UNIVERSITY OF CALGARY

Automated Example Oriented REST API Documenation

by

S M Sohan

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF DOCTOR OF PHILOSOPHY

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA

September, 2017

UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Automated Example Oriented REST API Documenation" submitted by S M Sohan in partial fulfillment of the requirements for the degree of DOCTOR OF PHILOSOPHY.

_____

Dr. Frank Maurer
Supervisor
Department of Computer Science
University of Calgary
Canada

_____

Dr. Mea Wang
Supervisory Committee Member
Department of Computer Science
University of Calgary
Canada

_____

Dr. Craig Anslow
Supervisory Committee Member
School of Eng. and Computer
Science
Victoria University of Wellington
New Zealand

_____

Date

Abstract

API documentation presents both a problem and an opportunity for API usability. Most developers learn to use an API based on its documentation and at the same time face obstacles using the APIs due to lack of necessary information in the documentation [**?**]. REST APIs provide interconnectivity between applications over HTTP. Documentation of a REST API is a key information source for software developers that use these types of API. As a REST API evolves, the documentation needs to be maintained alongside. This implies that REST API developers also need to efficiently generate and maintain the documentation of their APIs with the qualities that make the API usable. This introduces the problem of interest in this research: how to document REST APIs?

This is a problem because there is a lack of scientifically established guidelines and automated tool support for REST API documentation with usage examples. Existing local API documentation techniques and tools are not readily usable for the documentation of REST APIs because REST APIs have HTTP protocol specific parameters, and API clients are agonistic to the programming languages used to implement the APIs. Existing REST API documentation techniques rely on custom API definition languages that require manual effort to generate and maintain REST API documentation as a separate artifact as the API evolves.

In this reasearch, I proposed and evaluated a novel technique to solve REST API documentaiton requirements. I identified the requirements by studying the existing literature and the current state of REST API documentation in practice. The technique involves the interception of example REST API calls using an HTTP proxy server to auto-generate an accurate and updated REST API documentation.

Practioners can use the presented technique to automatically generate and maintain an updated documentation of REST APIs including the API structure and usage examples from their API test code. Researchers can apply the novel concept of using interception to automatically document API usage examples from test scripts to other forms of APIs.

Acknowledgements

I am grateful to Shahana, my wife, for supporting me and the family while I took the time away to work on this research.

I am thankful to my supervisor Dr. Frank Maurer for giving me the opportunity and valuable guidance to pursue this research. I am also thankful to Dr. Craig Anslow for his continuous encouragement and the countless number of hours that he spent to provide feedback about my work.

I am grateful to Cisco for allowing me to pursue this research alongside my job. I am grateful to the study participants in my case studies for sharing their time with me on the experiments.

Dedication

To my dear wife.

Contents

List of Tables

List of Figures

List of Symbols, Abbreviations and Nomenclature

Symbol	Definition
U of C	University of Calgary

Chapter 1

Introduction

Application Programming Interfaces, commonly known as APIs, are used to express a software component in terms of its operations, inputs, outputs, and their types[1]. Robillard defines an API as follows: An API is the interface to implement functionality that developers can access to perform various tasks [1, 2]. APIs enable multiple software components to interact with each other.

REST APIs are a type of APIs that are used to integrate software components using web technologies. Fielding defined Representational State Transfer or REST as an architectural style for developing distributed hypermedia systems [3]. For example, GitHub, an online code repostory has a REST API that can be used by third-party applications to integrate with GitHub features. The GitHub REST API[2] has a resource called `Repository` to denote a code repository that can be identified by the URL `/user/repos`. To create a new `Repository` for a user, the GitHub API can be invoked via HTTP *POST* at `/user/repos` using the following JSON representation of a `Repository`:

```
{
  "name": "Hello-World",
  "description": "This is your first repository",
  "homepage": "https://github.com",
  "private": false,
  "has_issues": true,
  "has_wiki": true,
  "has_downloads": true
}
```

---

[1]https://en.wikipedia.org/wiki/Application_programming_interface
[2]https://developer.github.com/v3/repos/#create

In today's world of technology, REST APIs have become ubiquitous and the primary choice for integrating Internet enabled applications due to its simplicity and similarity with HTTP [4]. For example, a real estate listing website uses a REST API to collect "walk score" and another REST API to show a map view of a property listing. By incorporating the map and walk score using REST APIs, the real estate listng site provides important information to their users. Most REST APIs, including these two examples, are often documented manually or using custom implemented tools specific to the APIs that are not publicly available. This requires effort to generate and maintain the documentation of REST APIs over time since there is a lack of reusable tool support.

Previous research in the area of API usability mostly focused on local APIs such as Java libraries. Researchers identified the documentation of APIs as both the primary source of information as well as the key obstacle for API usability [2]. Hence the quality of the API documentation plays an important role in API usability. To this regard, researchers have identified the qualities of "good API documentation" as follows: complete, correct, includes thorough explanations and code examples, provides consistent presentation and organization [2, 5]. Today, there are several tools such as Javadoc[3], UsETeC [6], Jadeite [7], APIMiner [8], Roast [9] that can be used to document local APIs with the aforementioned qualities .

While there are overlaps between the documentation requirements of local and REST APIs, there are significant requirements that are unique to each. The documentation of most local APIs contain a structrual definition of the API classses and methods. API usage examples for local APIs are documented using the same programming language of the API. On the other hand, REST APIs are defined using HTTP specific parameters such as API endpoints, query parameters, HTTP headers and body. REST API clients can use any programming language agnostic to the API implementation language. Due to such differences between local and REST APIs, the existing technique and tools for local API documentation cannot be readily used to generate REST API documentation.

