fan-in between implementation and web API.

XBMC The XBMC web API was developed from the beginning as a web API. There are generally fewer breaking changes, with most breaking changes representing refactorings. Also from the association rule mining, only web API-related files (specifically, the files used for web API implementation) were observed to co-change with the main web API files. In this case study, it is then the client's fault for the poor backwards compatibility management. Namely, the Android XBMC client is publicly available in the Google Play Store and multiple end-users complain that the latest version simply does not work with older XBMC servers. The client developers do make older versions of the client available through their website but this requires manual installation and require multiple clients to be installed for controlling different servers with different versions.

In sum, referring back to RQ4, our analysis resulted in a mixed picture. On the one hand VirtualBox shows no particular arrangements made in order to maintain its web API more or less stable than any other statically linked API, whereas XBMC contains a purpose-built web API with its own versioning and human-readable documentation. As a result, the clients under analysis also deal with their web API's differently (albeit neither with an ideal approach). The phpVirtualBox client simply releases one client version per web API version and Android XBMC keeps up with the latest version of the web API, disregarding backwards compatibility.

6.2. Recommendations

Based on our investigation and additional insights obtained from observing developer forums, we compiled a list of nine recommendations for web service API providers with regard to easing the evolution task for developers of API clients.

6.2.1. Do not change too often

Facebook is pushing monthly "breaking changes", yet a recent survey on API integration pain [10] revealed that this policy has caused distress amongst developers. It is unclear whether this has played a role in Facebook moving to quarterly updates (starting April 2013). It stands to reason that some business domains require more frequent breaking changes than others, nonetheless, frequency of change would be an interesting gauge to include in future research on metrics for service quality which to the best of our knowledge currently does not exist.

6.2.2. Old versions of the API should not linger too long

Looking at the scenarios where the web API provider *will* deprecate older versions of their web API, Google started off with a 1-year timeframe for the deprecation of Google Maps's version 2, and ended up extending it to 3 years. Yet, reaching the 3-year mark, many developers still flocked to the developer forums in hopes that the deadline would be extended further (which happened for another 6 months). Seeing as maintaining legacy code often implies a high level of effort, time and cost the message is: longer periods leave developers too relaxed about the change. While a large company like Google can perhaps afford to invest in long periods for backwards compatibility, this may not be true for all other web API providers.

It should be noted that this advice is not applicable to web API providers which decide not to deprecate their old web API versions.

6.2.3. Keep usage data of your system

By knowing which users are using which features, system maintainers can target those particular users via e-mail to remind them about upcoming changes.

In previous work [22] we have investigated how this could be achieved and indeed developed Serviz which provides an overview of the system in terms of time, users and versions. This approach was also adopted by Facebook as can be seen through the targetted e-mails sent to client developers who use a particular feature of a web API which will be affected by an upcoming backwards incompatible change.

6.2.4. Blackout tests

Before taking the old versions offline permanently, try it for short periods of time. While it requires no special tooling, Twitter's *blackout tests* approach has been successful in reminding developers that a change in the API is upcoming; the approach has also been appreciated by developers.

6.2.5. Provide an example of interaction with the API

Something not gathered directly from the analysis presented in this paper but rather collected from the API integration Pain Survey, is the developers' need for an (up-to-date!) example of how to interact with the API. Maleshkova et al. [23] also recognized this need stating that *"most [web] API descriptions are characterized by under-specifications"*. Indeed, recognizing this need, industry has now starting to create tools which aid in automated web API documentation (e.g. Miredot [26]).

6.2.6. Stability Status per Web API Feature

As a web API provider, tagging each of your web API's features with a *"stability status"* which indicates whether a feature is stable for production use or instead it is alpha/beta is welcomed by the interviewed developers. This way, developers aiming for stability are able to know which features to be wary of. This recommendation can also be the starting step for further investigation on automatically determining the stability status of a feature through e.g., repository mining techniques or unstructured data mining of bug trackers.

[26] Miredot — `http://www.miredot.com/`

6.2.7. Lookout for Young Web APIs

An observation recurring from nearly all the developer interviews warns client developers about how young web APIs tend to be very change-prone. This should be taken into account by client developers who are advised to, from early on, implement good separation of concerns between web API interaction and the core of the client. Web API age is another factor which may influence web API/service quality and it should be further investigated for inclusion in potential future service quality metrics.

