Preliminary Analysis of REST API Style Guidelines

Lauren Murphy
School of Information
University of Michigan
Ann Arbor, MI 48109
laumurph@umich.edu

Tosin Alliyu
Computer Science Department
Haverford College
Philadelphia, PA 19041
alliyut@gmail.com

Andrew Macvean
Google, Inc.
Seattle, WA
amacvean@google.com

Mary Beth Kery, Brad A. Myers
Human-Computer Interaction Institute
Carnegie Mellon University
Pittsburgh, PA 15213
mkery@cs.cmu.edu, bam@cs.cmu.edu

Abstract

We studied a collection of 32 publically published guideline sets for designing RESTful Application Programming Interfaces (APIs), each from a different company, to identify similarities and differences to see if there are overall best practices across ten different topics. Our contribution includes providing a list of topics that API authors can reference when creating or evaluating their own guideline sets. Additionally, we found that while some guideline sets attempt to enforce consistency, simplicity, and intuitiveness in the APIs that use these guidelines, cross-guideline set comparisons show a lack of consistency in some of the topics examined, and different interpretations of what is thought to be "simple" and "intuitive."

Keywords API Usability, API Style Guidelines, Developer Experience (DevX, DX), Web Services

1. Introduction

Application Programming Interfaces (APIs), are the way that libraries of code, SDKs, and frameworks are made available to programmers [1]. Increasingly, companies are providing APIs so others can access their data and services. For example, as of October, 2017, programmableweb.com lists over 18,400 APIs for Web services. There are starting to be a number of public guideline sets that attempt to help internal and external programmers develop high-quality APIs. We are investigating to what extent the guidelines in the sets are consistent with each other, and whether there are any universal best practices.

Several different groups of people are involved with APIs and they have different goals [1, 2]. First, there are the "API designers", who build the APIs to be released for use by the "API users," who are programmers that make calls to the APIs in their own programs. Another group are the product "consumers," who use the products that are made with programs designed by the API users. Another group that was not mentioned in previous work [1, 2] is the "API Guideline Authors," who create API design guideline sets to help the API designers create better APIs. Previous research around usability and APIs has been centered around the API user's experience, and has covered topics on non-web APIs [3-5], software patterns [6, 7], and documentation [8-10], which has been called Developer Experience (DevX or DX), by analogy with UX for User Experience. However, there has been little research on how API Designers work or what their needs are. By focusing on published design guideline sets for APIs, we hope to better understand common design issues that API designers face. Our findings may be helpful for current API Guideline Authors and API Designers' understanding of how guideline sets differ and are the same. Additionally, it may serve as a reference for API Authors creating new guideline sets for themselves or their company by alerting them about popular topics to cover and what decisions other API Authors are making.

Previous research had identified a few sets of guidelines aimed at helping API designers, including two books [4, 5]. However, with the explosion of the number of APIs, we were interested in newer guideline sets used by API designers today. Initial searches on Google and Github for API design guideline sets revealed 39 different sets, of which 32 were for REST APIs, two for JAVA APIs, one each for the programming languages Swift and Rust, one for .NET, one for TUK, and one not specific to any language or architecture style. As a result, we focused our analysis on the 32 REST API guideline sets (see Table 1). REST is generally a method of communicating over the Internet to access web services.

Some articles have discussed design issues with REST APIs [11, 12]. However, we are not aware of any previous work examining the contents of API design guideline sets.

PLATEAU'17 Workshop on Evaluation and Usability of Programming Languages and Tools, October 23, 2017, Vancouver, CA