---

[3]`http://www.oracle.com/technetwork/java/javase/documentation/`
`index-jsp-135444.html`
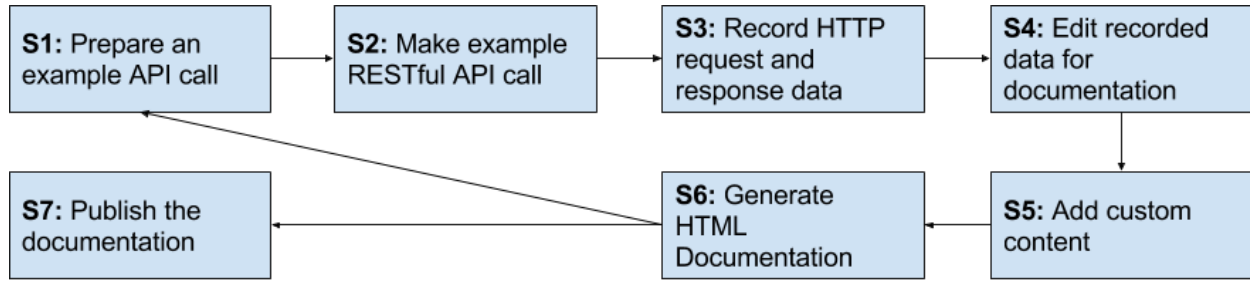
Figure 1.1: Manual REST API Documentation Steps

Currently, the process of documenting REST APIs are largely manual. Conceptually, the high level steps involved in the manual process can be identified as shown in Figure 1.1. These steps S1-S7 can be described as follows: S1 - API developer prepares an example API with the required HTTP parameters and request body, S2 - API developer uses a REST API client to make the example API call, S3 - the API developer records the HTTP data , S4 - API developer then edits the recorded data so that only relevant content is selected for documentation, S5 - API developers use any custom content to describe the API example, S6 - API developers combine the custom content with the edited content from the HTTP traffic into HTML to publish to the web, and S7 - API developer adds custom overview information to explain API concepts and general rules, and publishes the final documentation so other developers can learn the API. The steps S1-S6 are repeated for every API action that needs to be documented, and S1-S7 need to repeated for every version of the REST API as it evolves. As depicted here, this manual process can be both time consuming and error-prone. In my research, I have aimed to automate the repeating manual efforts involved in the aforementioned workflow.

This thesis is ogranized as follows. In Chapter 2, I discuss a case study to summarize the state of existing literature and industry practices related to the versioning, documentation, and change communication patterns of evolving Web APIs. The results are based on a study of nine industrial APIs selected based on their diverse application domain, maturity, and popularity including both open and closed source implementations. The primary findings from this study are as follows: Web APIs evolve frequently, often several times a week, including both compatible and incom-

patible API changes, using manual or bespoke methods to version, document, and communicate the changes as they evolve. I found a lack of reusable techniques in the existing literature to automatically generate and maintain the documentation of multi-version evolving REST APIs with usage examples. I persued further research to solve this problem as discussed in the following paragraphs.

In Chapter 3, I present the following list of REST API documentation requirements by studying the literature and applying the findings from Chapter 2: 1) automated, 2) example based, 3) executable, 4) version aware, 5) customizable, 6) reusable, and 7) collaborative. I discuss the available REST API documentation techniques that require repeatitive manual efforts to generate and maintain custom formatted API descriptions that fail to satisfy these requirements. I present, SpyREST, a novel technique to intercept example REST API calls using an HTTP proxy server to collect and autmatically synthesize API traffic to generate and update API documentation to satisfy the aforementioned requirements. The primary advantage of the presented technique is that the documentation is generated from executable code. As a result, a continuously updated REST API documentation with realistic usage examples is achievable with a reduced manual effort

In Chapter 4, I present a prototype implementation of the aformentioned technique to demonstrate a reusable tool and compare it against the existing REST API documentation tools that rely on custom API description languages. I also present an example where automated test code can be used to generate and maintain REST API documentation using the developed tool.

I discuss an industrial evaluation of the proposed technique and the tool in Chapter 5. The evaluation is carried out based on the data collected from an eighteen month of production use of SpyREST at Cisco, my current employer. From the study I found that it is feasible to use the proposed interception technique and automated tests to document evolving REST APIs satisfying the previously preseted REST API documentation requirements. Automated always-updated REST API documentation was found to help establish a quick feedback cycle among the stakeholders. I also discuss several refinements and limitations of the proposed technique based on the lessons

4

learned from this case study.

In Chapter 6, I discuss a controlled study performed with 26 professional software engineers to undestand the impact of usage examples in API documentation on REST API client developers. I mapped the participants into two groups and in API troduced usage examples as the control variable. I enhanced an existing REST API documentation with new usage examples. I gave the enhanced API documentation to participants from one of the study groups. I found that participants using the original documentation without usage examples faced common obstacles related to using appropriate data types, data formats, and required elements in the HTTP headers and request body. I observed that providing usage examples with data taken from API test code reduces the obstacles and improves API client developer success rate, reduces required trial attempts, and improves satisfaction ratings. The findings from this study provides an empirical evidence that by using the proposed technique of generating REST API documentation from API test code can help API client developers to achive a higher level of success with a reduced effort.

In summary, this thesis makes the following key contributions to the body of research in software engineering:

1. A list of REST API documentation requirements derived from the existing literature and the current state of industry practices to be used as a guideline by practitioners and researchers.

2. SpyREST, a novel technique and a reusable tool that uses interception to automatically generate and maintain evolving REST API documentation with usage examples from executable code.

3. An industrial evaluation of SpyREST to demonstrate the feasibility of the proposed technique that practitioners can follow to improve their REST API documentation experience.

4. An empricial evidence showing the common obstacles faced by REST API client

developers that can be reduced with usage examples to help the prioritization of REST API documentation efforts.

1.1 References

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

[2] M. Robillard, "What makes APIs hard to learn? the answers of developers," *Software, IEEE*, vol. PP, no. 99, pp. 1–1, 2011.

[3] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[4] J. Mangler, P. P. Beran, and E. Schikuta, "On the origin of services using RIDDL for description, evolution and composition of RESTful services," in *Proc. of IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, 2010, pp. 505–508.

[5] B. Myers, S. 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," *J. Organ. End User Comput.*, vol. 22, no. 1, pp. 23–51, 2010.

[6] Z. Zhu, Y. Zou, B. Xie, Y. Jin, Z. Lin, and L. Zhang, "Mining API usage examples from test code," in *Proc. of 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME),*. IEEE, 2014, pp. 301–310.