6.2.8. Version Accessibility

Ideally, a client will provide some degree of backwards compatibility. However, when that is not done, the client developers should make sure all previous versions of their client are as easily accessible. While not an ideal situation (e.g. an end user will require different clients to control servers with different versions as is the case for XBMC) it is still a better practice than simply discontinuing the older versions. In practice, Google for example provides a setting on the Android platform which automatically updates all applications to the latest version. Should this be used, and it is indeed important as in some cases using outdated versions represent a security risk, end users will be left with an application which is no longer compatible with an older web API. This calls for either easier switching to older versions of the same client or a better effort at not pushing backwards incompatible changes in a single e.g. Android application.

6.2.9. Robustness Against Changing Web APIs

Of particular use to client developers who integrate with web APIs is ensuring their client application is as robust as possible when web API providers push unexpected changes. In another paper [24] we have investigated mechanisms to support this and provide preliminary results of how mutation analysis can be used for this purpose.

To summarize, web service APIs drive the evolution of software. Clients are forced to update by the API providers which contrasts with the statically

linked libraries. However, in order to ease that evolution, we think the nine aforementioned guidelines should be taken into account.

6.3. Threats to validity

We now identify factors that may jeopardize the validity of our results and the actions we have taken or intend to take.

External validity. While we have quite some variety in terms of (1) developers working on web APIs, (2) API providers, as well as in (3) API client projects, it remains to be seen whether our observations still hold for (a) API providers who charge money for usage of the API, as they might be more reluctant when deprecating older version of the API which in turn might imply losing customers, and (b) for closed source API clients, whose developers might be inclined to upgrade quicker in order to satisfy their (paying) customer base with the latest security fixes and/or features. In future work, we will expand our investigation in this direction.

With regard to the generalizability of the end-to-end analysis, we studied two projects which are vastly different in terms of implementation technology, software architecture and versioning principles. They also exhibited differences in concerns with regard to backward compatibility. We acknowledge that two case studies systems form a limited basis to draw conclusions from and as such we will investigate more systems in future work.

Construct validity. We have measured the impact of evolving APIs on clients by investigating the code churn. While code churn is very valuable, it does not sufficiently take into account the relative complexity, nor the time needed to perform change tasks. In future work, through developer interviews we will investigate the actual *effort* of these maintenance tasks.

Reliability validity. There might be bias in the manual interpretation of the impact of change. To minimize bias the lead author who performed the investigation, thoroughly discussed all findings with the co-authors. Also a potential threat to reliability validity with the interviews is the existence of potential

inaccurate memories, misunderstandings, and miscommunications and misrepresentations. We attempt to mitigate this threat by recording and transcribing all the interviews.

7. Related work

Maintenance of service-based systems. Lewis and Smith were among the first to recognize that maintenance of service-based software systems is different from maintaining other types of software systems [25]. In particular, they highlight the importance of *impact analysis* for service providers as they have to consider a potentially unknown set of users.

In order to help mitigate the issues highlighted by Lewis and Smith, Espinha et al. address this lack of knowledge regarding the user-base of services by tracking how different users use a service-based system in different ways [20]. Through collecting runtime data of a web service-based system, the authors are then able to plot an overview of which services depend on which other services, as well as being able to do so for a specific period of time as well as for a specific user.

Fokaefs et al [26] also recognize the added challenges of web service evolution. The authors devised and evaluated VTracker, an algorithm for XML differencing, and based on its results analyze what actually changes between subsequent versions of web services and what is the effect on the maintainability of their clients. While very useful for XML-based services, some of the web APIs we analyzed such as Twitter or Facebook make use of JSON and have no equivalent of the WSDL document available.

Also Fokaefs and Stroulia [27] expanded on their previous work and defined a classification for the different types of changes which afflict different versions of web service interfaces through the usage of a tool (WSDarwin). This tool was also extended in further work [28] by the same authors where it is then able to automatically adapt clients to changed service interfaces.

Pautasso and Wilde study the different facets along which web services can

be described as *"loosely coupled"* and analyze different implementation technologies [11].

Maleshkova et al. study the state of the practice on what concerns web API implementation and amongst the findings, discovered that the majority of the web APIs are actually underspecified [23].

Evolution of APIs. Robillard and DeLine conducted a large-scale investigation among 440 professional developers at Microsoft to establish what makes APIs hard to learn [29]. Their observations are that the most severe obstacles developers face pertain to the documentation and other learning resources.

