Web APIs, Like Caterpillars, Are Temporary

Benjamin Cassell and Tyler Szepesi
University of Waterloo
{becassel, stszepes}@uwaterloo.ca

Abstract

Modern web services offer application programming interfaces (APIs) which developers can use to build applications while easily leveraging services that would be difficult or impossible for them to provide on their own. Often these web services give developers access to data or functions which are uniquely provided by the organization that runs the web service API. While recent work has been presented on how to design good locally-installed and accessed (traditional) APIs, we have found that the generally accepted maxims of design for traditional APIs are not a good fit for web APIs. In this paper, we characterize and investigate the design decisions made by the Twitter API, and contrast them against Joshua Bloch's commonly accepted maxims of API design. Based on our findings, we suggest a modified set of maxims for designers of web APIs, which better captures the nature of web services.

1. INTRODUCTION

Traditional APIs have long been an important tool for programmers. Proper utilization of APIs can save programmers time and resources, particularly when the services offered by the API would be complex or time consuming for the programmer to implement on their own (which is typically the case when the functionality of the API is not the focus of the programming project but rather just a step on the path to presenting something interesting). Web service APIs have continued this trend, offering not only access to time saving resources, but also functions and data that would otherwise be unavailable to programmers (such as the social networks of Twitter, Facebook and Google). Unfortunately, programmers are also a notoriously fickle crowd: Attempting to upgrade a traditional API in a manner that breaks compatibility with existing programs, even if for valid design reasons (like correcting a mistake made early in the life-cycle of the API), will probably result in many programmers abandoning newer versions of the API until absolutely forced to upgrade.

Examples of poor design decisions that could not be removed for compatibility reasons are present in many different traditional APIs. One particularly pertinent example is the non-generic collections library in Microsoft's .NET framework. In early versions of the .NET framework, object collections (such as lists and hash tables) were only available as "non-generics" which accepted and returned raw object types. .NET 2.0 saw the introduction of templated "generic" collection types (allowing for collections of any data type). It is well-accepted that generic collections out-perform their non-generic equivalents: They have better type safety, more functionality and better performance than non-generics, effectively rendering non-generic collections obsolete [5]. Despite this, nine years and a full two and a half major releases of .NET later, the non-generic collections library is still present in the framework. Microsoft has decided, in the interests of compatibility, to not deprecate and remove non-generic collections (which would prevent code in many legacy programs from being able to work under newer versions of the .NET framework), and thus the non-generic collections library lives on, despite the introduction of better alternatives into the API.

The reason that it is so difficult to change APIs once they have been made publicly available is that programmers (and the organizations that employ them) dislike being forced to make compatibility updates, and it is easier from a maintenance perspective to keep a version of an API that the programmer is convinced works rather than upgrade and devote resources to making code work with the newest version of the API. Many programmers will opt to simply not adopt newer versions of APIs instead of being forced to make compatibility upgrades. This means that exposed functionality in an API (available API calls, return types, parameter counts, types and orderings) are all very difficult to change in traditional APIs, so much so in fact that Joshua Bloch, while acting as Chief Java Architect at Google, has stated that, "Public APIs, like diamonds, are forever" [1].

We believe that the plodding, tectonic nature of traditional APIs contrasts heavily with the nature of modern web service APIs, which are becoming an increasingly integral part of many thousands of applications. The ProgrammableWeb database, which tracks web service APIs, began in 2005 with a mere thirty-two APIs listed. As of late 2011, ProgrammableWeb reported that they were listing 4000 Web APIs and that this number was increasing at an incredible (and increasing) rate [11].

In this paper we examine whether or not there are substantial differences between traditional APIs and modern web service APIs. Joshua Bloch has provided, in addition

to his comments on how APIs should evolve (or possibly *not* evolve), a list of good design decisions that traditional APIs should make in order to be successful [1]. We will examine the Twitter API, a large, heavily-used and very popular web service API, within the context of these properties to determine how well it is designed, in addition to any decisions that clearly mark a departure from conventional API wisdom resulting from the nature of the Twitter API being a web-based service.

The contributions of this paper are as follows:

- A characterization of the Twitter API, a very large and highly popular example of a modern web service API, based on comprehensive information aggregated from Twitter's API resources.
- An examination of the differences between two versions of the Twitter API (the original version 1, and the current version 1.1).
- An exploration of whether or not the Twitter API adheres to methodologies that are typical of well-designed traditional APIs, in addition to an exploration of why the Twitter API deviates, or why it is able to deviate, from such design principles in several areas.
- Evidence that modern web service APIs may deviate from the best-practices of traditional APIs due to their inherently different natures, and a new take on API design principles specifically for web APIs that corresponds to the properties unique to web service APIs.