[7] J. Stylos, B. Myers, and Z. Yang, "Jadeite: Improving API documentation using usage information," in *Proc. of Extended Abstracts on Human Factors in Computing Systems*, ser. (CHI EA). ACM, 2009, pp. 4429–4434.

[8] J. E. Montandon, H. Borges, D. Felix, and M. T. Valente, "Documenting APIs with examples: Lessons learned with the APIMiner platform," in *Proc. of IEEE Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 401–408.

[9] D. Hoffman and P. Strooper, "API documentation with executable examples," *J. Syst. Softw.*, vol. 66, no. 2, pp. 143–156, May 2003.

Chapter 2

A Case Study of Web API Evolution

2.1 Abstract

When applications are integrated using web APIs, changes on a web API may break the dependent applications. This problem exists because old versions of the APIs may no longer be supported, a lack of adequate documentation to upgrade to a newer version, and insufficient communication of changes. In this paper we conducted a case study of evolving Web APIs to investigate what changes are made between versions and how the changes are documented and communicated to the API users. The findings are a list of recommendations for practitioners and researchers based on API change profiles, versioning, documentation and communication approaches that are observed in practice. This study will help inform developers of evolving Web APIs to make decision about versioning, documentation and communication methods.

2.2 Introduction

Web APIs are used as a key interconnectivity mechanism to access software services over the Internet. Interconnected applications can provide better service at a lower cost to their users. For example, a business search portal can display the businesses near a user's current location on a map by using the Google Maps API. The actual implementation of Web APIs can follow various protocols, such as SOAP[1] and RESTful[2] web services. In this paper, the term Web API is used to describe any protocol unless otherwise specified.

Web APIs connect independently maintained applications. As a result, when a Web API evolves, all the integrated applications using the Web API may not be able to evolve on the same

---

[1] http://www.w3.org/TR/soap/
[2] http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

schedule. For example, in the case of the aforementioned business search portal, if the Google Maps API changes, it may also require the portal to change. This introduces unique challenges to the evolution of Web APIs compared because the evolution is out of the control of the API users. For example, API users of a local API can use a copy of an older version, but for evolving Web APIs, API users are forced to upgrade if older versions are no longer supported. Documentation of evolving Web APIs is an important topic for case study because of the unique challenges it presents. Web APIs are defined by their HTTP interface, including both HTTP headers as well as request and response data formats. As a result, for an evolving Web API, its documentation needs to indicate changes in any of these fields, unlike the local APIs, where typically no HTTP specific information is involved.

The primary goal of this research is to investigate the challenges that are currently faced by developers and users of evolving Web APIs so that future research can be performed to address the problems. To this regard, we analyzed existing literature on the topic of Web API evolution. We analyzed the evolution of multiple Web APIs to understand the current industry trends and challenges. The juxtaposition of current industry practices with the literature reveals insights that are otherwise unseen when viewed from a single perspective. We use the differences between the literature and industry practices to identify unresolved research problems.

In this case study, multiple Web APIs representing a variety of application areas are selected, analyzed and compared to gain a broader perspective about current industry practices on Web API versioning, documentation and communication of changes. The result of this analysis is a summary of different approaches to Web API evolution, and a list of recommendations for developers of evolving Web APIs.

The remainder of this paper is organized as follows: the next section is related work on Web API evolution. Then, we discuss the case study of multiple Web API evolution. In the following section, we list our findings, followed by a discussion about the findings. Finally, we conclude with the contributions of this work.

2.3 Related Work

We look at related work in the following areas: Empirical Research on Web API evolution, techniques to evolve Web APIs and, tool support for evolving Web APIs.

2.3.1 Empirical Research on Web API Evolution

Several case study papers were published on Web API Evolution. Maleshkova et al. analyzed multiple Web APIs to identify the different approaches related to Web API descriptions, data formats, protocols, reusability, granularity, and authentication [1]. They found that a lack of a standard format to document Web APIs and manual documentation led to API under-specification causing confusions about how to use the APIs for different use cases. They identified a need for automated approaches.

Fokaefs et al. performed an empirical study on Web API evolution involving multiple SOAP web services, Amazon EC2, FedEx Rate, Bing, Paypal, and Fedex Pack., and provided an evolution profile showing the percentage of changes between two consecutive versions of the Web APIs [2]. The same Web APIs were studied by Romano et at., but they provided an alternate evolution profile of the Web APIs. Instead of using percentage of change, they profiled the number of operations, parts and XML file elements from WSDL files for each version of the Web APIs [3]. These change profiles point to the ever changing nature of Web APIs.

Jun et al. performed case studies on the evolution of Web APIs and compared the evolution of Web APIs against local library APIs [4]. For studying Web APIs, a selection criteria was provided as follows: Web APIs with large number of clients, covering different application areas, owned by different companies in different countries, and well documented API reference and migration guides. They extracted 16 patterns to describe different types of changes on the Web APIs as follows: add or remove parameter, change type of return value, delete method, rename method, rename parameter, change format of parameter, change format of return value, change XML tag, combine methods, split method, expose data, unsupport request method, change default value of

parameter, chance default value of parameter, change upper bound of parameter, restrict access to API. They also presented six challenges specific to Web API evolution that are not found with library APIs such as: transformation between JSON and XML, M to N Mapping, delete method, authorization protocol change, rate limit, and authorization of API access.

Espinha et al. looked at the growing pains with Web API evolution from the perspective of Web API users by conducting interviews [5]. The interviewees were selected based on the activity of their open-source projects that used Web APIs. They found that when a Web API evolves, maintaining the integration with a Web API took more effort than integrating it the first time since maintenance is carried out over a longer period of time. This makes the evolution of Web APIs an important subject for a case study.

We distinguish our case study from these case studies as follows: we provide up-to-date information and include a diverse set of Web APIs. Our analysis uses new data sources such as API user forums, and question and answer sites. We focused this study on different approaches to versioning, documentation, and communicating changes as an API evolves.

