A Qualitative Study of REST API Design and Specification Practices

Michael Coblenz*, Wentao Guo†, Kamatchi Voozhian‡, Jeffrey S. Foster§
*Computer Science & Engineering Department, University of California, San Diego, La Jolla, California, USA
†Department of Computer Science, University of Maryland, College Park, Maryland, USA
‡Nokia, Inc., Sunnyvale, California, USA (work conducted at the University of Maryland)
§Department of Computer Science, Tufts University, Medford, MA USA
mcoblenz@ucsd.edu, wguo5@umd.edu, kirthigavoozhian@gmail.com, jeffrey.foster@tufts.edu

***Abstract*—REST APIs expose web services to clients. Although experts have recommended guidelines for REST API design, there is little empirical evidence regarding the relationship between adherence to guidelines and benefits to API consumers. We interviewed ten REST API designers to understand what practices REST API designers follow, what opinions they have of existing guidelines, and what challenges they face when designing and using APIs. Some guidelines were widely followed, and some were rejected as being inconsistent with good practice. The participants reported additional challenges: authentication and authorization are too hard to implement; specifications and documentation are frequently missing, vague, or outdated; and standard error reporting methods are insufficient for end users and for debugging purposes. Overall, our observations present opportunities for tool developers to significantly improve the situation by assessing and assuring conformance with guidelines; by aligning documentation with specifications and implementations; and by standardizing guidelines for API design.**

***Index Terms*—REST APIs, Web APIs, API design, empirical studies of programmers**

I. Introduction

Representational State Transfer (REST) is a widely used architectural style for web APIs. First proposed by Fielding more than twenty years ago [1], REST has grown and evolved significantly, and many authors have provided recommendations on designing and building usable REST APIs [2], [3], [4], [5], [6]. The design of REST aims in part to enable separate evolution and maintenance of client and server components, improving reliability of services relative to approaches that might require co-evolution of the two sides of the interface.

Unfortunately, poor API design can lead to security problems [7] and can cause delays and result in poor user satisfaction [8]. As shown by the proliferation of services to help monitor API errors [9], [10], real-world API failures are a significant concern in industry. However, despite a plethora of design recommendations, there is relatively little empirical evidence regarding which recommendations benefit REST API users, and few tools to help ensure developers follow those recommendations. We believe a key reason is that, while there is broad knowledge of good API design in general [11], [12], [13], [14], [15], there are few studies of good REST API design [16], [17]. Results from general API design studies may not provide sufficient guidance for REST API design, since REST APIs inherit structural requirements from the HTTP standard [18]. Moreover, REST APIs are language-independent, with both clients and servers written in multiple languages such as JavaScript, TypeScript, Python, Ruby, and Go. Finally, the client/server distinction has significant implications for modifiability, evolvability, and performance, since the organization running the server must pay for hosting costs. As a result, it is difficult to predict what kinds of tools would be most useful for REST API developers, and it is difficult for designers of REST APIs to choose practices that result in the most effective APIs to enable programmers to build reliable client applications.

In this paper, we address this gap by conducting an interview study that answers two research questions:

- RQ1: What opinions and practices do REST API designers and implementors have regarding existing REST API design guidelines?
- RQ2: What challenges do REST API designers face when trying to create reliable REST APIs?

Answering these two questions enables us to identify opportunities to build tools that help designers create better APIs. It is also progress toward an empirical basis for REST API design decisions. In the future, developers may be able to choose practices that have been found empirically to be helpful rather than using opinion-based advice.

We interviewed ten experienced REST API designers, implementors, and users. We asked about their experience and challenges with following REST API design recommendations, implementing and using secure APIs, testing and debugging APIs, specifying APIs, and using tools for implementing APIs. We used open coding to categorize participants' statements, and then we formed a set of themes by grouping related codes. Our goal was not to recommend participants' opinions for use but rather to leverage their opinions to identify opportunities for empirical evaluation and tool development. (Section II describes our methodology.)

Some guidelines, such as making APIs stateless and using JSON to return responses, were widely endorsed. Others, such as using hypertext as the engine of application state (HATEOAS [1]), were rejected as not being worthwhile or being inconsistent with good practice. Still others, like following HTTP verb conventions and making interfaces uniform, were

recognized as useful ideas that were difficult to follow in practice. (Section III-B discusses RQ1.)

Additionally, participants identified a number of other challenges in designing and using REST APIs: Even though a variety of libraries and external services provide these features, authentication and authorization are still difficult to implement. Missing, vague, and outdated API documentation is a critical challenge in using REST APIs. HTTP error codes are insufficient for conveying information about errors to clients and end users. And forward and backward compatibility of REST APIs are essential for API evolution but are difficult to get right. (Sections III-C through III-H discuss RQ2.)

Our findings suggest an opportunity to develop tools to make REST APIs easier to develop and use. Participants would benefit from tools that check REST API specifications, which formally declare properties of API inputs and outputs, against client and server code. The community could develop a standard approach for relaying more detailed error messages. And, given machine-checkable REST API specifications, tools may be able to identify potentially incompatible changes as APIs evolve as well as automatically generate examples as documentation. (Section IV describes proposed tools.)

In summary, this paper's main contributions are

1) An analysis of REST API guidelines and practices, as currently followed by designers and implementors, and
2) Proposals about how languages and tools could make application and REST API development easier and safer.

Our work is the first of which we are aware to compare and contrast REST API design practice with REST API design guidelines. Although our focus is on REST, REST APIs share similarities with non-REST APIs, especially those in distributed systems, so addressing challenges with REST APIs may also benefit other kinds of APIs.