The remainder of this paper is organized as follows: Section 2 provides a brief summary of background information and related work. Section 3 discusses how we gather our data, and gives details about the type of information we are interested in collecting. Section 4 discusses information and characteristics about the Twitter API that we discover through our data gathering process. Section 5 compares decisions made in the design of the Twitter API against good design principles for traditional APIs. Section 6 discusses potential future applications for the information that we have uncovered, and Section 7 concludes the paper.

2. BACKGROUND AND RELATED WORK

The topic of API design, and particularly the usability of APIs, has received a great deal of attention in recent years. Much of the research focuses on designing empirical studies to determine how usable an API is for developers building applications. One of the first authors to explore this space was Steven Clarke, who pioneered the concept of using the cognitive dimensions framework as a tool for studying APIs [2].

As the cognitive dimensions framework was one of the first methods that showed signs of success for studying API usability, it has since been used in a number of studies. A recent example of such a study was performed by Piccioni, Furia and Meyer. In this study, the authors explored a combination of interviews and systematic observations in order to provide a rigorous empirical study of API usability [6]. Additional work done by Farooq, Welicki and Zirkler suggests that peer reviews, in combination with usability tests (such as those presented by Clarke), are a very effective way to discover, and thus help eliminate, API usability issues [3].

Additional studies, such as the one performed by Grill, Polacek and Tscheligi in 2012, have been presented by researchers seeking to apply methods traditionally used within the context of HCI to the realm of studying API usability [4].

Good design decisions are critical to the API-building process. One of the key inspirational works on this topic in recent years (which also serves as the basis for our analysis of the Twitter API) is a set of API design maxims provided by Joshua Bloch at the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) conference in 2006 [1]. In his keynote address, Bloch presents 38 maxims that he derived from his time designing the Java Collections Framework. Of the 38 design principles, we have found that 16 apply to a web API framework such as the Twitter API. In this paper, we use these 16 principles, along with the case study of the Twitter API, as a starting point to define a new set of maxims specifically tailored for web APIs.

3. DATA GATHERING

Resources for the Twitter API are available online, in HTML format, for both the current version of the API (version 1.1) [10] as well as the deprecated legacy version (version 1) [9]. The web pages for both versions of the API contain a full list of the web service calls available in the versions, as well as associated links to web pages which describe each API call in detail. Because the volume of information for each version is actually quite large, it was necessary to create an application to gather and summarize the data on the API. Using a custom-built web page-scraping application written in C#, we automatically stepped through both lists of API calls, using simple HTML parsing and regular expressions to extract the links to the description pages of each web service resource. We then, in turn, scrape the pages of the individual API calls for the following information which we can use to characterize the Twitter API as a whole. Some of the most relevant information that we gather includes:

- **Name**: The name of the API call.
- **Version**: The version (or versions) that the API call belongs to, allowing us to classify when API calls were introduced or removed.
- **Resource URL**: The URL used to access the resource.
- **Last Update**: The date of the last update of the resource.
- **Rate Limited**: Whether or not the resource is rate limited.
- **Authentication Requirements**: The types of authentication requirements Twitter places on its resources, allowing use to determine if restrictions on resources has changed between versions.
- **Response Formats**: The types of responses available from the different resources, allowing us to determine whether Twitter has changed return types at a standards level.
- **HTTP Methods**: The HTTP methods associated with a resource. This allows us to determine whether

or not access types to different resources have changed between versions.

- **Parameter List**: The number of parameters accepted by any given resource request, in addition to their details (in particular, but not limited to, names and whether or not resources are optional).

A sample subset of some of the data we scrape can be seen in Figure 1. The data being demonstrated in the figure corresponds with the API call used to post messages (Tweets) to a Twitter account in version 1.1 of the API.

```
name: POST statuses/update,
version: v1.1,
resource_url: https://api.twitter.com/1.1/
    statuses/update.json,
last_update: Tue, 2012-11-20 08:24,
rate_limited: No,
authentication: Requires user context,
response_formats: json,
http_methods: POST,
params_count: 7,
params: (name: status, required: required)
(name: in_reply_to_status_id, required:
    optional)
(name: lat, required: optional)
(name: long, required: optional)
(name: place_id, required : optional)
(name: display_coordinates, required:
    optional)
(name: trim_user, required: optional)
```

Figure 1: Scraped data for Tweeting in Twitter v1.1.

In the interests of performance, reliability and repeatability when parsing the information that we require (as well as to mitigate the risk of being IP blacklisted by Twitter for rapidly scraping their publicly-available resource pages many times and placing unnecessary load on their servers with suspicious access patterns), we also compiled and produced a local snapshot of the Twitter API documentation. This snapshot, dated March 11th, 2014 was archived and used for any further parsing in place of retrieving live versions of the API documentation. It, along with our parser and all of our aggregate data, is available upon request.