2.3.2 Techniques to Evolve Web APIs

Kaminski et al. presented a design technique called "Chain of Adapters" to implement evolving Web APIs [6]. Using this approach, the source code of a Web API can evolve from a version to the next by introducing an adapter on the original version to implement any required change for new versions. This technique allows for multiple versions to be deployed concurrently since older versions are left unchanged.

Leskey suggests following a consistent and meaningful naming scheme to identify different versions of APIs [7]. Such a naming scheme can be used to identify if two versions of an API are compatible. Given the URL scheme, Leskey suggests a self documenting Web API, one that exposes API endpoints to describe the details for each version.

Treiber et al. showed a conceptual model to capture the changes of a Web API [8]. A graphical representation to display the evolution of SOAP Web Services is presented by Aversano et al.

where the differences between versions are highlighted based on their meta information found in the WSDL files [9].

Some papers focused on automatic migration of Web API clients to newer versions. For example, Wilde proposed a conceptual framework where WSDL can be extended so a client can be compatible with potential Web API changes [10]. A similar approach is proposed by Fang et al. where a fixed set of new XML nodes are used instead of arbitrary extensions [11]. Juric et al. proposed a number of versioning related extensions to WSDL that can be applied to version SOAP web services in multiple levels of granularity such as, service level and operation level [12]. Zuo et al. developed a formal XML based change specification format to describe the evolution of SOAP Web services [13].

For RESTful Web APIs, Mangler et al. introduced an XML based language called RIDDL [14]. RIDDL allows incremental composition of Web API documentation for different versions by adding a changelog to the documentation of an old version.

VRESCo demonstrates an approach where a router between the Web API client and server is used to automatically route Web API calls to new API versions. The router routes API calls to the desired version based on free form version names such as: latest, stable, and fixed version [15]. Meng et al. compared different approaches for evolving Web APIs across multiple criteria: granularity of evolution, terminal of evolution, type of evolution, scalability, and maintainability [16]. They identified an aspect oriented approach, where an intermediate tier between a Web API and its clients is used similar to VRESCo, can reduce coupling between the two.

Existing techniques to evolve Web APIs focused on finding different techniques for implementing evolving SOAP based Web APIs that cannot be readily applied to RESTful Web APIs. From our case study, we identified future research opportunities on techniques to solve the problems Web API evolution that are also applicable to RESTful APIs.

2.3.3 Tool Support for Evolving Web APIs

Analyzing the available tool support for evolving Web APIs helped us identify opportunities for future research in this area. WSDarwin introduces a layer of adapters on both the Web API and its client to automatically migrate Web API clients to use a newer version of an API [17]. WSDarwin uses WSDL description of the SOAP service to automatically infer the changes between versions and uses preset default values for newly added objects to adapt the client request. VTracker, predecessor of WSDarwin, produces a changelog between versions of WSDL files [2]. WSDLDiff extends this by incorporating the semantic meaning of WSDL with an XML Schema Definition into their changelog computation [3].

To identify if a new version of a Web API has any impact on a specific API client, Zou et al. show a technique to generate client specific change logs [18]. A customized per API client change log is generated by filtering the change log based on recorded API usage data so that only relevant API changes can be communicated effectively to each user.

Existing research on tool support for RESTful Web API evolution is rather limited. hRESTS is an HTML-based specification for describing RESTful Web APIs that can be used by computers to auto-generate client code [19]. RESTdesc is another specification for describing RESTful Web API documentation where APIs are described using pre and post conditions [20]. However, the input to both hRESTS and RESTDesc require manual effort to produce the desired specification.

In the literature, we found multiple tools exist that rely on WSDL files. WSDL solutions have limited applicability for RESTful Web APIs since WSDL files cannot be used to fully describe RESTful APIs. Based on our case study, we will discuss future research problems on tool development to support the challenges with RESTful Web API evolution.

2.4 Case Study

When companies evolve their Web APIs in practice the new changes can often break applications that rely on these APIs. The aim of this case study is to investigate current industry practices on

Web API Evolution to answer the following research questions.

2.4.1 Research Questions

RQ1 Versioning: How are Web APIs versioned as they evolve?

RQ2 Documentation: How are Web API changes documented as they evolve?

RQ3 Communication: How are Web API changes communicated as they evolve?

2.4.2 Selection Strategy

To gain a broad perspective, we selected evolving Web APIs to represent each of the following categories as shown in Table 2.1.

Industry - APIs representing different domains: social, business, and productivity.

Popularity - APIs with different number of users.

Maturity - Newer vs. established Web APIs.

Source code - Open source vs. closed source.

2.4.3 Research Method - Qualitative Analysis

The APIs in this case study are proprietary and open-source evolving Web APIs. For each API, we collected publicly available data from the following sources: API home page, formal API reference, and user forums and question-answer sites. A homepage provides a general introduction to the Web API. The formal API reference described the different elements of the API. The user forums and question-answer sites are used to communicate news and collect feedback about the Web APIs, for example StackOverflow.com.

The first author of this paper read the the API homepages and followed relevant hyperlinks to find answers to RQ1. A summary of the strategies were presented to the co-authors, which was used to categorize Web API versioning strategies into groups. Web API homepages contain changelogs and links to API references which were used to find answers to RQ2.

Table 2.1: Case Study - Representative Sample of Web APIs.

Web API	URL	Categories				Num of Releases	Dates
		Industry	Popularity	Maturity	Code		
Facebook Platform API	https://developers.facebook.com/docs/apps/versions	Social	High	Est	Closed	11	01/2013-07/2014
Twitter REST API	https://dev.twitter.com/docs/api	Social	High	Est	Closed	2	Not Available
Wordpress REST API	http://wp-api.org/	Business	Growing	New	Open	8	08/2013-06/2014
Salesforce & Chatter REST API	https://www.salesforce.com/us/developer/docs/api_rest/	Business	Popular	Est	Closed	4	01/2013-08/2014
Google Calendar API	https://developers.google.com/google-apps/calendar/	Business	Popular	Est	Closed	2	Not Available
Stripe Payment Processor API	https://stripe.com/docs/api	Business	Growing	New	Closed	30	09/2011-10/2014
Github API	https://developer.github.com/	Productivity	Growing	New	Closed	59	09/2012-10/2014
Google Maps API	https://developers.google.com/maps/web/	Productivity	Popular	Est	Closed	102	01/2011-09/2014
OpenStreetMap API	http://wiki.openstreetmap.org/wiki/API	Productivity	Growing	New	Open	1	Not Available

We used codes to annotate the API changelogs to identify the different types of API changes. The coding scheme started with a set of codes based on the codes from Jun et al. [4]. New codes emerged from our analysis as the existing codes did not describe an API change scenario. An overview of the used codes is provided in Table 2.2. The codes were applied to the different API elements as described through the following example Web API description:

"A Web API to post a comment on a photo."