Table 1. API Guideline Sets Evaluated

	Creator Name	Word Count	URL
1	Adidas	10,512	https://adidas-group.gitbooks.io/api-guidelines/content/
2	aGiftKit	1,085	https://github.com/aGiftKit/apiguide
3	Allegro Tech	6,745	https://github.com/allegro/restapi-guideline
4	Amazon	1,381	https://developer.amazon.com/public/apis/experience/cloud-drive/content/restful-api-best-practices
5	Apigee	7,820	https://pages.apigee.com/web-api-design-website-h-ebook-registration.html
6	Atlassian	3,936	https://developer.atlassian.com/docs/atlassian-platform-common-components/rest-api-development/atlassian-rest-api-design-guidelines-version-1
7	Australian Digital Transformation Office	5,181	https://apiguide.readthedocs.io/en/latest/principles/index.html
8	Australian Taxation Office	3,487	https://github.com/ato-team/restful-api-design-guidelines
9	CDiscount	5,551	https://github.com/jMonsinjon/archi-api-guidelines/tree/master/src/docs/asciidoc/api
10	Cisco	9,463	https://github.com/CiscoDevNet/api-design-guide
11	Cloud Foundry	5,081	https://github.com/cloudfoundry/cc-api-v3-style-guide
12	Darrin	7,310	https://github.com/darrin/yaras/blob/master/restful-standards.md
13	Finnish Government	1,991	https://github.com/6aika/development_guide
14	Geert Jansen	9,174	http://restful-api-design.readthedocs.io/en/latest/intro.html
15	GitHub	3,210	https://developer.github.com/v3/
16	GoCardless	2,186	https://github.com/gocardless/http-api-design
17	Google	18,886	https://cloud.google.com/apis/design/
18	Haufe	15,446	https://haufe-lexware.gitbooks.io/haufe-api-styleguide/content/
19	Heroku	2,131	https://geemus.gitbooks.io/http-api-design/content/en/
20	IBM	2,370	https://github.com/watson-developer-cloud/api-guidelines
21	Inaka	1,598	https://github.com/inaka/rest_guidelines
22	Keboola	987	http://docs.keboolaconnector.apiary.io/#reference
23	Matteo Canato	1,463	https://github.com/mcanato/rest-api-standards
24	Microsoft	16,333	https://github.com/Microsoft/api-guidelines/blob/vNext/Guidelines.md
25	Paypal	3,813	https://github.com/paypal/api-standards/blob/master/api-style-guide.md
26	REST cheat sheet	826	https://github.com/RestCheatSheet/api-cheat-sheet
27	Squareboat	1,622	https://github.com/squareboat/api-guidelines
28	Thomas Hunter II	4,362	https://codeplanet.io/principles-good-restful-api-design/
29	Unoexperto	1,709	https://github.com/unoexperto/rest-api-design-guidelines
30	Vlad Mandrychenko	1,782	https://github.com/vmandrychenko/http-api-guidelines
31	White House	1,480	https://github.com/WhiteHouse/api-standards
32	Zalando	17,722	https://github.com/zalando/restful-api-guidelines

In this paper, we seek to understand differences and similarities in REST API design guideline sets.

2. Quick Introduction to REST

REST (REpresentational State Transfer) is a commonly used architectural style for web services. According to Roy Fielding, the inventor of REST [13], APIs that implement a RESTful architecture style to design their structure and behavior should be stateless, so that any changes in the implementation do not cause an API user's code to crash unexpectedly, have a decoupled client-server relationship, explicitly address cacheability, have a uniform interface, and optionally provide code on demand. An API that is RESTful will allow a user to provide a URI/URL and an operation – performed using HTTP verbs – in order to do some action on an object or set of objects stored in a server.

The following terms related to REST [2] will be used throughout the summary of our results:

- Resource - A collection of one or more homogeneous objects. For example, dogs.
- Identifier - A unique reference to a single instance of an object. For example, Fido23.
- Sub-resource - a resource that can be found hierarchically beneath an identifier, for example, Fido23/checkup_dates.
- Field name - the string name (also called keys or properties) associated with a value of an instance. For example, dog_name.

Guideline sets disagree about what the parts of the URL should be called. Usually, to reference a single object, the sets use the name "identifier" but others call it an "element," "document resource," "resource-id," "resource," or "document." Similarly, for referencing a collection in a URL, most sets use the term "resource", but some say "collection," "collection resource," or "resource-collection." We acknowledge that these differences exist, but since 16 of the 32 sets use the terms "identifier" and "resource" – and Fielding's dissertation [13] also uses "resource" to mean more than one object – for the rest of the paper we will also be doing so.

3. Methodology

To collect the guideline sets we evaluated, we began with an initial list of guideline sets known to the authors from well-known tech companies, along with the well-known API design books [4, 5]. From there, we obtained the rest of our sources by searching on both Google and Github with terms such as "API design guideline," "API styleguide," and "API Guidelines."