Dig and Johnson try to understand the nature of changes to APIs [7]. From the five case studies that they analyzed in detail, they found that over 80% of the API-breaking changes can be classified as being refactorings.

Dagenais and Robillard present *SemDiff*, tool-support for recommending API-method replacements for methods that were broken during the evolution of the API [30].

McDonnell et al. through a study on API stability and adoption in the Android ecosystem have found that, despite the added benefits of newer versions of APIs, developers tend to be slow in adopting the newer versions [31].

An interesting non-peer reviewed work in this field is a survey [10] conducted on the pains of web API integration which presents many complaints from web API client developers.

Daigneau focuses specifically on the brittleness of web APIs in his book on service design patterns [32]. He proposes the *Single Message Argument* pattern, which suggests to refrain from creating signatures with long parameter lists. Daigneau further states that long parameter lists *"[...] signal the underlying framework to impose a strict ordering of parameters which, in turn, increases client-service coupling and makes it more difficult to evolve the client and service at different rates."*

Mileva et al. [33] present an interesting analysis on the usage of different versions of statically linked APIs. As an example they show that the the 3.8.1

version of the junit library is more popular than the latest 4.4 version. They explain this usage trend through the fact that there has been a big API change between *3.x* and *4.x* and that developers were reporting "lots of work". The choice to remain with an older version of an API that developers have when working with statically linked libraries is in most cases not present when working with web APIs.

Romano and Pinzger [34], through the analysis of web service WSDL files, make use of fine-grained changes applied to service interfaces as an attempt to measure how often these changes happen and what types of changes happen between versions of services.

Kaminski et al [35], mindful of the pains of web service evolution, propose the *"chain of adapters"* technique as a means for unmanaged web service evolution where the older web service interfaces are pushed into different namespaces and a translation layer translates and forwards the calls as necessary to the older versions. This approach is also supported by an Eclipse plugin which facilitates the use of this technique on WSDL-based web services.

The work of Xing and Stroulia [36] also supports the task of web service evolution with resort to a tool (Diff-CatchUp) which through the use of heuristics, attempts to automatically suggest replacement APIs for such APIs which broke backwards compatibility.

Web APIs. Ly et al. [37] note that despite the availability of a number of best practices, e.g., REST principles, and a plethora of software components and technologies, discovering and exploiting Web APIs requires a significant amount of manual labour. Notably developers need to devote efforts to interacting with general purpose search engines, filtering a considerable number of irrelevant results, browsing some of the results obtained and eventually reading and interpreting the Web pages documenting the technical details of the APIs in order to develop custom tailored clients. In this light, Ly et al. attempt to automate the extraction of relevant technical information from web pages.

8. Conclusion

In this paper we perform an exploratory study regarding the impact of web service API evolution. Our contributions are:

- An interview with six professional developers to ask them about their experiences with web APIs that evolved.
- A study into the evolution policies of four high-profile web APIs (Google Maps, Twitter, Facebook and Netflix).
- An investigation of ten open source clients integrating the aforementioned four web APIs to see the impact of web API evolution on source code.
- A list of nine recommendations for developers of web APIs and client applications integrating web APIs.
- A study on the code impact on both server and client-side code for both VirtualBox and XBMC.

Our findings suggest that web APIs still fall short of an industry standard. Different web API providers adhere to different practices and what would seem like essential features (e.g. versioning), are in fact neglected (e.g. by Facebook).

Our study also stresses the importance of developing clients for change on what concerns web API integration. The promise of loosely coupled web service APIs comes, in fact, at the cost of having changes forced upon the client developers. Should developers fail to implement proper separation of concerns, switching to different web API providers may also prove more difficult than what *"loosely coupled"* would otherwise suggest. While some web API providers may allow developers to use their old web API versions for extended periods of time, in general, all web API providers will sooner or later impose changes on their clients.

From our two end-to-end case studies we have also found that still some web API providers consider their web API just as any other statically linked API. Changes are pushed regularly and clients will also resort to per-version client releases which in turn implies requiring different versions of the client when communicating with different versions of the web API.

From these observations stems our claim that while technology-wise web APIs seem to offer a "loosely coupled" way of connecting web API provider and client, organizationally web API provider and client are "strongly tied" when the web API starts to evolve as in most cases the client is forced to "co-evolve".