II. Method

Because of the difficulty of recruiting experts, we recruited a convenience sample of ten programmers who each had at least a year of experience developing REST APIs. When recruiting our participants, we first asked them to complete introductory surveys. We obtained informed consent and conducted semi-structured interviews via Zoom that lasted up to 90 minutes. Our study was approved in advance by our IRBs. Our participants were willing to participate as volunteers.

We first asked general questions such as "What were the most challenging parts in learning how to develop an API?" Then, we asked them to focus on a particular REST API they had worked on recently. We asked about several REST API guidelines (RQ1; Section III-B). We used prior work [19], [2], [3], [4] to construct a list of guidelines to ask about. We focused on those (Table II) we thought had significant implications for API design practice or offered opportunities for tool development. In the interview, we summarized each guideline and asked for opinions. We also asked participants how they would improve the guidelines.

Afterward, we asked about various aspects of API design that we hypothesized, based on the literature, might be particularly challenging or benefit from additional tool support (results in Section III):

a) Security: We asked what security problems participants had seen in their and others' APIs as well as what techniques participants use to ensure their designs are secure. We also asked about the security benefits of frameworks and languages, such as Python.

b) Bugs: We asked if there were design choices that were more likely to lead to bugs or vulnerabilities in client code.

c) Testing and Debugging: We asked about techniques to make APIs more debuggable and about techniques used for testing. We also discussed the requirements that participants focused on in testing, such as performance and scalability.

d) Specifications: We discussed specification approaches that participants use, such as OpenAPI; the alignment between specification techniques and participants' needs; and which aspects participants did not document but wished they had.

e) Tools: We asked for opinions about the frameworks, languages, and other tools participants use when implementing APIs and how those tools influenced their designs.

f) Errors: We asked how API implementations report errors and how applications handle errors.

g) Data interchange: We asked what data formats participants' APIs transmit and why.

After each interview, the first and third authors discussed and summarized the findings. We conducted interviews until the newest interviews revealed only minimal additional information. The third author, who had extensive experience implementing REST APIs, transcribed the recordings and conducted an initial open coding on the transcripts. Then, the authors worked together to review and revise the codes and then identify themes among the codes. Our focus is on identifying hypotheses, not on making statistical claims, so we did not compute inter-rater reliability [20].

Limitations: Because of the multitude of recommendations for REST APIs, we focused on guidelines and challenges we felt could inform practice and the development of tools.

There is a risk of bias due to participant selection. Although the practical difficulties of recruiting experienced API designers limited our recruitment to a convenience sample, we recruited broadly from people in multiple organizations, and who worked on differently-sized projects, although medium-size organizations may be under-represented in our participant pool. There is also a risk of misinterpretation when analyzing interview data. To mitigate this risk, multiple authors with different perspectives read and discussed the data, coding scheme, and resulting themes.

Due to an error, P6's recording is missing. We instead analyzed notes taken during the interview.

III. Results

After describing the participants (Section III-A), we present the results of our analyses of the interview data. Section III-B compares and contrasts existing REST API design guidelines with the participants' practices.

TABLE I
DEMOGRAPHICS OF INTERVIEW PARTICIPANTS. "LARGE" MEANS AT LEAST 1000 EMPLOYEES.

ID	Education	Org type	Roles	Design exp.	Spec. tools	Impl. APIs in	Invoked APIs in
P1	BS (unspecified field)	Large consultancy	Software eng.	$\geq$ 3 years	OpenAPI, RAML	Java	Java
P2	MS communication eng.	Large consultancy	IT solution analyst	$\geq$ 3 years	OpenAPI, RAML	Java	Java
P3	MS (not CS)	Govt. consulting	Personal projects, Software eng.	$\geq$ 3 years	None	PHP, Python, Go	PHP, Python, Go
P4	MS CS	Govt. consulting	Design, impl., docs. integration	$\geq$ 3 years	OpenAPI, Stoplight.io	Ruby, Python, Go, JS	Ruby, Python, Go
P5	CS PhD student	University	Researcher	$\geq$ 3 years	OpenAPI	JS	JS
P6	BS CS	Large company	Software eng.	$\geq$ 3 years	None	Java, JS	Java, JS
P7	Sc.B. CS	University	Organization service projects	$\geq$ 3 years	None	Python, JS	Python, Go
P8	BS CS	Large company	Developer, project manager	1–3 years	OpenAPI	Python	Python, Java, C#
P9	CS PhD	Startup	Software eng.	1–3 years	OpenAPI	Go	Go, Python, Bash
P10	BS Info & Decision Sci.	Large company	Senior software eng.	1–3 years	OpenAPI	Python	Python, C++, Objective-C

Sections III-C through III-H discuss the challenges participants identified and participants' recommendations for addressing those challenges. Some of those challenges lead directly to bugs, and others divert developer attention and add unnecessary complexity. Participants described the difficulty of using third-party authentication and authorization frameworks and security testing. They also cited insufficient documentation of input requirements as a key source of bugs. In spite of research proposing automated testing methods, participants reported using manual testing methods—finding automated approaches, such as property-based testing—too complex. Missing, vague, or outdated documentation and specifications are key challenges. Participants also told us that generating and reporting errors is difficult, inconsistent, and bug-prone, and that forward and backward compatibility remains a key challenge despite using versioning techniques.

The paper supplement includes a list of all the codes we used and the number of occurrences of each.