Twelve of the 39 guideline sets that we discovered can be found on a site called apistylebook.com, which provides outbound links as well as other material. Table 1 shows the full set of the 32 REST guideline sets and where they are located. The length of the sets we evaluated range from 826 words up to 18,886 words. Word count is one of the few common attributes among the different guideline sets, and acts as a proxy for how much content is in each set.

Once we gathered these sets, we began the task of qualitative analysis and coding. The first two authors read through all of the sets to understand what topics were covered. Then, they collectively identified 27 topics included in some or most of the guideline sets, such as naming conventions, security, and error responses (see Table 2 for the list of all topics). Definitions of all 27 categories were constructed by the coders as a reference to help when deciding which category a rule belonged in. The two coders split the sets in half, so that one person took odd numbered sets and the other even numbered. After separating the sets, both authors carried out the coding process for each guideline set using the 27 topics as categories to identify whether or not each guideline set includes some rule or discussion about that topic. In the case where there was ambiguity about how to code a particular rule, discussions were carried out either between the two coders or among the four authors to resolve these ambiguities. This identification stage allowed us to recognize patterns about omissions and unique additions of rules among the sets. In addition to this, we were able to recognize which categories out of the 27 are addressed by a majority of the sets and which categories are less discussed.

Next, with guidance from the other coauthors, we selected 10 out of the 27 categories for a more in-depth analysis. These 10 categories are discussed because they were either identified as interesting or important topics by practicing API designers or, through our initial analysis, they seemed to be the most discussed and/or controversial. These 10 categories are addressed in sections below.

With the 10 selected categories, we re-analyzed each guideline, gathering and noting what each guideline says about each category. We used a cross evaluation process so that each coder each worked on a different set of guidelines from the previous analysis stage. Under each category, we organized the information so that similar statements made by different guidelines were placed next to one another, which thereby revealed interesting patterns, ideas and concepts.

4. Versioning

A disagreement within the API design community, as evidenced by the conflicting recommendations in the guideline sets, is how to identify the version of the APIs. While Adidas, Google, and Zalando recommend using the three-number semantic versioning format, MAJOR.MINOR. PATCH, other guidelines suggest a two-number semantic versioning format, ordinal numbers with no decimals or dot notation, and some even propose date versioning. Although

[2] Some definitions adapted from the guideline sets from Thomas Hunter II.

Table 2. The 27 categories identified across all the guideline sets of Table 1. The ten highlighted were used for an in-depth analysis. "Frequency" counts how many guideline sets mentioned each category.

27 Categories	Frequency (out of 32)
Status Codes	30
Response Structure/Format	29
Standard Methods	29
Naming	28
Versioning	28
Pagination	24
URI/URL Structures	24
Error Response	22
Filter	17
HTTP Field/Header	15
Security	15
Backwards Compatibility	13
Naming Resources	13
Caching	12
Documentation	12
URI Field	12
Sorting	11
Action Resources	10
CORS	9
Long running operations	7
Rate Limiting	6
Gzip Compression	5
Metadata	4
Naming Collections	4
Custom Methods	2
Empty Responses	2
Rules for API Users	2

the Zalando guideline set uses a three-number semantic versioning format, they refer to the last number as DRAFT instead of PATCH, where the DRAFT number is only included for unreleased API definitions that are still under review. Adidas, Google, and Zalando each recommend similar rules for when to increase each number:

- Increment major version when you make incompatible API changes.
- Increment minor version when you add functionality in a backwards-compatible way.
- Increment patch version when you make backwards compatible bug fixes.
- Increment DRAFT version when you make changes during the review phase that are not related to production releases.

Another versioning issue about which the community disagrees is where to place the version: in the URL or in the HTTP header. Eleven companies require that version numbers be placed in the URL, while eight require it to be in the header. For example, GoCardless suggests that versions be passed in the HTTP header and provides the example: GoCardless-Version: 2014-05-04. In contrast, Paypal recommends that version number should be in the URL, providing the URL format: /v{version}/. The IBM guideline set is unique since it states the minor version should be passed as a required parameter that takes a date (i.e., ?version=2015-11-24), while the major version should be placed in the URL path with the format of a prefixed "v" followed by an ordinal number (i.e. /v1/). However, Apigee and the Finnish government's guideline sets provide the following rationale for deciding between putting the version in the URL or in the header:

- "If it changes the logic for handling the response, put it in the URL so it is easily seen."
- "If it does not change the logic for each response, like OAuth Information, put it in the header."