4. DATA ANALYSIS

Our analysis of the two versions of the Twitter API has revealed substantial updates to the API between versions. Table 1 shows the number of publicly accessible API calls that were available during each version's lifespan. Rows *v1* and *v1.1* show API calls available to each version that differ in some way from equivalent API calls in the other version of the API. Row *v1 and v1.1* shows API calls that are present in both versions of the API and stayed wholly unmodified during the transition between versions. The number of calls present in version 1.1 of the API that share a name with calls from version 1 of the API but that were updated or changed in some way is shown in row *Updated in v1.1*.

32 entirely new API calls were introduced to the Twitter API in version 1.1. Interestingly, when moving from version 1 to version 1.1 of the API (and contrary to the notion that "APIs are forever"), 59 API calls were not considered important enough to make the cut and were removed from the API. On the other hand, 68 API calls were brought forwards to the next version of the API in an updated or modified form (retaining at least their names).

Version	API Calls
v1	127
v1.1	100
v1 and v1.1	4
New to v1.1	32
Updated in v1.1	68
Removed after v1	59
Total v1	131
Total v1.1	104
Total (No Updates)	163
Total (Updates)	231

Table 1: Total number of API calls present across different versions of the API.

Vers.	Required	Optional	Not Required	Null
v1	81 (64%)	21 (16%)	24 (19%)	1 (1%)
v1.1	100 (100%)	0 (0%)	0 (0%)	0 (0%)

Table 2: Breakdown of authentication requirements (API calls with requirements and percentage of total API calls). Excludes the authorization and authentication API calls themselves. Calls that were missing documentation are classified under *Null*.

Out of a grand total of 163 unique (by name) API calls, only four (roughly 2.5%) were entirely unmodified between versions 1 and 1.1 of the Twitter API. All of these unmodified API calls are authentication and security-related (`GET oauth/authenticate`, `GET ouath/authorize`, `POST oauth/access_token`, `POST oauth/request_token`). Twitter has supported authentication and authorization through the open source OAuth 1.0 standard since version 1, but additionally supported basic HTTP authentication in version 1 of the API. Version 1.1 of the API, however, removed support for basic authentication, and now provides authentication and authorization exclusively through OAuth 1.0.

Additionally, whereas a significant number of API calls did not require authentication and authorization in version 1 of the Twitter API, all calls (with the exception of the authentication and authorization calls themselves, for obvious reasons) now require either application or user-level authentication in version 1.1 of the API [7]. The breakdown of authentication requirements for the Twitter API is shown in Table 2.

Along with changes to the authentication requirements of the API, there have also been changes to the way Twitter wants clients to access resources. In particular, although many resource URLs in version 1 of the Twitter API used HTTP addresses, all resource URLs in version 1.1 of the API use HTTPS and thus provide built-in security over TLS. In addition to providing peace of mind for clients, the security and authentication changes in version 1.1 of the API (in particular, the move to strict OAuth authentication) allow Twitter to provide much tighter control over access to resources. They also allow Twitter to collect more in-depth tracking metrics and more effectively rate-limit users and

Vers.	GET	POST	DELETE	HEAD	PUT
v1	87 (66%)	48 (37%)	8 (6%)	1 (1%)	1 (1%)
v1.1	68 (65%)	44 (42%)	1 (1%)	0 (0%)	0 (0%)

Table 3: Breakdown of HTTP methods (API calls using methods and percentage of total API calls).

Vers.	JSON	XML	RSS	Atom
v1	127 (100%)	110 (87%)	8 (6%)	8 (6%)
v1.1	100 (100%)	0 (0%)	0 (0%)	0 (0%)

Table 4: Breakdown of response formats (API calls using formats and percentage of total API calls). Excludes the four authentication calls from both versions (which do not use a response format).

applications that are abusing the API [8].

Beyond making the switch to HTTPS for security reasons, Twitter has also made other changes to their services at the application layer: Whereas multiple API calls in version 1 of the API made use of HTTP `PUT` and HTTP `HEAD` requests, all API calls in version 1.1 of the API rely exclusively on HTTP `GET`, HTTP `POST` and HTTP `DELETE` requests (which are also used by version 1 of the API). The breakdown of HTTP method usage by API calls can be seen in Table 3. Note that the displayed percentages do not sum to 100%, as some API calls support more than one HTTP method.

Furthermore, Twitter has, in new versions of the API, limited the formats in which it provides access to its resources. As can be seen in Table 4, although almost every API call in version 1 of the Twitter API supported returning responses in both JSON and XML formats (with a few even supporting responses in RSS and Atom publishing formats), Twitter API version 1.1 only supports returning results in JSON. This is in line with Twitter's publicly announced position on formats, having stated that they would discontinue RSS, Atom and XML support due to their infrequent modern usage [8].