A. Participants

We recruited ten participants ($N = 10$). Table I shows key characteristics of the participants. Four identified as women and six as men. Participants were located primarily in the US, though one was in the UK and one was in India. All participants had bachelor's degrees, eight of which were in computer science or related disciplines. Three participants had master's degrees, and one had a Ph.D. Two were Ph.D. students with significant experience with REST API development, and eight were employed in the software industry. Participants worked in contexts ranging from individual projects (P3, P5) to large corporations with many stakeholders (P1, P2).

All participants had some API design or implementation experience. Six reported three or more years of experience, three reported between one and three years, and one did not respond on the survey but confirmed experience verbally. The participants reported experience implementing REST APIs in Python (5), Go (3), JavaScript (3), Java (2), and PHP (1). They had specified their APIs with tools including OpenAPI (7), RAML (2), and Stoplight.io (1).

We also asked participants about their experience *using* REST APIs. Six reported three of more years of experience, three reported one to three years, and one did not respond. They reported experience using REST APIs in Python (6), Go (4), Java (3), Bash (1), C# (1), C++ (1), JavaScript (1), Objective-C (1), and PHP (1).

B. API Guidelines

RQ1 asks: *what opinions and practices do REST API designers and implementors have regarding existing REST API design guidelines?* We addressed this question by aggregating guidelines from several sources [1], [5], [6], prioritizing guidelines that seemed to represent opportunities for tool development or which we believed had significant impact on API quality.

Table II lists guidelines we discussed and summarizes our findings about each one. Key findings include the inability of developers to assess uniformity; the universality of stateless design; and the ambiguity of HTTP verb semantics. Next, we describe our findings regarding the guidelines.

1) Use a Uniform Interface: Uniformity, i.e., *consistency*, is commonly recommended for REST API design, and can enable users to predict how an unfamiliar part of an interface will behave based on experience with other parts [21]. However, the open-ended, vague nature of the guideline leaves our participants confused about what they could do to obtain or check uniformity. When asked how to ensure uniformity, P3 said there was a "part of me that is never satisfied with anything I do" regarding naming and other patterns. P3 later said the hardest part of designing a REST API is structuring and naming the interface. P7 wanted a tool to test and quantify uniformity. P8 mentioned that organizing resources to be future-proof is challenging: "I feel like you always end up eventually wanting to…merge something or change something."

Uniformity pertains *within* an API, but uniformity can apply across APIs as well. However, this uniformity can trade off with other design decisions. P5 and P3 favored uniformity across APIs over simplicity. "Dailymotion and

TABLE II
SUMMARY OF REST API GUIDELINES AND CORRESPONDING FINDINGS

Guideline	Source			Summary of Findings
	(a)	*(b)*	*(c)*	
Use a uniform interface	•	•		Designers cannot assess uniformity or choose the most appropriate uniform approach
Use HATEOAS [19]	•	•		Well-defined formats (e.g. JSON) must be extended with application-specific semantics; standard formats, such as HTML, do not suffice.
Statelessness	•	•	•	Good for scalability; can be tricky in complex applications
Provide code on demand to clients	•			Avoid due to security risks
Use HTTP methods and follow verb conventions		•	•	Verb conventions are followed inconsistently; need community standards and checking tools
Filter, sort, and paginate to limit response size		•	•	A standard is needed for pagination parameter naming and semantics.
Use JSON to transfer data			•	JSON is conveniently human-readable, but code receiving JSON inputs must be flexible

(a) Fielding [1] (b) Microsoft [5] (c) Stack Overflow [6]

YouTube... have pretty much the same data, but they're completely different. I'd have to learn them from scratch—they use different type of authentication, different endpoints... but if [there's a choice between] the simple and having standardized API, I would go with the standardized one" (P5). P3 phrased the issue in terms of opinions about design: "I think if you provide appropriate tooling that enforces those opinions—whatever you decide they are—it is possible to make dramatic improvements... because it's ultimately *different assumptions* that lead to a lot of problems."

2) Use Hypertext as the Engine of Application State (HATEOAS): According to Fielding, "A REST API should be entered with no prior knowledge [of the data format and API semantics] beyond the initial URI (bookmark) and set of standardized media types..." [19]. In contrast, our participants universally described REST APIs as having documentation and specifications that describe data formats and API semantics. Perhaps the large range of possible data types and semantics have precluded standardization, though JSON is widely adopted. This represents a compromise, since standardized tools can parse JSON, even if the semantics are API-specific.

3) APIs Should Be Stateless: P3, P4, P7, and P8 try to maximize statelessness. P3 said, "State in general is a pain in the [rear]." P7 described learning from Uber [22], which transitioned a monolithic codebase to microservices, which often offer (stateless) REST APIs. P4 said that stateless APIs were less likely to break if a client accidentally made the same request multiple times. P8 said statelessness was required, as their hosting method involved Dockerizing and deploying multiple instances of an API.

However, P4 said while they "generally do want things to be stateless where possible," there are cases where state is useful: If there is "an accumulation of a complex state" as the user is "interactively working through something, building something"—e.g., a shopping cart for a web store—then some state might be stored on both the client and the server. Another exception is when the server must be authoritative, e.g., a server should not simply trust a client's authentication cookie.

4) APIs Can Provide Code on Demand to Clients: Fielding [1] proposed that APIs can send code to clients, but this practice was derided by P3, P7, and P10. As P10 put it, "That's just a huge security hole waiting to happen."

5) Use HTTP Methods and Follow Verb Conventions: HTTP requests are tagged with a verb, such as PUT or POST, which indicates some high-level semantics of the request. However, participants reported that verb conventions are only followed inconsistently, despite their potential benefits.