Overall, there is a lack of consistency across guideline sets, with the two different locations and four different ways of identifying a version.

5. Backwards Compatibility

Despite the fact that backwards compatibility is a key aspect of API design, only 12 of 32 guideline sets provide rules on backwards compatibility. When code that was written with, say, version 1 of an API also works with another version – such as 1.1 or 2 – then the changes implemented in the new version are considered backwards compatible. Since APIs might be used by up to millions of places, the costs of editing all of the code that uses the API can be very high. While eight guideline sets explicitly discourage making backwards incompatible changes, they still provide rules for versioning, for when breaking changes are unavoidable. CDiscount and Zalando state the same exact rule word for word: "Strongly encourage using compatible API extension and discourage versioning." (This points to an interesting pattern that we recognized among the API design guideline sets; they derive or copy rules from one another, often with

general citations.) Even though Zalando discourages versioning, they still provide specific rules for versioning and incrementing version numbers when backwards incompatible changes are made (as discussed in the previous section).

The two major principles mentioned within rules about backwards compatibility are Postel's law[3] and the rules for compatible extensions. Postel's law, also known as the Robustness Principle, states, "be conservative in what you send and liberal in what you accept from others." [14]. The Adidas guideline set is the only one that defines some rules for extending APIs:

- You MUST NOT take anything away.
- You MUST NOT change processing rules.
- You MUST NOT make optional things required.
- Anything you add MUST be optional.

Various guideline sets recommend that APIs follow Postel's law and the rules for compatible extensions in order to avoid versioning.

Another principle mentioned by just Microsoft is the Principle of Least Astonishment, where Microsoft states that anything in violation of this principle is a breaking change. The Principle of Least Astonishment states that the result of performing some operation should be obvious, consistent, and predictable, based upon the name of the operation and other clues [15].

The next sections discuss changes that different guideline sets noted as breaking and nonbreaking changes.

5.1 Breaking (Backwards-incompatible) Changes

Renaming, removing, changing or adding new field names are identified as breaking changes by some guideline sets. Cisco, Darrin, Google, and IBM all mention removing fields as a breaking change.

Cisco, Google, and Microsoft include a rule about changes to the behavior of an API. For example, Google specifically states that changing the behavior of existing requests is backwards incompatible. In addition to these, Cisco, IBM, and Microsoft identify changes to error or status codes as breaking changes.

5.2 Non-Breaking (Backwards-compatible) Changes

Darrin, GoCardless, Google, and Haufe all agree that, in some situations, adding or deprecating a field is not a breaking change. IBM agrees with this but restricts it to the addition of output-only fields. Cisco, GoCardless, and IBM suggest that the addition of optional parameters is non-breaking. Furthermore, adding new HTTP methods is considered a backwards compatible change by Atlassian and Google. The Atlassian guideline set specifically states that the addition of methods to existing resources is backwards compatible, while Google states that an HTTP binding to a method and adding a method to an API interface is not a breaking change.

Rules for determining backwards compatibility seem to be based off of what would intuitively be most likely to break code. Though there is disagreement about whether adding a field is considered backwards compatible, other rules do not seem to differ among guideline sets.

6. Naming

Naming in API design refers to the style and word restrictions for names of objects such as variables, classes, functions, identifiers, and resources. In general, guideline sets recommend using plural nouns rather than verbs. The Australian Taxation Office goes so far as to suggest simplifying a plural noun such as "parties" into "partys." Casing has less consistency across guideline sets, where 7 of 11 that mention case recommend using camelCase for field names, and 4 of 11 preferring snake_case. The preference for camelCase over snake_case may appear because guideline sets are attempting to be consistent with JavaScript conventions (which traditionally recommends camelCase), as mentioned in one of the sets. The guidance for the case of path segments of the URL are also split, with some preferring kebab-case while others say snake_case. Most guideline sets include rules about naming that aim for a consistent, intuitive aesthetic and grammatical style instead of specifying specific words that must or must not be used (although Adidas, Allegro Tech, CDiscount, Darrin, Google, and Microsoft all do mention some specific words to use or avoid). These kinds of rules are more widely applicable and apply across a set of APIs, whereas rules for specific words have limited application.