Endpoint - URLs to access the the API. **Resource** - An input or output object. For example, comment, photo, etc. **Method** - An API operation on a resource. e.g. post a comment. **Field** - An attribute of the resource. e.g. timestamps of a comment. **FieldValue** - Possible value for a field. For example: a timestamp value of January 1, 2015. **Error** - Error cases. e.g. not found error. **Authentication** - Authentication methods. e.g. HTTP Basic authentication.

These codes were applied to the text from the Web API homepages and API changelogs for multiple versions to produce a change profile. The first and second authors of this paper arrived at the same conclusion when they independently applied the codes on 18 snippets of changelogs randomly sampled from the studied Web APIs. The coding process was manual since the data comprises of unstructured text that are not suitable for automated tagging. Here is an example of a change log from WordPress API with the codes:

"Add post meta endpoints (**AddEndpoint**). Creating, reading, updating and deleting (**AddMethod * 4**) post meta (**AddResource**) is now possible by using `the/posts/<id>/meta` endpoints."

We analyzed the API user forums and newsboards to find answers to RQ3. Messages in these sources were filtered to focus on the evolution of Web APIs. These data sources provided the perspectives of both the developers and users of Web APIs. To find evolution related messages, based on trial search runs, full text search features on the forums were used to search for these keywords: version, release, migration, breaking, deprecate, change.

16

Table 2.2: List of Codes

Category	Description
Add<APIElements>	New API elements introduced: Adds a new resource, field, field value, endpoint or method
Remove<APIElements>	API elements completely removed: Removes a resource, field, field value, endpoint or method
Change<APIElements>	Existing API element is replaced by another: Replaces or renames a resource, field, field value, endpoint or method
ErrorConditionChange	Changes the possible error conditions: New, changed or removed error response
ChangeFieldDataType	Changes the data type of a field: Single item to an array or vice versa, number to strings or vice versa, required to optional or vice versa
BehaviorChange	Changes the outcome or fixes bug: Uses new formula for computing API return values
ChangeAuthentication	Changes how API clients are authenticated: New, changed or deprecated API authentication methods
ChangeAuthorization	Changes who can access which API elements: Add, remove or modify permissions of API user

2.5   Findings

In this section we present our findings for RQ1-3.

2.5.1   RQ1 Versioning

Patterns of Changes

Based on our coding scheme, we identified the following new change patterns to describe changes between versions of Web APIs in addition to the 16 patterns described by Jun et al. [4] as follows:

Move API elements - We found 114 occurrences of element moves, where API elements are moved under new hierarchy or renamed. For example, Stripe API moved two objects under a new API element as found in the following change log from version 2013-08-13:

> "Remove fee and fee_details properties on charge and transfer objects.
> Instead, fee information is now stored on the corresponding balance trans-
> action. "

Rename API elements - We found that 31 API elements were renamed for consistency. The following example from the Facebook API shows a rename [3]:

> "Facebook will be renaming the adgroup_status value ADGROUP_PAUSED
> TO PAUSED to be consistent with the rest of the object's APIs. "

Behavior change - We have found 247 examples where Web APIs changed the resultant data while keeping the API interfaces intact. These are commonly triggered by bug fixes. For example, Wordpress REST API 1.1 changed as follows [4]:

> "Correct password-protected post handling. Password-protected posts
> could previously be exposed to all users, however could also have broken
> behavior with excerpts. Password-protected posts are now hidden to unau-
> thenticated users.. "

---

[3]https://developers.facebook.com/docs/apps/migrations/ads-api-changes-2014-07-02
[4]http://make.wordpress.org/core/2014/06/23/json-rest-api-version-1-1/

Another example that demonstrates a behavior change can be found from the 2013-10-29 version of Stripe API [5]:

> "When we apply a $Y coupon to a $X dollar invoice, we are no longer applying the remainder of the coupon to the account balance if $Y > X$. Applications of coupons to $0 invoices will no longer count as a redemption of the coupon."

Post condition change - In the studied Web APIs we found examples where the immediate result of an API call remained unchanged but new post conditions are imposed requiring additional work for the API users. The following change on Facebook platform API illustrates this [6]:

> "Starting May 13th, 2014, developers that accept payments will be required to subscribe to and honor Realtime Updates to ensure order fulfillment and appropriate handling of disputes from all payers. If you do not subscribe to and honor these updates, Facebook reserves the right, under our Developer Payment Terms to withhold payouts and/or stop your app from accepting payments. "

HTTP header change - Web APIs also change the custom HTTP headers that are used as meta data. For example, the following shows a change when the WordPress Web API introduced two new headers to describe the pagination status of an API call [7]:

> "Send X-WP-Total and X-WP-TotalPages headers for information on post/pagination counts"

Similarly, the Github API changed their custom headers about pagination in the following example [8]:

---

[5] https://stripe.com/docs/upgrades#2014-01-31
[6] https://developers.facebook.com/docs/apps/migrations/payments-disputes-realtime-updates
[7] http://make.wordpress.org/core/2013/09/12/json-rest-api-version-0-5/
[8] https://developer.github.com/v3/versions/

"No longer using the X-Next or X-Last headers. Pagination info is returned in the Link header instead."

Error condition change - Web APIs change the error conditions that may require the users to adapt their integration. For example, in Salesforce API Version 29 the error codes were changed [9]:

"...different ExceptionCode and StatusCode values are now returned for some error conditions when saving user records. We strongly recommend that you test your code and modify it accordingly if your clients rely on specific ExceptionCode and StatusCode values being returned... "

These extended list of change patterns need to be addressed by future work on finding techniques and developing tool support for evolving Web APIs.