P9 expressed support for verb conventions because they reduce developer confusion. P1 said that not following verb conventions is bad design and is hard to fix later because it requires rewriting the API. As an example, P1 described an API for fetching data that used a PUT method with a user ID in the request body. This was less convenient than using GET because the request body had to be handled with every call.

In contrast, P2, P7, and P8 gave reasons for breaking verb conventions. P8 reported rarely using PUT and PATCH because "myself and everyone I work with are just so used to using GET and POST that... that's on your fingertips." P2 uses POST to fetch data if the number of required query parameters exceeds the limit for GET. P7 uses POST for all actions that push things, including deletions, because of a particular implementation choice they made. However, P7 also said that if they had to redesign their API, they would probably use conventional HTTP verbs.

P3 observed widespread deviance from HTTP verb conventions, which they attributed to a lack of enforcement. P3 said, as there are no consequences for breaking conventions, "it just feels too loose to be a standard, because it'll just accept any kind of behavior that you program it to accept." P3 blamed this on the disorganization of the REST community, saying, "it's kind of ownerless."

P5 argued that REST APIs would be easier to use if they were more object-oriented: a client could complete some action with one call instead of stringing together multiple HTTP requests. P5 acknowledged that this might not be as efficient and proposed an intermediate API to address the performance concern.

6) Filter, Sort, and Paginate Data to Limit Response Size: REST APIs must limit response sizes to maintain good performance. Participants reported that *pagination*, in which multiple

API calls are used to return different pages of a lengthy result, is one area that might benefit from standardization. P5 said that, frustratingly, different APIs deal with pagination in different ways. P8 indicated that the names used in pagination often differ across APIs: "Some APIs use `skip` and `take` parameters, specifying where in the results to start and how many to give.... I have seen some applications that just use a token, like a next page token or last page token."

7) Use JSON to Transfer Data: As discussed earlier, JSON is a widely recommended data transfer language, and our participants indicated it is widely adopted. Several participants had strong feelings about JSON compared to other alternatives. P3 said they "hate" being required to use XML, and P8 said XML is "a pain to work with." P4 prefers JSON to XML because it is more compatible with their code. P3 also prefers JSON to CSV, which they find harder to lint and more likely to yield results that cannot be reliably parsed or interpreted. The human-readable nature of JSON makes it easier to debug.

Participants also reported problems with JSON. First, client applications typically expect a consistent set of fields to be populated, which leads to bugs if the server omits fields (P5). Code that accepts JSON must be written in a flexible way, lest it break when the server sends an unexpected response. P10 said, "you kind of have to be willing to accept a missing value or something." Second, this approach requires carefully encoding data, typically as text. This can be particularly inconvenient when both ends of the API expect Java objects (P2). Third, checking conformance to the specification requires additional tools, such as Swagger.

Some participants use a specification approach such as Protocol Buffers [23]. Like Swagger tools, this ensures the data conforms to a specification, but Protocol Buffers represents the data more compactly (in binary rather than JSON). However, protobuf data is inconvenient to debug, since it is not human-readable. This approach is also inconsistent with Fielding's architectural goal of decoupling clients and servers. In some cases, a protobuf layer was wrapped in a JSON layer to provide convenient interfaces for both internal and external clients.

C. Documentation and specification

Five participants volunteered on the survey that missing, vague, or outdated documentation is a key challenge when using REST APIs, and we received 14 comments about outdated documentation in the interviews. P4 and P10 said that keeping documentation up to date is a key challenge when maintaining REST APIs. P3, P5 P7, and P9 recommended including *examples* as a particularly effective form of documentation.

We found support for documenting all aspects of an API design: the request parameters, the response body, possible error messages, and the meaning of each HTTP verb for each endpoint. Participants also suggested documenting which APIs can return large volumes of data. P8 mentioned that documentation often neglects some important constraints, such as disallowed parameter values or invocation ordering constraints. P8 explained how these omissions arise: "either because I just forgot to document it or it wasn't [an] expected constraint, [I] sort of figured [it] out after the fact."

Six comments extolled the benefits of examples. Four emphasized the benefits of *executable* examples in particular. "Facebook [has] an example ready to go, and you just type in your API key... and you can just run that against your own account... so you don't have to try to look up examples on Stack Overflow..." (P7). Examples and learning-by-doing have also been found to be effective in other domains, such as mathematics [24], and there is a substantial body of work showing that examples can lead to superior skill acquisition compared to studying only abstract knowledge [25].

P2 explained the challenge of documenting fields appropriately for multiple client perspectives. "I created the API, which was more looking into one particular line of business... but then when I moved into the corporate world, their description was completely different."

Fourteen comments reported that documentation is difficult to keep up to date. A common theme was that the only way to ensure documentation is up to date is to automatically generate it from the source code. "There are people who are like 'code should be simple and self documenting,' and I'm like, 'no, it should *generate* documentation'" (P10). P8 and P10 use tools that extract comments from source code to generate documentation, but comments can be outdated. P10's group uses a tool to generate reminders every six months for engineers to check documentation. P10 also remarked: "I will certainly start from the documentation. But the minute my observed behavior deviates, I'm calling the [author]."

Since incorrect or outdated documentation is a key challenge, tools that automatically generate documentation from implementations can be valuable. As P2 remarked, "the Swagger documentation is the truth." Seven participants reported using specification tools for REST APIs: OpenAPI, Swagger, or RAML. Five had experience with Protocol Buffers [23], which are used to describe binary interfaces in general. These specification tools can have both significant costs and benefits. As P10 explained: "We said, 'Hey, we're going to use Swagger to make this API easier to consume.' And then my employees came back to me and said, 'Man, Swagger is a pain in the [rear].' And I said, 'is [it worth the hassle]? If we don't use Swagger, and I send every person that has a question about this API to your desk, how are you going to feel?' And they said, 'well, okay, I'm gonna get Swagger working.'"