In terms of parameters that are accepted by API calls, we see several interesting trends from the collected data, which are visualized in Table 5 (which details the total and average numbers of both required parameters and parameters in general accepted by API calls), as well as Figure 2 and Figure 3, which provide breakdowns of how many API calls accept varying numbers of parameters and required parameters respectively.

In both versions of the API, the average number of parameters accepted by API calls is larger than three, and approaches very close to four for version 1.1 of the API. In addition to generally accepting more parameters per API call than version 1, version 1.1 of the API also has more required parameters per API call on average. For both versions, the average number of required parameters per API call is less than 1, and when examining the breakdown in Figure 3, one can clearly see that the vast majority of API calls do not take any required parameters at all. This number is slightly misleading however: Although the API documentation marks some parameters as "optional" (or "semi-optional" in some cases, which we have simply merged into "optional"), there are several instances of API calls which require that one of multiple optional parameters are provided, essentially rendering these optional parameters required as well. The number of required parameters we show here thus under-represents the "true" number of required parameters across all calls in the API.

Vers.	Total	Average	Req. Total	Req. Average
v1	467	3.56	88	0.67
v1.1	397	3.82	78	0.75

Table 5: Breakdown of total and average number of parameters and required parameters accepted by API calls.

Most API calls accept a small number of parameters (with two parameters being the most popular number to accept between both versions of the API), and the general trend is that after two parameters, the number of API calls which accept a given number of parameters decreases as the number of parameters increases. There are however, two interesting exceptions: Significantly more API calls accept four parameters than three, and likewise, significantly more API calls accept six parameters than five. Version 1 of the API holds the dubious honour of providing the API call with the most accepted parameters, 13. No API call in version 1.1 of the API accepts more than 11 parameters.

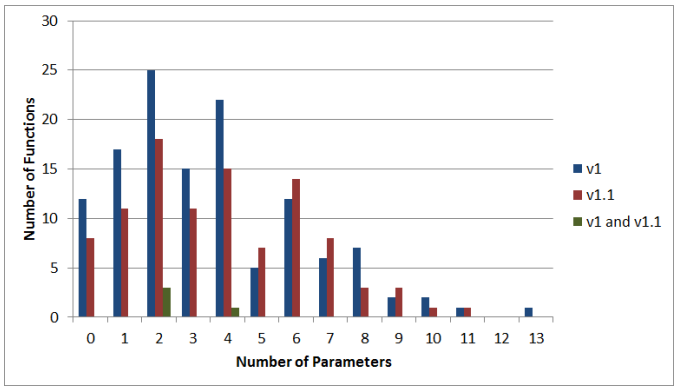

Figure 2: Number of parameters accepted by API calls.

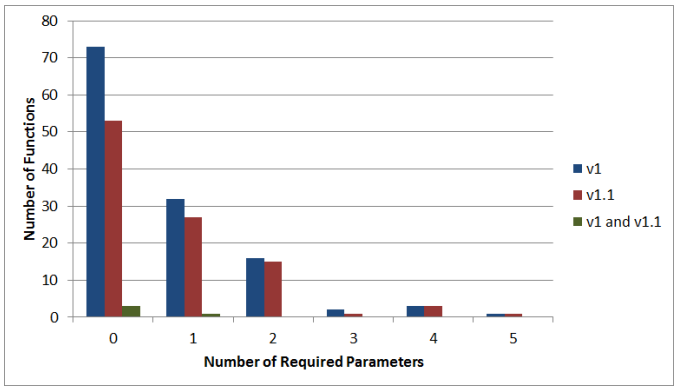

Figure 3: Number of required parameters accepted by API calls.

5. API MAXIMS

In this section, for each of the 16 applicable maxims of API design, we ask the following question: Does the Twitter API adhere to the maxim? In addition to answering this, we also discuss whether or not web APIs in general should be following each maxim.

APIs can be among your greatest assets or liabilities. The core concept of this maxim is that good APIs attract users and bad APIs discourage users from adopting it. If it is possible to quickly and easily build an application against the API, then it is more likely that users will actually use the API. Alternatively, an API that is hard to understand does little to ease high development costs, and users of the API will be discouraged and seek alternatives.

When putting this maxim in the context of the Twitter API, we need to consider the fact that there is only one Twitter, so an unhappy user has no alternative. From the perspective of Twitter being the sole provider of a service, this maxim does not apply. However, Twitter is not a required service, so unhappy users of the Twitter API could simply choose to not build applications that use Twitter. From the perspective of Twitter as a business trying to attract customers, the API is an asset or liability.

Unlike Twitter, other web APIs provide access to a service that is available through alternative APIs. A prime example of a service that is available through a web API by many different providers is maps. Google Maps, MapQuest, Microsoft Bing, OpenLayers, and more all provide their own web API to a map service. In the presence of competition, the API is certainly an asset or liability. We therefore conclude that this maxim does apply to web APIs.