Version Identifiers

We analyzed different approaches to identify the versions of Web APIs to understand if the version identifiers can be used to infer information about their evolution. We can categorize the Web APIs into these groups: **Numbered identifiers** - Contiguous integers are used to identify new versions by Salesforce and GitHub Web APIs. Such naming scheme clearly communicates the sequence between versions but cannot be used to infer compatibility between versions.

Timestamped identifiers - Stripe Web API uses date as a version identifier. While timestamped versions provide the time frame of a version, they do not provide any compatibility information between versions.

Identifiers with major and minor version numbers - Most of the Web APIs studied use major and minor version numbers to identify a version. Facebook, Twitter, WordPress, Google Calendar, Google Maps and OpenStreetMap follow this approach. However, an explanation of the major and minor version numbers aren't provided and cannot be used to infer compatibility information.

---

[9]https://na1.salesforce.com/help/pdfs/en/salesforce_winter14_release_notes. pdf#rn_186_api_objects

Also, web APIs often evolve with breaking changes without issuing a new version identifier. For example, Facebook, Google Calendar and GitHub APIs often receive breaking changes without using a new version identifier.

Upgrade Mode

From our case studies, we found the Web APIs followed several different approaches to versioning.

Single Version - Web APIs that force the API clients to migrate either because only one version is available or deprecated older versions will be removed. These include Facebook REST API (90 days), Twitter REST API (no specific time frame), Github API (no specific time frame), OpenStreetMap API (no specific time frame), Google Calendar (6 months), Google Maps (no specific time frame).

Multiple Versions - Multiple versions of the Web API are supported for longer duration. These include Wordpress REST API, Salesforce API, Stripe API.

Both of these categories, single and multiple available versions, have their advantages and disadvantages. Forcing API clients to use a single stable version makes it easy to maintain the Web APIs at the expense of scheduling flexibility for API users. These APIs keep a deprecated version for a limited period of time to provide a buffer time for the API users. For example, Google Calendar API is supporting its deprecated version for 6 months.

Maintaining multiple versions of the Web APIs provide more scheduling flexibility to the API clients but requires more effort from the API publisher. As a result, not all the older versions are maintained. When API clients lag far behind the latest version, migrating to the latest version becomes harder. For example, here is an excerpt of a comment from a Stripe Web API client developer [10]:

"...I just learned that we're about 14 versions behind and would like to work on upgrading but there are lots of breaking changes..."

While both the forced and on demand modes are found in practice, future work needs to be

---

[10]https://groups.google.com/a/lists.stripe.com/forum/#!searchin/api-discuss/version/api-discuss/x8TM-yHnhYQ/y9JJKytSiUAJ
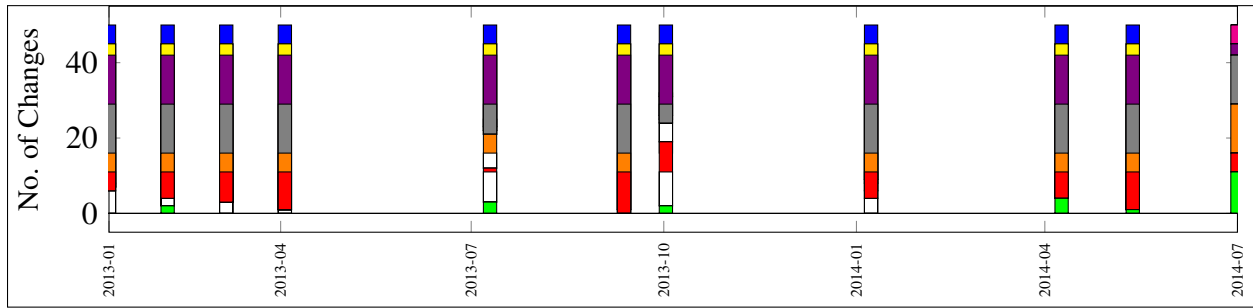
(a) Facebook API Changes (11 Releases)
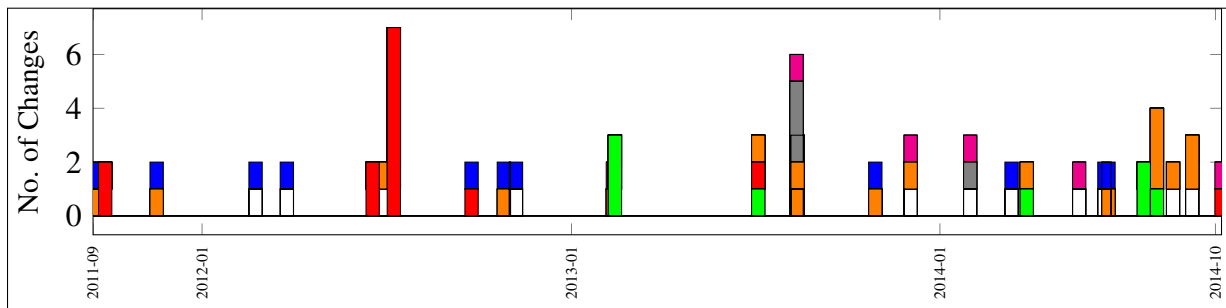
(b) Stripe API Changes (30 Releases)
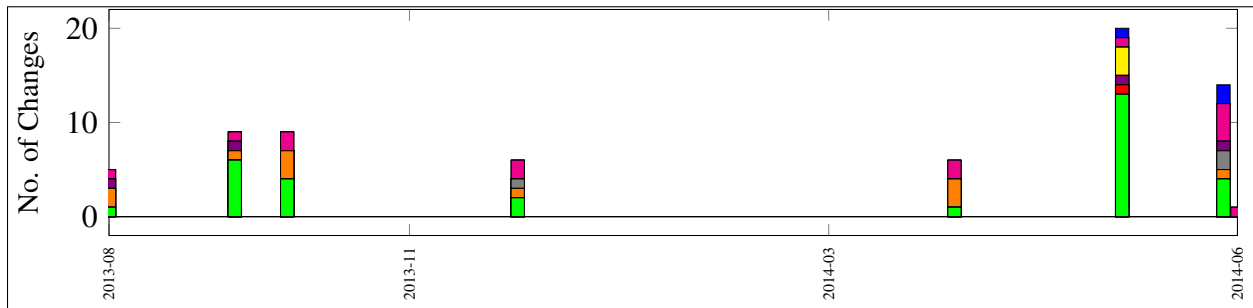
(c) WordPress REST API Changes (8 Releases)
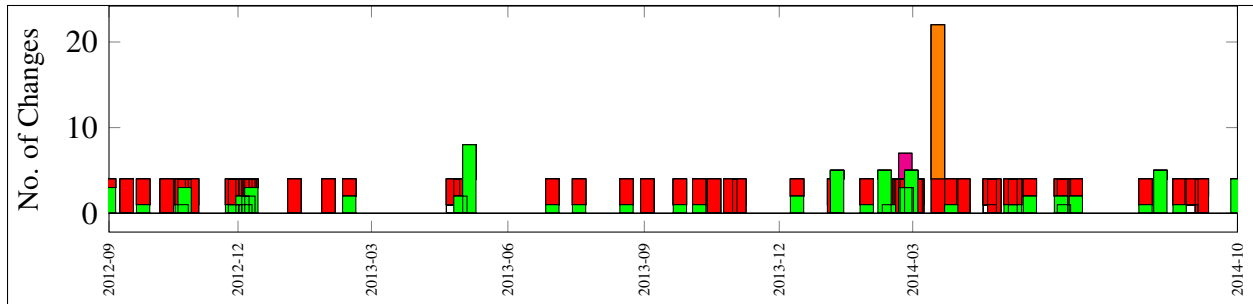
(d) GitHub API Changes (59 Releases)