Design processes varied. P7, P8, and P10 added endpoints one at a time as needed. P1 favored writing a specification ahead of time: "We should always go to our specification first, because that's the entry point of our API development."

D. Errors

Participants indicated that it can be unclear which error codes to use; that writing and reporting human-readable error messages is too hard; and that it can be difficult to write correct error-handling code in API implementations.

The HTTP standard [18] specifies a collection of *status codes*, which the server sends in responses. For example, code

500 indicates *internal server error*. Unfortunately, status codes do not provide enough information for the client. As a result, APIs typically send a natural-language error messages in an ad hoc format. Several participants (P3, P8, P9) proposed a standard JSON field for a human-readable error message. P2, P4, and P9 use standard structures for all error messages. P9, for example, returns a JSON object with three fields: the HTTP error code, an API-specific error code (whose meaning is explained in API documentation), and a textual error message.

It is not obvious what information to include in error messages. To aid developers of client applications, messages should be precise and contain detailed information in terms of API abstractions. However, if the client application simply displays such messages to the end user, then a detailed error message may be confusing rather than helpful. Since error messages are not typically localized, they may not even be readable by the end user. P3 observed that too-detailed error messages can leak sensitive information about a server.

Another challenge with textual error messages occurs in APIs that are implemented in terms of other APIs. If API A receives an error message from API B, it is not clear how to use any textual component of the error message to generate an appropriate error message for A's clients.

When generating HTTP status codes, it can be unclear whether to return 200 ("OK") or 500 ("internal server error"). P6 said that if 200 is returned when even a minor error happens, the user can get the wrong impression as to the outcome. In addition, returning 200 for minor errors can cause tools that track API success rates to fail to raise necessary alarms. On the other hand, for debugging purposes P3 found it more convenient to send 200 with an error message in cases of invalid input. We did not find other participants who seemed to agree, although P7 writes client code that interprets both 200 with an error message and 500 as errors.

Implementing error handling can be challenging: when failures occur, the implementation may need to revert state back to before the failure. P7 recommended: "Only ever commit state at one point in your program, or do it in a way that rollback is cheap."

E. Forward and backward compatibility

In the pre-interview survey, four participants reported backward compatibility as a primary concern. Typically, maintainers either do not control all API clients or, even if they do, cannot kill and restart all clients at once. If the client is a web app, users may use an old version of a web app for a long time if they do not refresh their browser window (P6). Thus, backward-incompatible changes, such as renaming fields, are problematic. Introducing incompatible changes requires recognizing the incompatibility and introducing a versioning scheme, but this recognition can be error-prone. P6 recommended anticipating future needs and building in opportunities for extensions (*forward* compatibility).

F. Security

We asked participants what security problems they had observed and what they do to make sure their APIs are secure. Themes included the challenges of handling authentication and authorization and unclear best practices for testing.

Although authors have discussed the challenges of handling authentication and authorization since at least 2016 [26], the situation has not been resolved. Authentication typically requires using complex external services, such as OAuth [27]. According to P7, one tricky aspect is managing state and data across the different services that implement authentication.

To avoid having to store password information, P8 uses Amazon Cognito [28]. P10 also tried to use Cognito, but failed: "As somebody developing services…I never want to store PII on my system. I want somebody else to store it and give me some surrogate ID that I can use to track the user within my application." But: "After a week I just gave up. …There's all these keys and certificates…" P10 proposed that someone should create a solution analogous to *Let's Encrypt*: "Somebody needs to do that for app-level authentication and authorization, because it's never *not* been a nightmare."

For authorization, P9 uses Open Policy Agent [29] for defining access control policies. OPA provides a language, REGO, to express access control policies. P9 tries to align the resource paths with the authorization requirements so the system can easily reject inappropriate requests. That is, the software engineering considerations for *implementing* authorization impact the *API design* itself.

For testing security, participants used fuzzing (P3, P6); testing by security experts (P3); review by a security team (P2); and public bug bounties (P3).

G. Causes of bugs

We asked participants whether there are particular API design or implementation choices that lead to bugs. Participants reported that insufficient documentation of input requirements leads to bugs. "[The] author thought their API was well-defined, but you try to use it, and …this is an ID. What is it an ID of?" (P6). A particularly insidious cause of bugs is when misformatted input is not detected, e.g., when an incorrectly formatted date is sent (P6). Likewise, API implementations frequently fail to validate their input sufficiently, causing unsafe or incorrect behavior on user inputs (P1, P6).

Implementation languages and frameworks also introduce their own bug patterns. Dynamically typed languages, such as Python, can result in incorrect behavior. For example, if a user passes a string instead of a collection, iteration is over the characters in the string rather than the single string in the input (P3). As another example, in Go, if an invocation's request context is omitted in a database call, that call cannot be cancelled, which can lead to leaked threads (P9).

H. Testing Methodology

Despite existing research on automated test generation, participants (P1, P3, P4, P8) reported primarily using manually written tests. Although some said testing performance is important, none of the participants reported writing dedicated performance tests; instead, they use tools or re-purpose unit

tests for this purpose. P8 monitors response times once features are deployed and redesigns if they are slow enough to impact user experience.