Public APIs, like diamonds, are forever. Once an API is made available to the public, users will build applications against it. As more applications are built, the API becomes very difficult to change. For traditional libraries, where the user must explicitly update their copy to a newer version, changes to the API may cause users to refuse to upgrade. This is problematic, as updates to the API may accompany other important changes, such as fixes for security flaws.

From Section 4 we see that Twitter has very clearly ignored this maxim. In version 1.1, only four API calls remained identically the same, and 68 of the API calls were changed. Not only did these API calls change, but the changes were primarily in the form of restrictions that break compatibility. For example, version 1.1 restricts all return values to be formatted in JSON, and all API calls must use authentication. A significant number of API calls were even outright removed.

Changes of the magnitude demonstrated by Twitter are unprecedented in traditional APIs, and represent a major shift in paradigm. In a traditional API, the user can simply refuse to update their local copy of the library, and continue to use the legacy API. In contrast, a change made to a web API can be forced upon the user by simply refusing to service deprecated API calls (which Twitter does - version 1 of the API is now completely inactive, with the exception of API calls that were brought forward unmodified). By fundamentally changing the nature of programmer interaction with the APIs, web services are not bounded by legacy decisions in the API.

APIs should be easy to use and hard to misuse. An API is meant to facilitate a task, which implies that the API should make the task easier, not translate one hard task into a new hard task. To that end, the simplest tasks the API facilitates should be made easy, while still allowing more complex tasks to be performed. Additionally, the API should make it difficult, if not impossible, to perform the task incorrectly.

One of the most basic tasks a developer would need to perform when interacting with Twitter is accessing the *Tweets* (or statuses, as the Twitter API documentation calls them) for a particular account. This is achieved by issuing an HTTP request, such as the one shown in Figure 4. As a single GET request, which has no required parameters, the only complexity of this API call comes from authentication.

In version 1 of the Twitter API, the extra complexity for authorization would not have been required. Instead, one of two optional parameters would be specified to indicate which user's Tweets to access. However, as of version 1.1, Twitter requires that all API calls be authenticated to prevent malicious use of the API [7]. The validity of the decision to require authentication for security reasons aside, version 1.1 has made the simplest tasks more complex for the developer using the API.

On the other hand, by requiring the authentication field, it is more difficult to misuse the Twitter API. For example, version 1 of the statuses/user_timeline.json call implicitly expects one of two optional parameters to be specified, but does not enforce this in the API itself.

```
> GET /1.1/statuses/user_timeline.json
HTTP/1.1
> Host: api.twitter.com
> Accept: */*
> Authorization:
OAuth oauth_consumer_key="***",
oauth_nonce="***",
oauth_signature="***",
oauth_signature_method="***",
oauth_timestamp="***",
oauth_token="***",
oauth_version="1.0"
>
```

Figure 4: Sample request to the Twitter API. The value "*" in the authorization field would be replaced by the appropriate application and user credentials.**

It is our belief that the maxim of making an API easy to use and hard to misuse is of obvious value to web APIs. Regardless of the purpose, APIs are built to facilitate tasks and good APIs make core functionality both simple and stable.

APIs should be self-documenting. The perfect API requires no supplementary documentation because the API itself leaves no doubt about how it should be used. In practice, this ideal API is not reasonable because developers come from different backgrounds and therefore interpret the intention of the same API in different ways.

With a basic understanding of what Twitter provides as a service, the API is fairly clear about the functionality it provides. The one notable exception is the choice to refer to "Tweets" as "statuses". It was not immediately apparent to us, when reading through the API, which call would allow us to post a Tweet because of the mismatch in names.

As with traditional APIs, clearly and accurately express-

ing intention is very important, and this communication starts with the exposed calls of the API. By reading an API call and its parameters, the core usage should be clear to the user, to the extent that the user can identify if the API call is needed to perform the task of interest. We conclude that it is important for web APIs to be self-documenting.

Example code should be exemplary. If example code is provided, it will be used. It is extremely common for developers to copy example code, which provides functionality that is almost exactly what is desired, and make minor modifications to make it fit a desired purpose. This is especially true for examples that are provided with an API.

In version 1.1 of the Twitter API, all API calls require authorization and so it is not possible to provide complete examples in the documentation itself. With that said, Twitter does provide a sample construction of the URI that would be used in the HTTP request. Additionally, Twitter provides a tool which will construct sample API requests within the application management portal, but it is up the the user to specify the call being made and the parameters to use.

Web APIs, which are called via HTTP, are not as straightforward as calling a traditional library API call, because an HTTP request must be constructed and sent to an actual server that supports the request. As such, while it is sometimes possible to give complete examples that can be used directly by developers, it is often much more difficult. We believe that, in the context of web APIs, examples do not have the same place as they do in traditional APIs, and so this maxim is not as strongly applicable as it is for traditional APIs.