[3] Named after internet pioneer Jon Postel who wrote about this in the TCP specification.

7. URL/URI Structure

URL/URI structure refers to the decisions an API designer makes about the hierarchy of resources and identifiers when constructing a URL or URI. The structure of the URI/URL can reveal information about how the authors of the guideline set think about the relationships between resources and identifiers. With the notable exception of a few guideline sets (Australian Taxation Office, Geert Jansen, and REST cheat sheet), the majority suggest that nesting should stop after one sub-resource (resource/identifier/sub-resource). Interestingly, GoCardless discourages nesting altogether, instead using filters to extract identifiers from resources. Their reasoning: "Nested resources enforce relationships that could change and makes clients harder to write," is an interesting argument for maintaining backwards compatibility that no other guideline set addresses. While few guideline sets provide reasoning for the construction of relationships – many just stating the syntax to implement it – a few do discuss how to determine if a resource can function as a sub-resource. For example, if there is a one-to-many relationship and a set of identifiers are associated with an identifier, such as a set of messages that belong to one user, then the messages should be able to function as a sub-resource. Overall though, guideline sets do not often focus on low level implementation details of the URI/URL construction.

8. Response/Structure Format

REST API guideline sets include several rules about the structure and format of the response that API users receive after a method is completed. There is a general consensus that JSON should be used as the default format. However, Adidas and Darrin propose using HAL format. Several guideline sets disagreed about whether to pretty print or keep the response minified. GoCardless and IBM recommend that the response should be pretty printed by default while Allegro Tech, Heroku, and Squareboat recommend keeping the JSON responses minified.

Other rules under this category include the format to use for date and time (where some recommendations specified ISO-8601 format and others specified RFC-3339 format), how to present nested objects, and the format for linked elements, as well as rules about what object types to use within a response. While most guideline sets agreed on certain rules such as JSON format, time and date format, other rules such as which object types to use within a response and what format to use for linked elements are not as consistent.

9. Standard Methods

HTTP defines a variety of standard verbs – which indicate an action to be performed on a resource or identifier – and most guideline sets recommend using at least GET, POST, PUT, and DELETE as their methods. IBM is a bit different in that they only mention using GET, POST, and PUT, and Google, instead, maps GET, POST, PUT, and DELETE to their own set of five standard methods (List, Get, Create, Update, and Delete). Other methods are mentioned by various numbers of guideline sets: PATCH (by 17), HEAD (10), and OPTIONS (8). A few sets go into depth, explaining which verbs should be idempotent, cacheable, or have side effects. In addition, a few sets have rules about behaviors of some of the methods - such as permitting a "soft" DELETE that allows the API user to reverse a prior deletion – but most do not specify what kinds of actions can be done with the methods. Standard methods are a popular topic to cover in guidelines sets, likely because of their fundamental role in REST APIs. Analysis across guideline sets reveals a high level of consistency and agreement as well, which is not often seen in other topics.

10. Custom Methods

Custom methods still use HTTP Verbs to complete actions, but they are distinct from standard methods because their functionality is incompatible with traditional implementations of standard methods. For example, POST traditionally creates an identifier or modifies the content of one; however, Google has specified a custom method called Move, which uses POST to change the parent of an identifier. Only two companies mention custom methods. Cisco advises against creating custom methods, while Google outlines examples of common custom methods and any guidelines that would go along with them. Common custom methods outlined by Google are BatchGet (used on multiple identifiers), Cancel (stopping an outstanding operation), Move (mentioned above), Search (alternative to List with different semantics), and Undelete (uses POST to bring back a deleted identifier from within the last 30 days). However, other custom methods are possible. Custom methods have a low level of agreement due to the fact that only two guideline sets mention them, and disagree fundamentally about whether to make them available.

11. Error Responses

Sixteen of the guideline sets propose similar formats for the error response – the response that is sent back to API users when an error has occurred – which generally includes

a code, error type, and message. Code refers to either the HTTP status code or internal error code, error type provides further information about the type of the error (e.g., "Invalid Request") and message is the description of the error. While this is the basic format that the guideline sets propose, some sets have additional fields within their error responses. Darrin, the Finnish Government, and Matteo Canato state that error responses should include a common HTTP status code, a message for the developer, a message for the end-user, an internal error code, and links for where developers can find more information. On the other hand, the Australian Digital Transformation Office only recommends providing the message for the end-user, an internal error code and links for developers. IBM has the basic format of an error response code, error type and description, but states that the description can be optional. While there are some deviations, the most common, basic fields to include (code, error, and message) try to provide the simplest response while still providing enough information to be helpful.