22
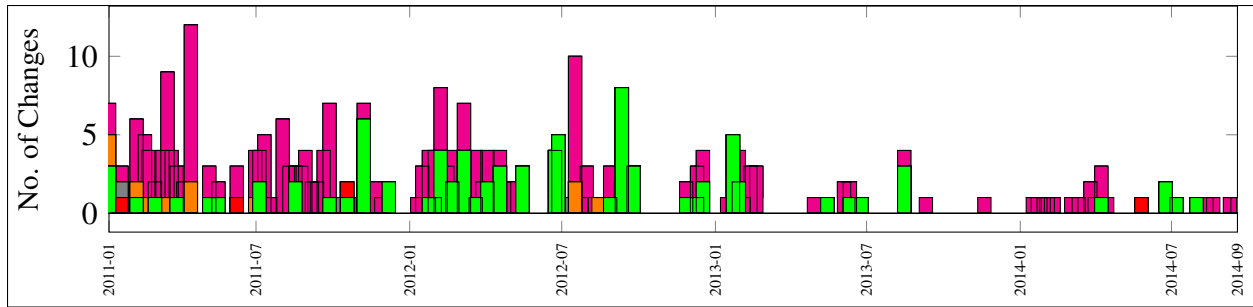
(e) Google Maps Web API (102 Releases)
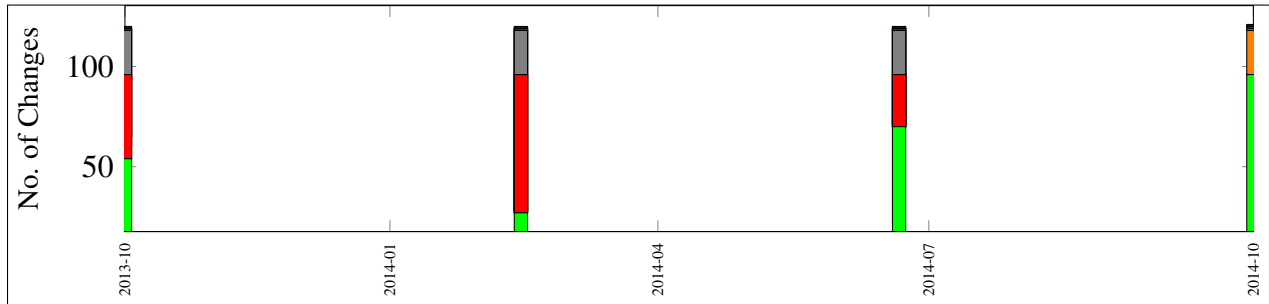
(f) Salesforce REST API (4 Releases)

AddAPIElements	RemoveAPIElements	ChangeAPIElements	ChangeFieldDataType
ErrorConditionChange	ChangeAuthentication	BehaviorChange	ChangeAuthorization
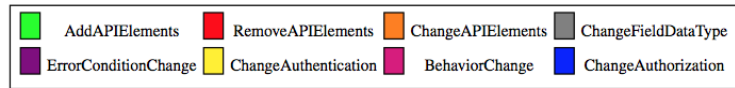
(g) Legend

Figure 2.1: Case Study - Change Profiles for the Web APIs.

carried out to compare and contrast the benefits of these modes in greater detail.

Change Profiles

The Web APIs evolve at different frequencies as well as the actual change that takes place during their evolution varies from one API to another. Fig. 2.1 shows an aggregated view of the count of different categories of changes observed for different Web APIs based on the aforementioned coding scheme.

The APIs here show very different change trajectories. Facebook API shows an alternating pattern of smaller and larger number of changes between subsequent versions, and each release consists of multiple categories of changes. On the other hand, Stripe and GitHub API changes are

more frequent, but on the majority of the releases only include a single category of change. Word-Press REST API releases contain multiple categories of changes, but shows only a single API element removal for the studied period. GoogleMaps API changes are also very frequent, dominated by AddAPIElement and BehaviorChange categories in a way that once new API elements are introduced, the subsequent releases primarily focus on bug fixes. The GoogleMaps API shows signs of stability with time. The Salesforce API changelog shows a different picture. Unlike the other APIs studied, Salesforce API evolves quarterly and involves a larger number of API elements with each release. We observed most Web API releases broke backward compatibility as shown by the presence of colors in addition to the green areas on the charts. We consider API changes to be backward compatible when changes are limited to only AddAPIElements category.