Avoid fixed limits on input sizes. As systems evolve, their ranges of acceptable inputs may evolve concurrently. By placing limits on input sizes the API is more likely to become obsolete, as it may not be able to handle input that was not originally expected.

One of Twitter's claims to fame is that it only allows 140 characters to be used per Tweet. Despite forcing brevity (and thus possibly some sorely needed wit) upon its user base, Twitter is an extremely popular service that continues to limit the input of its primary function to 140 characters. Another example of a limit placed on input sizes can be seen from the required q parameter to the API call search/tweets, which will not accept more than 1,000 UTF-8 characters. It is clear from these examples that Twitter is more than willing to disregard the maxim of avoiding fixed limits on input size.

A primary difference between traditional library APIs and a web API, such as the one provided by Twitter, is that a service on the other end of an API must process the request. If every user of the API was able to make unbounded requests to the API, the API's servers would quickly become either filled with the state for a small number of users or overwhelmed by processing requests. It is therefore important for web APIs to place reasonable restrictions on the input that they accept, making this maxim not applicable to web APIs.

Names matter. When an API is used in its intended manner, the code should read like prose. In other words, simply reading out the code should indicate what the API is doing on behalf of the user. If the API, when used as designed, is ambiguous or unclear when read out, it is a sign that the API will likely be misused by a user that expects to derive meaning from the names in the API.

The names of Twitter's API calls are grouped together according to functionality. For example, Twitter organizes a collection of Tweets into lists and there is a set of API calls that operate on collections of Tweets. Every API call that operates on a list embeds this fact into the URI itself so that it is clear to the developer what the API call is for. The result is meaningful names that help the developer understand each API call's purpose.

Making calls to web APIs does not encompass simply calling functions, however, but rather building and issuing HTTP requests. The result is that a string of API call names cannot be read directly out of a program that uses a web API. In the sense of an API call by itself, as with traditional API calls, the name should be sufficient to express the purpose of the API call. At a more abstract level, web API calls are often organized in a hierarchical fashion, which allows a set of API calls to be explicitly grouped, and can be leveraged to provide added meaning.

When in doubt, leave it out. The fundamental reasoning to err on the side of caution, when considering what to include in the API, is that, once something has been added to the API, it cannot be removed. Once a developer has used a feature of the API, it becomes very costly to remove that usage from their application. Alternatively, new additions to the API are easy for developers to incorporate.

As we found with the maxim relating to public APIs being forever, Twitter is clearly not afraid to remove entities from the API. In the switch from version 1 to version 1.1, 59 of the 131 API calls were completely removed from the API. This represents a very large deviation from traditional APIs, such as the .NET collections library example provided in Section 1.

One possibility is that Twitter was overly aggressive in deprecating functionality in the switch from version 1 to version 1.1. However, it was not the removal of functionality, but rather the added requirement of authentication, that caused the most stir in the community. We believe that the nature of web APIs is that there is an expectation of change, and thus the maxim of leaving entities out of the API is not relevant in a web scenario.

Keep APIs free of implementation details. The more implementation details are exposed by the API, the harder it is for a user to understand how to use the API. The user of a well-designed API should not be required to have knowledge of implementation details. As a general rule, this means avoiding over-specification in the API.

The Twitter API does not give away very much in the way of implementation details. In looking through the set of API calls that are exposed to a developer, it is basically impossible to tell what structures and algorithms are being used to store and operate on Tweets. From this we can say that Twitter definitely has kept the API free of implementation details.

By the nature of web APIs, there is a much larger degree of separation between the program using the API, and the implementation of the API itself. A program written against a traditional library API makes direct API calls to the implementation, whereas programs written against web APIs build requests that are sent across the network to the implementation of the API. The increase in separation implicitly makes it easier to hide implementation details, to the point that we feel this maxim does not apply to web APIs.

Minimize mutability. The inverse phrasing of this maxim

is to maximize idempotence. Mutability complicates the usage of an API and creates implicit state that must be kept in mind by the user of the API. In contrast, immutable state provides convenient properties, such as thread-safety, which makes it easier for the user to conceptualize the proper use of the API.

As of version 1.1 of the Twitter API, all of the calls that mutate state are made apparent by the nature of these calls being POST requests. Of the 100 available calls in version 1.1, 58 are only available via GET requests, making them completely mutation-free. The remaining API calls allow support for updating the state stored on the Twitter servers, such as posting a Tweet or sending direct messages to other users. Overall, we believe that Twitter has effectively isolated state manipulation API calls so that the mutability is minimized and idempotence is maximized.

In a web service scenario, idempotence is a very powerful tool for handling consistency when a request fails, by allowing the request to be reissued without fear of corrupting state. By eliminating mutability from web API calls, flexibility is greatly increased, and so the maxim of minimizing mutability is considered a central principle for web APIs.