P9 uses PyTest, while P3 writes Bash scripts that use tools such as `curl`. Tests both check correctness and help establish rate limits. Participants also use unit testing and fuzzing to identify edge cases that cause bugs. P3 observed that testing is inevitable because it is challenging to consider all possible edge cases when writing a specification. P9 indicated that integration tests are critical to check the whole workflow. P8 uses low-cost tests in continuous integration but uses Docker for more thorough integration tests before deployment.

Participants felt that the benefit of property-based testing [30] was not worth the cost. P3 said, "I feel like software engineers tend to be curious and attracted to property testing, and then they kind of look at it and go, I could spend X amount of time trying to figure out how to do this, or I could just point a fuzzer at this, and I would get good enough of a result with less time…."

IV. Discussion

Based on the challenges reported by participants, we believe there are several opportunities to improve REST API development and use. New language-based tools may be able to infer specifications from implementations, keep documentation consistent with implementations, and aid in automatic generation of test cases. Authentication and authorization frameworks are still too hard to use and represent a research or design opportunity. Tools could leverage specifications to help developers identify some cases in which version numbers need to change. Finally, although enforcing guidelines currently relies on human analysis, new tools may be able to analyze adherence to guidelines automatically.

A. API Documentation and Specification

We found that documentation is a key challenge with using REST APIs (Section III-C), including ensuring that documentation stays up to date. There are existing approaches to specifying APIs [31] and performing some limited checking of APIs using specifications [32], [33], [34], but we believe there are several unmet needs.

First, new tools that use type checking could ensure clients and servers conform to API specifications. As with compiler designs, rather than building one tool per combination of client and server languages, we propose building one tool for each client language and one for each server language. Although many clients are written in dynamic languages, recent years have seen widespread introduction of static typing capabilities for such languages, including TypeScript for JavaScript [35], PyType [36], and Sorbet for Ruby [37]. Thus, we believe the time is right for building static analysis tools for enforcing API specifications, leveraging these type systems.

Second, to help make documentation easier to create, new tools should automatically generate specifications from implementations. We believe this can be done using type inference with an expressive type system to discover the types of input parameters and of API results on server-side code. Such an analysis will need to be path-sensitive [38] to separate the different request–response combinations.

Third, given how helpful our participants found examples, we propose tools to automatically generate examples from specifications. Such a tool would likely need to be dynamic since the examples would be concrete, and static analysis is less good at tracking concrete values. Interactive authoring via tools such as CodeScoop [39] might also be possible.

Automated testing tools could leverage stronger specifications. Schemathesis [32] enables property-based testing for APIs that are specified with OpenAPI, but OpenAPI specifications do specify relationships between inputs and outputs. RESTler [33] is a stateful REST API fuzzer, but it cannot distinguish between correct and incorrect responses that do not return error codes. We found a need for an approach that is simpler for users than property-based testing and more thorough than RESTler.

B. Error Handling

The loose coupling between REST API servers and clients makes propagating errors difficult, as described in Section III-D. HTTP includes standard error codes, but as there are relatively few codes, they are generally not very specific. This can make it difficult for a client or an end user to handle the error. We believe that a new approach is needed.

We propose that the community develop a standard approach for relaying detailed error information. This could take the form of a specific JSON response field with an error code whose meaning is documented in the API specification. Since the documentation is separate, it could be localized to allow speakers of a variety of languages to understand the meanings of errors. The field could also include an optional structured component to carry error-related values, as is common in many languages with exceptions. For example, if a request includes several fields of data, one of which is invalid, the error could describe which field is invalid. Tools for checking and inferring documentation (Section IV-A) could be extended to document under what conditions errors are emitted.

C. Security

Participants indicated that handling authentication and authorization in REST APIs is an important challenge (Section III-F). P10's proposal to make an authentication/authorization framework akin to *Let's Encrypt*, which would make appropriate recommendations for each developer, is a compelling opportunity. *Let's Authenticate* [40] may meet some of the needs but we are not aware of a usability evaluation of that system. Information flow-based approaches could help developers identify when sensitive information could leak from an API [41]. Best practices for security testing should be better defined, particularly for developers who lack access to security experts.

D. Versioning and Forward and Backward Compatibility

Participants reported that versioning is important for API evolution, as is ensuring forward and backward compatibility

(Section III-E). However, participants did not report using any tools (such as exist for Protocol Buffers) to help. We envision that if an API had a machine-checkable specification (Section IV-A), a tool could identify potentially incompatible changes, such as changing API parameters, return values, or method names. It could then prompt for version number and documentation changes. While this approach would not identify all incompatibilities, it could help API designers consider the implications of their changes on existing clients.

E. Checking Other Guidelines

Machine learning and natural language processing techniques may be able to evaluate interface uniformity (Section III-B1). Program analysis techniques may be able to warn developers about API methods that are not stateless (Section III-B3) and ensure that stateful endpoints are associated with reasonable HTTP verbs.

V. Related Work

In 2000, Fielding proposed Representational State Transfer (REST) [1] as a set of architectural constraints for web-based APIs. These included guidelines such as *statelessness* (state relevant to a request should be sent with the request, not stored on a server) and *uniformity*. Since 2000, developers have adapted Fielding's original REST guidelines for their own use. The result is a loose set of guidelines and practices that differ significantly from Fielding's vision. Our focus is on identifying current practice and the challenges therein.

Murphy et al. [42] studied public REST API design guidelines, constructing a list of topics authors can consider but finding inconsistent specific guidance. Later, Murphy et al. [17] interviewed 28 professionals regarding their training and design processes for APIs. While the interviews were not focused on REST API design, sixteen interviewees worked on REST APIs, and hence the study did include some results related to our paper. They also found that it can be unclear which fields to return for a given request, since one API may serve many different clients, and that excess data increases costs. Designers found it particularly challenging to discern which use cases were most important. Consistent with our findings, Murphy et al. found that many organizations develop their own API guidelines, but that, because these guidelines are not well enforced, it is challenging to apply them consistently.