As the evolution is indeed inevitable, we also found that the different evolution policies impact the satisfaction of web API client developers. To help mitigate this problem, we provide a list of recommendations such as not changing the API too often and performing blackout tests.

Future work. We aim to extend our investigation to a wider range of API providers and a larger selection of projects using these APIs. Another aspect we want to consider in future work is to make use of a change distilling tool such as proposed by Fluri et Gall [38] and categorize the different types of changes as to be able to determine the nature of each change as well as to quantify its relative impact. Additionally, we aim to analyze whether web service API changes impact open-source and closed-source applications differently. Do these closed-source projects apply more urgency to their changes due to their paying customers? While the proposed recommendations stem from having been successful in real-world projects, an interesting aspect which we would like to further investigate is under which conditions these recommendations may in fact not be applicable or may require adjustments.

Finally, we also want to investigate whether the closed-source API providers' policies differ from those of open-source APIs where client developers have no direct say in the evolution process.

Acknowledgments

The authors would like to acknowledge NWO for sponsoring this research through the Jacquard ScaleItUp project.

References

[1] C. Burns, J. Ferreira, T. Hellmann, F. Maurer, Usable results from the field of api usability: A systematic mapping and further analysis, in: Symposium on Visual

Languages and Human-Centric Computing (VL/HCC), IEEE, 2012, pp. 179–182. doi:10.1109/VLHCC.2012.6344511.

[2] S. Raemaekers, A. van Deursen, J. Visser, Measuring software library stability through historical version analysis, in: Proceedings of the International Conference on Software Maintenance (ICSM), IEEE CS, 2012, pp. 378–387.

[3] B. Dagenais, M. P. Robillard, Recommending adaptive changes for framework evolution, in: Proceedings of the International Conference on Software Engineering (ICSE), ACM, 2008, pp. 481–490.

[4] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, S. Weerawarana, Unraveling the web services web: an introduction to SOAP, WSDL, and UDDI, Internet Computing 6 (2) (2002) 86–93.

[5] S. Vinoski, Restful web services development checklist, IEEE Internet Computing 12 (6) (2008) 96–95.

[6] M. M. Lehman, L. A. Belady, Program Evolution: Processes of Software Change, Academic Press, 1985.

[7] D. Dig, R. E. Johnson, How do APIs evolve? A story of refactoring, Journal of Software Maintenance 18 (2) (2006) 83–107.

[8] M. Laitinen, Object-oriented application frameworks: Problems and perspectives, Wiley, 1999, Ch. Framework maintenance: Vendor viewpoint, p. 9.

[9] R. Lämmel, E. Pek, J. Starek, Large-scale, AST-based API-usage analysis of open-source Java projects, in: Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), ACM, 2011, pp. 1317–1324.

[10] S. Blank (YourTrove), Api integration pain survey results, website last visited September 27, 2013 (2011).
URL https://www.yourtrove.com/blog/2011/08/11/api-integration-pain-survey-results/

[11] C. Pautasso, E. Wilde, Why is the web loosely coupled? a multi-faceted metric for service design, in: Proceedings of the International World Wide Web Conference (IW3C2), ACM, 2009, pp. 911–920.

[12] T. Espinha, A. Zaidman, H.-G. Gross, Web api growing pains: Stories from client developers and their code, in: Proc. Conference Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), IEEE CS, 2014, pp. 84–93.

[13] G. Alonso, F. Casati, H. Kuno, V. Machiraju, Web Services: Concepts, Architectures and Applications, 1st Edition, Springer Publishing Company, Incorporated, 2010.

[14] C. Pautasso, E. Wilde, REST: From Research to Practice, Springer, 2011.

[15] M. Maleshkova, C. Pedrinaci, J. Domingue, Supporting the creation of semantic RESTful service descriptions, in: Proceedings of the 3rd International SMR2 2009 Workshop on Service Matchmaking and Resource Retrieval in the Semantic Web, collocated with the 8th International Semantic Web Conference (ISWC), 2009, `http://ceur-ws.org/Vol-525/`.

[16] E. Babbie, The practice of social research, 11th edn., Wadsworth Belmont, 2007.

[17] J. W. Creswell, Research Design: Qualitative, Quantitative, and Mixed Methods Approaches, SAGE Publications, 2009.