12. Status Codes

Thirty of the 32 guideline sets provide rules for status codes – indicators of the success of an HTTP request – and many of the codes overlap. A few talk about status codes in a general sense, suggesting to use commonly understood codes but to be as specific as possible. Most provide a list of codes that should be used by APIs and a portion of them further delve into which specific standard methods can result in which status codes. For example, the status code 207 can only result from a POST action according to CDiscount. Of the 35 unique status codes mentioned, only six (codes 200, 201, 400, 401, 404, and 500) are mentioned by half or more of the guideline sets. As another popular topic besides standard methods, rules regarding status codes lack consistency in which HTTP status codes to use – though all adhere to the standard definitions for each individual code. A 400 error, for example, signifies a bad request. Twenty-one of the 32 guideline sets recommend using it, and Google and the Australian Taxation Office even provide specific situations that narrow down the cause of the bad request – out of range and failed precondition – while Paypal joins these two guideline sets in mentioning a third situation, an invalid argument.

13. Documentation

In addition to providing rules on how to design APIs, 12 of the 32 reviews guideline sets give suggestions on how to document the API. There seems to be significant disagreements among the recommendations that the different sets suggest. For example, one set, Thomas Hunter II, discourages the use of automatically generated documentation, whereas Allegro Tech, Australian Digital Transformation Office, and Haufe recommend using automatic generators for the documentation, and explicitly mention using Swagger (swagger.io) as the automatic generator. Also, Thomas Hunter II, after suggesting against using a generator, states that if a generator must be used, the generated documentation should afterwards be manually doctored and made presentable. Whereas the sets generally have good agreement about documentation, the amount of detail each set includes about documentation varies widely; for example, Google has an entire section that is just over 1,150 words, while Apigee has a small section of just two sentences.

14. Discussion

The rules themselves revealed patterns in popularity of topics and even sub topics inside of the categories. Of all the 27 categories identified, five of them (Versioning, Naming, Response Structure/Format, Standard Methods, and Status Code) are mentioned by at least 27 of the 32 guideline sets. Each have sub topics that some guideline sets would cover, some of which we mention above in the analysis. For example, many sets would talk about the grammar of naming, with sub topics focusing on nouns versus verbs or singular versus plural – with extra emphasis on resource names. Additionally, while most talk about casing, sub topics for casing focus on JSON field names, path segments, query parameters, and HTTP Headers. In some cases, the topics or rules mentioned in one guideline set are copied directly into another set, as mentioned earlier in the section on backwards compatibility. With each category in our analysis, we mention sub topics that are the most consistent or contentious between guideline sets, though not every rule in a category is also a part of a sub topic.

We note that with the lengths of the guideline sets varying so much, it would be very difficult for the shorter sets to cover the same content or provide equal depth. It is our impression that there could be some intent behind the length of a guideline set. While it might not always be the case, some API authors may, for example, write a shorter set so it is more easily digested so that the API developers could read it entirely. A longer set may then function more as a resource to reference.

Though a style guide is meant to enforce consistency for the API designers who follow it, cross-guideline set comparisons allows us to better understand where industry consistency is lacking, as seen in versioning for example. Other rules though, such as using nouns for resource names, are

consistent across almost all guideline sets. Additionally, the rules seem to agree that simplicity and intuitiveness are encouraged in the design of an API. The mostly consistent rule for path segments – not going deeper than a sub-resource – functions as a way to keep a URL simple and short. This has the benefit of promoting intuitiveness in that it is easier to create a mental model of the hierarchy of resources if you are limited in how deep they can run.

15. Limitations and Future Work

One limitation in our research so far is that it focuses solely on RESTful APIs. Although this allowed us to gain a comprehensive understanding of the issues and decisions related to the design of REST APIs, we have no comparison point for other types of APIs. In addition to this, a majority of the design guideline sets were written by either tech companies or people within the tech industry. This points to the issue of possible domain and field biases, since the guideline sets may have presented rules specific to the environment and requirements of the tech industry. Additionally, because we did not talk directly with API guideline authors about their rationale or intentions when creating the guideline sets, we cannot definitively know why certain decisions were made.