Documentation matters. No matter how well-designed an API is, users need API documentation. The documentation is how the designers of an API communicate the intentions of the API, and this communication is imperative for developers to understand exactly what to do and what not to do. Good documentation is critical to the success of an API, as bad documentation can cause active harm.

Deciding whether or not documentation is "good" is a highly subjective decision. Instead of making authoritative statements about quality, we will instead provide the anecdotal evidence that, when writing this paper, the only source of information on how the Twitter API should be used came from Twitter's provided documentation. From our experience, the documentation was both complete and thorough, but did require a basic understanding of how to issue HTTP requests with authentication.

Regardless of whether or not Twitter meets the criteria for "good" documentation, we believe that the API documentation for a web API is just as important as the API documentation for traditional APIs. If a user cannot understand how to make the API perform required tasks, then the API has not met the basic criteria for success.

Consider the performance consequences of API design decisions. This maxim comes with the qualification that it should not be used as a reason to warp the API. Within reason, the API should be designed with performance in mind, such that the API itself does not cause artificial limits to performance.

An explicit component of the Twitter API is rate limiting users of the API. This is in direct conflict with this maxim, as it is a very explicit and artificial limit on the performance that the API can provide. The reasoning behind Twitter's rate limiting of client requests is to reduce opportunities for malicious users to abuse the API. For example, without rate limiting developers could write applications that pulled data from the Twitter servers at extremely high rates, to the point where the Twitter servers could not keep up.

Web APIs, unlike their traditional counterparts, have a natural tension created by the fact that behind the API is a set of servers providing the service. The greater the performance capability that the API provides, the more expensive it is for the provider to maintain the service. As such, web APIs are in a unique position where they must consider performance consequences of the API design, not only to allow acceptable performance for their users, but also to protect the service from malicious users and prevent denial of service.

Fail fast. Ideally, all problems would be reported at compile time. While this is not reasonable for all possible problems, the API should cause as many misuses to be reported at compile time as possible. If a misuse of the API cannot be reported at compile time, the failure should be made clear right away, so that the user of the API can handle the error as soon as possible, to avoid further issues.

The Twitter API is very flexible when it comes to accepting input. For example, passing garbage values for optional parameters, or adding extra parameters that are not part of the call will simply be ignored. However, the API will return error messages when required parameters are incorrect or missing. Overall, we believe that Twitter would have to be more strict about garbage input to truly qualify as failing fast. This is especially true if the user makes a mistake when constructing the API call, and does not realize their error because the Twitter API did not complain about the bad input.

A poorly formatted API call is easily caught by the compiler, but a poorly formatted message is more difficult to detect. In general, a failure when working with web APIs cannot be caught until runtime, making them more difficult to work with than traditional API calls. We believe that this makes the maxim of failing fast even more important for web APIs, as the developer cannot rely on the compiler to help them detect misuses of the API.

Use consistent parameter ordering across methods. If two different API calls take the same two parameters, it is more convenient for the user if the same order is used by both API calls. Changing the order of parameters between API calls easily leads to confusion for the developer trying to use the API, and should make no difference from the perspective of the API.

Within the documentation, the Twitter API adheres to the maxim of presenting a consistent parameter ordering. That said, actual calls to the Twitter API do not enforce a particular ordering of parameters, as the GET and POST parameters in HTTP requests do not need to be specified in a particular order.

The important distinction between web APIs and traditional APIs is that HTTP requires that the parameter name be provided along with its values. This allows calls to the API to provide the parameters in any arbitrary order, as long as the required parameters are present. Therefore, the maximum of using consistent order for parameters is not relevant for web APIs.

Avoid long parameter lists. The basic issue with long parameter lists is that they are easy to get wrong. This is particularly important if two or more parameters share the same data type, as the compiler will not be able to identify that the API is being called incorrectly.

As shown in Section 4, in version 1.1 of the Twitter API, 48 of the 100 API calls accept three or fewer parameters. Additionally, only four API calls have more than three required parameters, with the most required parameters being five. If we consider three parameters to be a reasonably small number of API calls, then the Twitter API has successfully

avoided long parameter lists. However, one of the major issues with long parameter lists in traditional APIs is that the compiler cannot distinguish a programmer's intention if an ordering error is made when providing parameters in a list. As we noted in the previous maxim, this is not an issue for HTTP, because the parameter name must be provided along with its value.

While avoiding long parameter lists makes the usage of the API more clear, there is certainly less pressure on web API designers to limit the number of parameters. Additionally, the ability to make parameters optional simplifies the basic use of an API call, while still allowing more complex invocations if required. We thus conclude that, while this maxim is not irrelevant, it is also not a first-class design concern for web APIs.

Avoid return values that demand exceptional processing. If the API requires additional code to be written by the user to handle special cases, it becomes possible for these edge cases to be forgotten. The example provided by Bloch is if an API returns NULL instead of a zero-length array. The programmer must now explicitly add code to check if the return value is NULL before operating on the array.