[18] J. C. Munson, S. G. Elbaum, Code churn: A measure for estimating the impact of code change, in: Proceedings of the International Conference on Software Maintenance (ICSM), IEEE CS, 1998, pp. 24–33.

[19] T. Zimmermann, A. Zeller, P. Weissgerber, S. Diehl, Mining version histories to guide software changes, IEEE Transactions on Software Engineering 31 (6) (2005) 429–445.

[20] T. Espinha, A. Zaidman, H.-G. Gross, Understanding the interactions between users and versions in multi-tenant systems, in: Proceedings of the International Workshop on Principles of Software Evolution (IWPSE), ACM, 2013, pp. 53–62.

[21] B. Dudney, J. Krozak, K. Wittkopf, S. Asbury, D. Osborne, J2EE Antipatterns, 1st Edition, John Wiley & Sons, Inc., New York, NY, USA, 2002.

[22] T. Espinha, A. Zaidman, H.-G. Gross, Understanding the runtime topology of service-oriented systems, in: Proc. of the Working Conf. on Reverse Engineering (WCRE), IEEE CS, 2012, pp. 187–196.

[23] M. Maleshkova, C. Pedrinaci, J. Domingue, Investigating web APIs on the world wide web, in: Proc. European Conf. on Web Services (ECOWS), IEEE CS, 2010, pp. 107–114.

[24] T. Espinha, A. Zaidman, H.-G. Gross, Web api fragility: How robust is your web api client, Tech. Rep. TUD-SERG-2014-009, Delft University of Technology (2014).
URL `http://arxiv.org/abs/1407.4266`

[25] G. Lewis, D. Smith, Service-oriented architecture and its implications for software maintenance and evolution, in: Proceedings Frontiers of Software Maintenance, IEEE CS, 2008, pp. 1–10.

[26] M. Fokaefs, R. Mikhaiel, N. Tsantalis, E. Stroulia, A. Lau, An empirical study on web service evolution, in: Web Services (ICWS), 2011 IEEE International Conference on, 2011, pp. 49–56.

[27] M. Fokaefs, E. Stroulia, Wsdarwin: Studying the evolution of web service systems, in: A. Bouguettaya, Q. Z. Sheng, F. Daniel (Eds.), Advanced Web Services, Springer New York, 2014, pp. 199–223.

[28] M. Fokaefs, E. Stroulia, Wsdarwin: Automatic web service client adaptation, in: Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research, CASCON '12, IBM Corp., Riverton, NJ, USA, 2012, pp. 176–191.

[29] M. P. Robillard, R. DeLine, A field study of API learning obstacles, Empirical Software Engineering 16 (6) (2011) 703–732.

[30] B. Dagenais, M. P. Robillard, Semdiff: Analysis and recommendation support for API evolution, in: Proceedings of the International Conference on Software Engineering (ICSE), IEEE, 2009, pp. 599–602.

[31] T. McDonnell, B. Ray, M. Kim, An empirical study of API stability and adoption in the Android ecosystem, in: Proceedings of the International Conference on Software Maintenance (ICSM), IEEE CS, 2013, pp. 70–79.

[32] R. Daigneau, Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services, Addison-Wesley, 2011.

[33] Y. M. Mileva, V. Dallmeier, M. Burger, A. Zeller, 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, ACM, 2009, pp. 57–62.

[34] D. Romano, M. Pinzger, Analyzing the evolution of web services using fine-grained changes, in: Proceedings of the 2012 IEEE 19th International Conference on Web Services, ICWS '12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 392–399.

[35] P. Kaminski, M. Litoiu, H. Müller, A design technique for evolving web services, in: Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research, CASCON '06, IBM Corp., 2006.

[36] Z. Xing, E. Stroulia, Api-evolution support with diff-catchup, Software Engineering, IEEE Transactions on 33 (12) (2007) 818–836.

[37] P. Ly, C. Pedrinaci, J. Domingue, Automated information extraction from web APIs documentation, in: X. Wang, I. Cruz, A. Delis, G. Huang (Eds.), Web Information Systems Engineering (WISE), Vol. 7651 of LNCS, Springer Berlin Heidelberg, 2012, pp. 497–511. doi:10.1007/978-3-642-35063-4_36.

[38] B. Fluri, H. C. Gall, Classifying change types for qualifying change couplings, in: Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on, 2006, pp. 35–45.

TUD-SERG-2014-017
ISSN 1872-5392