Kotstein and Bogner [43] asked experts to rate the importance of 82 REST API design rules, finding that the experts gave high importance to 28 of the rules. We focus on guidelines that offered opportunities for tool development, since tools could make creating usable APIs easier. Our approach was also broader, focusing not only on given guidelines but also on practices that the experts used.

Wang et al. [44] studied relationships between REST API changes across versions and frequency of related questions on StackOverflow, finding that adding methods was associated with more questions than other kinds of changes. Our study focuses on guidelines and practices, not on evolution. Macvean et al. [16] studied how an API review process at Google improved APIs and the efficacy of API designers. Our work focuses on design decisions, not on processes.

Liu et al. [11] studied six months of high-severity bugs in Microsoft's Azure cloud computing environment. The most common causes of bugs were incorrect or missing fault detection and handling; data races; and inconsistent assumptions about data formats. 60% of the data-format incidents were due to interface problems, typically because of software updates that changed interfaces without considering existing clients.

OpenAPI [31] is the most prevalent approach for specifying REST API interfaces. Swagger [45] provides tools for creating, updating, and sharing OpenAPI definitions. JSON Schema [46] allows annotating and validating JSON documents against a schema. Sohan et al. [47] found in an experiment that usage examples improve users' success rates in using APIs relative to only providing specifications.

McLellan et al. [12] showed how API usability studies revealed shortcomings in one API's design and documentation and demonstrated that some API users find examples particularly helpful. Robillard [13] found learnability benefits of examples. Myers and Stylos [14] observed that usability problems have resulted in bugs and security problems, proposing user-centered design and evaluation methods to help designers make APIs more usable. Oliveira et al. [15] showed how latent security implications of API designs correlate negatively with developers' ability to identify security concerns in code.

Koci et al. [48] used log information to infer usability challenges in APIs; we elicited advice from experts directly.

VI. Conclusion and Future Work

We interviewed ten experienced REST API designers and implementors to understand opinions and practices about REST API design guidelines (RQ1) and to identify challenges in developing high-quality REST APIs (RQ2). We identified opportunities to improve the way REST APIs are designed and implemented. New tools could provide client developers with reliable documentation by making it easy for API designers to create specifications for their APIs that are consistent with the API implementations. The tools may be able to automatically generate appropriate examples and test cases from the specifications, reducing the documentation and testing burden on developers. Clients could rely on the accuracy of the specifications and examples, making development easier.

Existing REST API design guidelines are helpful but do not provide enough guidance regarding how to make APIs as usable as possible. Specific opportunities for community consensus include error handling, reporting, and propagation; pagination and identification of APIs that can return very large results; and clarity regarding HTTP verb usage.

These findings represent an opportunity for future research. Following guidelines has a cost, both in developer time and in effects on the API. We hope that future research will evaluate guidelines in a quantitative way to assess both cost and benefit. Establishing clearer API design guidelines may improve usability, allowing developers of client applications to learn new APIs more effectively and use them more successfully.

References

[1] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," PhD, University of California, Irvine, 2000. [Online]. Available: https://www.proquest.com/docview/304591392/abstract/CF90BE4588104415PQ/1

[2] M. Massé, *REST API Design Rulebook*. O'Reilly, 2011.

[3] H. Subramanian and P. Raj, *Hands-On RESTful API Design Patterns and Best Practices: Design, develop, and deploy highly adaptable, scalable, and secure RESTful web APIs*. Packt Publishing Ltd, 2019.

[4] M. Biehl, "RESTful API design: Best practices in api design with rest," 2016.

[5] Microsoft. (2022) RESTful web API design. [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design

[6] J. Au-Yeung and R. Donovan. (2020) Best practices for rest api design. [Online]. Available: https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/

[7] I. OWASP Foundation. (2023) Owasp api security project. [Online]. Available: https://owasp.org/www-project-api-security/

[8] Miley. (2020) 6 api design flaws you should absolutely avoid. [Online]. Available: https://www.dailycupoftech.com/6-api-design-flaws-you-should-absolutely-avoid/

[9] AkitaSoftware.com. (2023) Akita software. [Online]. Available: https://www.akitasoftware.com

[10] Google. (2023) Build RESTful APIs. [Online]. Available: https://docs.apigee.com

[11] H. Liu, S. Lu, M. Musuvathi, and S. Nath, "What bugs cause production cloud incidents?" in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 155–162. [Online]. Available: https://doi.org/10.1145/3317550.3321438

[12] S. McLellan, A. Roesler, J. Tempest, and C. Spinuzzi, "Building more usable APIs," *IEEE Software*, vol. 15, no. 3, pp. 78–86, 1998.

[13] M. P. Robillard, "What makes APIs hard to learn? answers from developers," *IEEE Software*, vol. 26, no. 6, pp. 27–34, 2009.

[14] B. A. Myers and J. Stylos, "Improving API usability," *Commun. ACM*, vol. 59, no. 6, p. 62–69, may 2016. [Online]. Available: https://doi.org/10.1145/2896587

[15] D. S. Oliveira, T. Lin, M. S. Rahman, R. Akefirad, D. Ellis, E. Perez, R. Bobhate, L. A. DeLong, J. Cappos, Y. Brun, and N. C. Ebner, "API blindspots: Why experienced developers write vulnerable code," in *Proceedings of the Fourteenth USENIX Conference on Usable Privacy and Security*, ser. SOUPS '18. USA: USENIX Association, 2018, p. 315–328.