The Twitter API adheres to the maxim of avoiding return values that demand exceptional processing. An example of an API call in the Twitter API that follows this maxim is friendships/no_retweets/ids.json. The purpose of this API call is to list all user ids that the authorized user (on behalf of whom this API call is being made) has requested not to receive re-Tweets from. The return value of this API call is a JSON array, but if no such users have had their re-Tweets blacklisted then the return value is simply an empty array. In addition, because all responses are provided in JSON format, clients can make use of widely-available and powerful JSON parsing libraries to do any heavy lifting required to extract meaningful results.

As with traditional APIs, if the developer using a web API must explicitly check for special return values, it is possible for the check to be forgotten. In general, web APIs must avoid this, and also adhere to other reasonable decisions that reduce the amount of work that clients must do which could instead be done server-side. It is therefore still important for web APIs to avoid return values that demand exceptional processing.

6. DISCUSSION

From our exploration of Bloch's maxims for API design, we limit the list of maxims and introduce our own to produce the following set of maxims for web service API design:

1. APIs can be among your greatest assets or liabilities.
2. APIs should be easy to use and hard to misuse.
3. APIs should be self-documenting.
4. Names matter.
5. Maximize idempotence.
6. Documentation matters.
7. Fail fast.
8. Avoid return values that demand exceptional processing.
9. Consider the security consequences of API design decisions.

The novel maxim that we have added here captures the added importance of security in the design of a web API. Unlike traditional library APIs, a web API provides access to services that are executing and storing state on an external server. This is a fundamental difference which alters the core of the API design, and should be considered a top priority in all matters when designing a web-based API.

Although the results in this work are interesting, they are not necessarily complete. It would be interesting to see, for example, how an open source web-based API or protocol differs form the Twitter API in terms of design decisions and evolutionary patterns. Even a comparison against another API of the same sort might be enlightening, for example a direct comparison of the decisions made in the Twitter API against the decisions made in APIs provided by Facebook or Google (both of whom offer similarly large and popular web service APIs for a variety of different products).

7. CONCLUSION

In this paper we presented an investigation of good API design philosophies, with a particular focus on the relatively new scenario of web APIs. As an example of a successful web API, Twitter forms the foundation of our discussion. Of particular interest we have found that, between the two versions of its API, Twitter made drastic changes that go against the commonly accepted maxims of traditional API design set out by Joshua Bloch. By walking through the 16 relevant maxims and considering them in the context of web APIs, we have found that there are fundamental differences between web APIs and traditional APIs. These differences manifest themselves in a way which suggests that it is necessary to create a new set of paradigms which web service APIs should be designed to follow. Accordingly, we outlined a new set of design maxims specifically tailored to web API creation, which we believe would be appropriate and beneficial to consider when designing a web service API.

8. ACKNOWLEDGEMENTS

This research was made possible in part by funding from GO-Bell Scholarships. We would like to additionally thank Twitter for providing their API and associated public documentation.

9. REFERENCES

[1] Bloch, J. How to design a good API and why it matters. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications* (2006), OOPSLA '06, pp. 506–507.

[2] Clarke, S. Measuring API usability. Dr. Dobb's Journal, pp. S6–S9.

[3] Farooq, U., Welicki, L., and Zirkler, D. API usability peer reviews: A method for evaluating the usability of application programming interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2010), CHI '10, ACM, pp. 2327–2336.

[4] Grill, T., Polacek, O., and Tscheligi, M. Methods towards api usability: A structural analysis

of usability problem categories. In *Human-Centered Software Engineering*, M. Winckler, P. Forbrig, and R. Bernhaupt, Eds., vol. 7623 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 164–180.

[5] Microsoft. When to use generic collections. http://msdn.microsoft.com/en-us/library/ms172181(v=vs.110).aspx, 2014.

[6] Piccioni, M., Furia, C., and Meyer, B. An empirical study of API usability. In *Empirical Software Engineering and Measurement, 2013 ACM / IEEE International Symposium on* (Oct 2013), pp. 5–14.

[7] Twitter. Changes coming in version 1.1 of the twitter API. https://blog.twitter.com/2012/changes-coming-to-twitter-api, August 2012.

[8] Twitter. Overview: Version 1.1 of the twitter API. https://dev.twitter.com/docs/api/1.1/overview, May 2013.

[9] Twitter. REST API v1 resources. https://dev.twitter.com/docs/api/1, March 2014.

[10] Twitter. REST API v1.1 resources. https://dev.twitter.com/docs/api/1.1, March 2014.

[11] Wagner, J. The increasing importance of APIs in web development. http://code.tutsplus.com/articles/the-increasing-importance-of-apis-in-web-development--net-22368, October 2011.