Web API Growing Pains: Stories from Client Developers and Their Code

Tiago Espinha, Andy Zaidman, Hans-Gerhard Gross
Delft University of Technology, The Netherlands
{t.a.espinha, a.e.zaidman, h.g.gross}@tudelft.nl

***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 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. Our study is complemented with a set of observations regarding best practices for web API evolution.**

I. Introduction

Modern-day software development is inseparable from the use of Application Programming Interfaces (APIs) [1]. 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 [2]. 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 [3], [4]. However, what happens when these web APIs start to evolve? Lehman and Belady emphasize the importance of evolution for software to stay successful [5], and updating software to the latest version of its components, accessed through APIs [6]. In the context of statically linked APIs, Dig and Johnson state that *breaking changes* to interfaces can be numerous [6], and Laitinen says that, unless there is a high return-on-investment, developers will not migrate to a newer version [7].

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 [8]. 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" [9] 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"* [10] on web services. Indeed, all the web APIs make use of REST interfaces which 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 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 of web API clients to find out **[RQ3]** what the impact on source code is when web APIs start to evolve.

The remainder of this paper is structured as follows: in Section II we describe our experimental setup including how the projects were selected and how we calculate the impact on code, Section III describes the interviews with client developers and the lessons learned across different domains, Section IV looks at the different web API policies from different providers, Section V presents the impact web API evolution has on code and Section VI frames these results with our research questions and provides a list of recommendations for API providers. Lastly, we discuss related work in Section VII and present our conclusions in Section VIII.

II. Experimental Setup

Our exploratory study is composed of three parts. We started by interviewing six developers (Table II) who maintain clients for web APIs as to obtain anecdotal evidence of developers

978-1-4799-3752-3/14 © 2014 IEEE

CSMR-WCRE 2014, Antwerp, Belgium

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

TABLE I
STATISTICS PER PROJECT

who had to undergo web API evolution in their clients. In the second part of our study we analyze common evolution policies (i.e. deprecation periods, breaking change notifications, etc) from four major web API providers. This allows us to identify potential best practices. Lastly 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.

A. 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 V) and sent out public calls for participation on social networks.

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 II provides an overview of the web APIs each of the 6 interviewees developed clients for.

The interviews took on average thirty minutes per developer and were either performed face-to-face or via Skype in the format of a semi-structured interview [11]. The ten starting questions that we used during the interview are listed in Table III and cover several web API-related issues, such as: maintenance effort, frequency of version upgrades, security, developer communication and implementation technologies.

B. 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 [1] 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.

Google Maps	1 developer
Google Search & Bing	1 developer
Redmine API	1 developer
Google Calendar	1 developer
Unnamed Payments Aggregator	2 developers

TABLE II
INTERVIEWED DEVELOPERS

C. 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 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[2] 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[3] as having been changed.

D. 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

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

[2]Web API Design — http://bit.ly/apigee-web-api-design

[3]Completed Changes — http://bit.ly/fb-completedchanges

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 [12], which we define for each file as

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

The code churn we analyze and display in Table I (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 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.

III. 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 II).

The most interesting findings obtained through the interviews are presented in the subsections below.

A. 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"*. Two of the interviewed developers (one for Google Maps and one 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.

B. Evolution Policies

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. 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.

The developer interviewed in the context of the Redmine API claimed his team is usually given four months and while he was generally happy with this pace, 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

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 III
Questions Asked During the Developer Interviews

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 [9] 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 scenario 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.

C. 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. 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.

D. 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. 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.

E. 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.

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 a very strong distaste for SOAP, claiming it is unnecessarily complex. The developer behind the integration with Google Calendar said that while it took some effort when Google Calendar switched from SOAP (XML) to REST (JSON), after the initial effort was complete the changes later on became much easier.

F. 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.

IV. 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"* [9]. 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.

A. 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[4] 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.

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

B. Twitter

The Twitter API allows client developers to manage a user's tweets as well as the timeline. While this web API[5] 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"[6], 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!"*[7]

C. Facebook

The Facebook API is extensive and allows for client developers to access many data related to users' posts and connections. Facebook's[8] 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 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)[9], 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[10] to developers whenever the

[5]Twitter API v1.1 — https://dev.twitter.com/docs/api/1.1/overview
[6]https://dev.twitter.com/blog/planning-for-api-v1-retirement
[7]Discussion API v1's Retirement — http://bit.ly/apiv1-retirement
[8]Facebook Breaking Change Policy — http://bit.ly/fb-changepolicy
[9]Completed changes — http://bit.ly/fb-completedchanges
[10]Example e-mail: http://bit.ly/so-migrationemail

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 [13] where we investigated to which extent such a mapping affects system maintenance.