[16] 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*, ser. CHI EA '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 849–858. [Online]. Available: https://doi.org/10.1145/2851581.2851602

[17] L. Murphy, M. B. Kery, O. Alliyu, A. Macvean, and B. A. Myers, "API designers in the field: Design practices and challenges for creating usable APIs," in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2018, pp. 249–258.

[18] R. Fielding, M. Nottingham, and J. Reschke. (2022) Rfc 9110, http semantics. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9110.html

[19] R. Fielding. (2008) REST APIs must be hypertext-driven. [Online]. Available: https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

[20] N. McDonald, S. Schoenebeck, and A. Forte, "Reliability and inter-rater reliability in qualitative research: Norms and guidelines for cscw and hci practice," *Proc. ACM Hum.-Comput. Interact.*, vol. 3, no. CSCW, nov 2019. [Online]. Available: https://doi.org/10.1145/3359174

[21] J. Nielsen and R. Molich, "Heuristic evaluation of user interfaces," in *SIGCHI conference on Human Factors in Computing Systems*, ser. CHI 1990, 1990.

[22] Uber. (2020) Introducing domain-oriented microservice architecture. [Online]. Available: https://www.uber.com/blog/microservice-architecture/

[23] Google. (2022) Protocol buffers. [Online]. Available: https://developers.google.com/protocol-buffers

[24] X. Zhu and H. A. Simon, "Learning mathematics from examples and by doing," *Cognition and instruction*, vol. 4, no. 3, pp. 137–166, 1987.

[25] A. Renkl, "Instruction based on examples," in *Handbook of research on learning and instruction*. Routledge, 2011, pp. 286–309.

[26] M. Green and M. Smith, "Developers are not the enemy!: The need for usable security apis," *IEEE Security & Privacy*, vol. 14, no. 5, pp. 40–46, 2016.

[27] A. Parecki. (2022) Oauth 2.0. [Online]. Available: https://oauth.net

[28] Amazon. (2022) Amazon cognito. [Online]. Available: https://aws.amazon.com/cognito/

[29] O. P. A. contributors. (2022) Open policy agent. [Online]. Available: https://www.openpolicyagent.org

[30] G. Fink and M. Bishop, "Property-based testing: a new approach to testing for assurance," *ACM SIGSOFT Software Engineering Notes*, vol. 22, no. 4, pp. 74–80, 1997.

[31] T. O. Initiative. (2022) The openapi initiative. [Online]. Available: https://www.openapis.org

[32] D. Dygalo. (2021) Schemathesis: property-based testing for API schemas. [Online]. Available: https://dygalo.dev/blog/schemathesis-property-based-testing-for-api-schemas/

[33] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Checking security properties of cloud service REST APIs," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 387–397.

[34] R. Waller. (2022) Openapi validator. [Online]. Available: https://github.com/openapi-library/OpenAPIValidators/

[35] M. Corporation. (2022) Typescript. [Online]. Available: https://www.typescriptlang.org

[36] G. Corporation. (2022) PyType. [Online]. Available: https://github.com/google/pytype

[37] S. Corporation. (2022) Sorbet. [Online]. Available: https://sorbet.org

[38] M. Das, S. Lerner, and M. Seigle, "Esp: Path-sensitive program verification in polynomial time," *SIGPLAN Not.*, vol. 37, no. 5, p. 57–68, may 2002. [Online]. Available: https://doi.org/10.1145/543552.512538

[39] A. Head, E. L. Glassman, B. Hartmann, and M. A. Hearst, *Interactive Extraction of Examples from Existing Code*. New York, NY, USA: Association for Computing Machinery, 2018, p. 1–12. [Online]. Available: https://doi.org/10.1145/3173574.3173659

[40] J. Conners, S. Derbidge, C. Devenport, N. Farnsworth, K. Gates, S. Lambert, C. McClain, P. Nichols, and D. Zappala, "Let's authenticate: Automated certificates for user authentication," in *Network and Distributed Systems Security (NDSS) Symposium*, 2022.

[41] J. Parker, N. Vazou, and M. Hicks, "Lweb: Information flow security for multi-tier web applications," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, jan 2019. [Online]. Available: https://doi.org/10.1145/3290388

[42] L. Murphy, T. Alliyu, M. B. Kery, A. Macvean, and B. A. Myers, "Preliminary analysis of REST API style guidelines," in *PLATEAU 2017*, 2017.

[43] S. Kotstein and J. Bogner, "Which restful api design rules are important and how do they improve software quality? a delphi study with industry experts," in *Service-Oriented Computing*, J. Barzen, Ed. Cham: Springer International Publishing, 2021, pp. 154–173.

[44] S. Wang, I. Keivanloo, and Y. Zou, "How do developers react to RESTful api evolution?" in *International Conference on Service-Oriented Computing*. Springer, 2014, pp. 245–259.

[45] S. Software. (2022) Swagger. [Online]. Available: https://swagger.io

[46] J. Schema. (2022) Json schema. [Online]. Available: https://json-schema.org

[47] S. M. Sohan, F. Maurer, C. Anslow, and M. P. Robillard, "A study of the effectiveness of usage examples in rest api documentation," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2017, pp. 53–61.

[48] R. Koçi, X. Franch, P. Jovanovic, and A. Abelló, "A data-driven approach to measure the usability of web apis," in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug. 2020, pp. 64–71.