With these limitations in mind, our future work includes doing the same process we mentioned above in our methodology section for different types of APIs and then using this information to compare and contrast across different kinds of guideline sets. Apart from gathering APIs of different types, we may also consider evaluating APIs from different fields in order to examine and determine domain-specific conventions versus general conventions recommended for all APIs. Interviewing API guideline authors about their guideline sets may also provide useful information regarding the decisions they had to make when creating the sets and why there is conflict between sets.

16. Conclusions and Implications

Our work brings up interesting points of contention and agreement around decisions in API design that API designers and guideline authors may need to consider. For guideline authors, our categories may be a resource for understanding potential topics they may want to address as well as informing them of what may be most important to cover in their own guideline sets. API designers may benefit from understanding how other guidelines differ, either as a way to suggest an alternative to their own practices or to better understand why their set suggests a specific rule.

Acknowledgments

This work primarily took place while the first two authors were in an REU program at CMU, funded by NSF grant IIS-1644604. Additional funding was provided by a gift from Google. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the funders.

References

[1] B. A. Myers and J. Stylos, "Improving API Usability," *Communications of the ACM*, vol. 59, pp. 62-69, July 2016.

[2] J. Stylos, *Making APIs More Usable with Improved API Designs, Documentation and Tools*. PhD Dissertation: Computer Science Department, Carnegie Mellon University. 2009.

[3] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, "Comparing the Usability of Cryptographic APIs," in *2017 IEEE Symposium on Security and Privacy*, 2017, pp. 154-171.

[4] J. Bloch, *Effective Java Programming Language Guide*. Boston, MA: Addison-Wesley, 2001.

[5] K. Cwalina and B. Abrams, *Framework Design Guidelines, Conventions, Idioms, and Patterns for Resuable .NET Libraries*. Upper-Saddle River, NJ: Addison-Wesley, 2006.

[6] J. Stylos and S. Clarke, "Usability Implications of Requiring Parameters in Objects' Constructors," in *International Conference on Software Engineering (ICSE'2007)*, Minneapolis, MN, 2007, pp. 529-539.

[7] B. Ellis, J. Stylos, and B. Myers, "The Factory Pattern in API Design: A Usability Evaluation," in *International Conference on Software Engineering (ICSE'2007)*, Minneapolis, MN, 2007, pp. 302-312.

[8] J. Stylos, A. Faulring, Z. Yang, and B. A. Myers, "Improving API Documentation Using API Usage Information," in *VL/HCC'09: IEEE Symposium on Visual Languages and Human-Centric Computing* Corvallis, Oregon, 2009, pp. 119-126.

[9] B. A. Myers, S. Y. Jeong, Y. Xie, J. Beaton, J. Stylos, R. Ehret, J. Karstens, A. Efeoglu, and D. K. Busse,

"Studying the Documentation of an API for Enterprise Service-Oriented Architecture," *JOEUC: The Journal of Organizational and End User Computing*, vol. 22, pp. 23-51, Jan-Mar 2010.

[10] M. Robillard and R. DeLine, "A field study of API learning obstacles," *Empirical Software Engineering*, vol. 16, pp. 703-732, December 2011.

[11] A. Macvean, L. Church, J. Daughtry, and C. Citro, "API Usability at Scale," in *27th Annual Workshop of the Psychology of Programming Interest Group - PPIG 2016* Cambridge, UK, 2016, pp. 177-187.

[12] A. Macvean, M. Maly, and J. Daughtry, "API Design Reviews at Scale," in *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems (CHI EA '16)* Santa Clara, CA, 2016, pp. 849-858.

[13] R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*. PhD Dissertation, Information and Computer Science, University of California, Irvine. 2000.

[14] J. L. Ordiales, "Why you should follow the robustness principle in your APIs." July 27, 2017, https://jlordiales.me/2017/03/25/postel-law-api/.

[15] P. Seebach, "The cranky user: The Principle of Least Astonishment," in *IBM DeveloperWorks*, 2001, https://www.ibm.com/developerworks/library/us-cranky10/us-cranky10-pdf.pdf.