D. 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 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.

V. Impact on Client Code

When analyzing the impact web API evolution has on different clients, several considerations must be made regarding each project's code base. In Table I we present 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.

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

A. 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 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 I 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

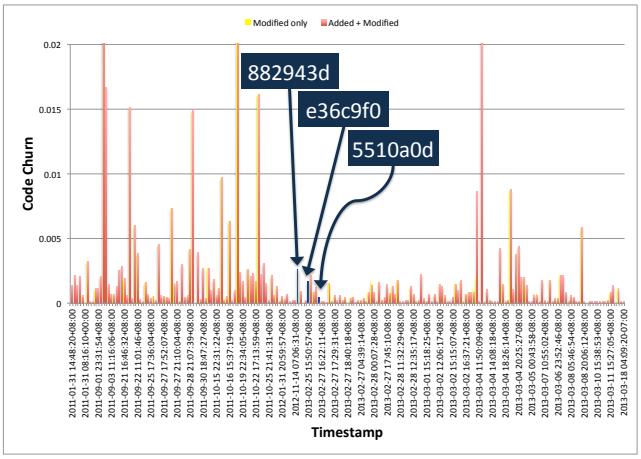

Fig. 1. Code churn per commit netputweets

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.

B. 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"[11]. 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 -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.

C. 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.

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

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.

D. 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.

VI. Discussion

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

A. Answering the Research Questions

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

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 provide 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.

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 ($\sim$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.

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.

B. Recommendations

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

1) Do not change too often: Facebook is pushing monthly "breaking changes", yet a recent survey on API integration pain [9] 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).

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). The message is: longer periods leave developers too relaxed about the change.

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

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. This approach has been studied in previous work [14] and it was also adopted by Facebook.

4) Blackout tests: Before taking the old versions offline permanently, try it for short periods of time. 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.

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. [15] also recognized this need stating that *"most [web] API descriptions are characterized by under-specifications"*.

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.

7) Bonus — Client Developers: 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.

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 seven aforementioned guidelines should be taken into account.

C. 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.

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.

VII. 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 [16]. In particular, they highlight the importance of *impact analysis* for service providers as they have to consider a potentially unknown set of users.

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 [13].

Pautasso and Wilde study the different facets along which web services can be described as *"loosely coupled"* and analyze different implementation technologies [10].

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 [15].

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 [17]. 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 [6]. 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 [18].

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 [19].

An interesting non-peer reviewed work in this field is a survey [9] 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 [20]. 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."*

VIII. 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 seven recommendations for developers of web APIs and client applications integrating web APIs.

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.

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. 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?

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] S. Raemaekers, A. van Deursen, and J. Visser, "Measuring software library stability through historical version analysis," in *Proc. Int'l Conf. on Software Maintenance (ICSM)*. IEEE CS, 2012, pp. 378–387.

[2] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," in *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 2008, pp. 481–490.

[3] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, "Unraveling the web services web: an introduction to SOAP, WSDL, and UDDI," *Internet Computing*, vol. 6, no. 2, pp. 86–93, 2002.

[4] S. Vinoski, "Restful web services development checklist," *IEEE Internet Computing*, vol. 12, no. 6, pp. 96–95, 2008.

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

[6] D. Dig and R. E. Johnson, "How do APIs evolve? A story of refactoring," *Journal of Software Maintenance*, vol. 18, no. 2, pp. 83–107, 2006.

[7] M. Laitinen, "Object-oriented application frameworks: Problems and perspectives," M. Fayad, D. Schmidt, and R. Johnson, Eds. Wiley, 1999, ch. Framework maintenance: Vendor viewpoint, p. 9.

[8] R. Lämmel, E. Pek, and 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.

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

[10] C. Pautasso and E. Wilde, "Why is the web loosely coupled? a multi-faceted metric for service design," in *Proc. Int'l World Wide Web Conf. (IW3C2)*. ACM, 2009, pp. 911–920.

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

[12] J. C. Munson and 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.

[13] T. Espinha, A. Zaidman, and H.-G. Gross, "Understanding the interactions between users and versions in multi-tenant systems," in *Int'l Workshop on Principles of Software Evolution (IWPSE)*. ACM, 2013, pp. 53–62.

[14] ——, "Understanding the runtime topology of service-oriented systems," in *Proc. of the Working Conf. on Reverse Engineering (WCRE)*. IEEE CS, 2012, pp. 187–196.

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

[16] G. Lewis and 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.

[17] M. P. Robillard and R. DeLine, "A field study of API learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011.

[18] B. Dagenais and M. P. Robillard, "Semdiff: Analysis and recommendation support for API evolution," in *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE, 2009, pp. 599–602.

[19] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *Proc. Int'l Conf. on Software Maintenance (ICSM)*. IEEE CS, 2013, pp. 70–79.

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