Google Calendar API, not shown in a chart, changed the format of all API request and response data from a custom XML to JSON in a single release, effectively forcing all its users to upgrade with a deprecation policy of one year. A plot for the Twitter and OpenStreetMap APIs could not be produced since changelogs for these APIs were no longer available.

These change profiles have little similarity, implying the fact that a common solution to solve their evolution related challenges may not work. Future research on Web API evolution needs to consider these variation of change profiles to find effective solutions for the case in hand.

Implementation

In our case studies, we included two open-source Web APIs, WordPress and OpenStreetMap, because it gave us access to their source code to study how versioning is actually implemented. Wordpress REST API is developed as a plugin for Wordpress. As a result, a new version of the Web API is released with a new version of the plugin and it all API clients are affected when a new version is deployed. The source code for OpenStreetMap.org Web API shows the use of configuration files for API Versioning. We found its implementation only allows for a single API version.

To summarize the change profiles, version identifiers and implementation of evolving Web API

24

versions, we observed little convergence and a lack of any common scheme that was followed by the studied Web APIs.

2.5.2   RQ2 Documentation

Web API documentation is used by API users as the primary source of information for system integration. We analyzed the API reference documentation to understand the contents and the presentation style of their documentation. The documentation is commonly comprised of the following:

Web API access information - Web API documentation includes information about the URL where the API can be reached. The studied APIs used one of the URL, HTTP headers, or a user interface to identify specific API versions.

Authentication - Web API documentation includes information about authentication for their API clients. This is also used to implement client specific rules, such as rate limits and time zones.

API Interface definition - The interface definition is a major part of the Web API documentation where each API element is described with a general overview and related business rules.

Example usage - Examples that describe different use cases of the Web APIs are also commonly found in the documentation. The examples contain the data and HTTP headers for sample requests and the corresponding API responses.

Some of the studied Web APIs also provide live web browser based API explorers, where API users can make API calls without needing to write any code. However, the studied Web APIs that offered Live API Explorer only did so for the latest API versions.

Web API documentation needs to include additional information compared to local APIs since HTTP is used for communication. As an API evolves lack of documentation for older API versions causes difficulties with the upgrade process as found in the following comment from a Stripe API user[11]

---

[11]`https://groups.google.com/a/lists.stripe.com/forum/#!searchin/api-discuss/version/api-discuss/li4PyVcweiw/NT9SFTtF-vQJ`

25

"Does the full API documentation only reflect the current version of the API? Is there a way to access the API docs for outdated versions? ...That would be very helpful. When you are trying to upgrade from one version to another it's impossible to know the implementation differences. We are currently about 4 API versions behind and are stuck behind a version that causes a significant amount of work on our end to support. I'd like to be able to upgrade incrementally through each version."

We also investigated how the API documentation is generated. We found that, Web API documents are manually generated and updated, unlike the documentation for local APIs which are are typically auto-generated. For WordPress and OpenStreetMap, this is apparent from their source code, as we found the use of manually edited wiki and Markdown files to generate the documentation[12].

In our case studies, we found one example of self-documenting Web API as discussed by Laskey [7]. The Salesforce Web API has API endpoints that describe all available versions of the API that can be used to programmatically determine the changes between versions.

To summarize, we found that Web API documents are manually generated and widely vary in terms of both their contents and presentation formats.

2.5.3   RQ3 Communication

Communicating updates and changelogs is an important activity for evolving Web APIs. The communication channels used by the studied Web APIs is categorized as follows:

API home page - Web APIs announce the API changes in their home pages. In practice, the announcements include partial change log with the key changes that are made in a release. These announcements are unstructured text and do not follow any standard format even for the same API.

API response - Some Web APIs use custom HTTP headers to indicate when a call is made to a deprecated version. For example, Wordpress REST API sends the X-WP-DeprecatedFunction

---

[12]http://en.wikipedia.org/wiki/Markdown

header in response to API calls made to deprecated endpoints.

Email - Facebook sends customized email alerts to its API clients based on their usage of the API, similar to the idea as presented by Zou et al. [18]. API calls are recorded to determine if a change has an impact on users. The granularity of the change tracking often results into false-positives, as found in this comment by a Facebook API user [13]:

> ...I have received the same notification for one of my apps. My app definitely does not read or create comments on Facebook posts or objects ... Thus Facebook's message is not relevant to every app they send it to. The change should only affect apps which read or publish comments.

Newsfeed - To keep up-to-date with API changes, the Web API clients are requested to subscribe to newsfeed, typically delivered via mailing lists and Twitter feeds. User feedback is also collected on these platforms. For example, at the time of writing this paper there were 13,239 questions tagged against google-maps-api-3[14].

To summarize, we found both formal and informal channels are used to communicate Web API changes. Most messages are primarily driven by manually edited unstructured text.

2.6 Discussion

Our case study findings such as a lack of standard approaches to deal with Web API versioning, documentation and change communication confirms the findings from previous case studies. In addition to the Web API change patterns identified by previous case studies, we have identified six new change patterns: 1) move API elements, 2) rename API elements, 3) behavior change, 4) post condition change, 5) HTTP header change, and 6) error condition change. When creating API changelog and developing related tool support, these new change patterns can be used to communicate the changes using a shared vocabulary. In addition to these new change patterns,

---

[13]http://stackoverflow.com/questions/16270446/updating-app-for-breaking-change-non-thread

[14]https://stackoverflow.com/questions/tagged/google-maps-api-313.239

from our case study we have compiled a list of recommendations and identified new research problems related to RQ1-3 as discussed in the following paragraphs.

RQ1 Versioning - From our case study, we found that when a new version is released for a Web API, the version identifiers provide little information about backward compatibility and the impact of the new version on existing API users. As a result, the API users are left to follow the free-form newsfeeds to understand the impact for each new API version. To improve this situation, for practitioners, we recommend using Semantic Versioning. Semantic Versioning is a naming technique for versioning software so that it is possible to infer if two versions are backward compatible simply by interpreting the version identifiers[15]. Future research can be carried out to automatically generate a Semantic Versioning identifier based on the compatibility between versions.

Our change profiles show that Web APIs often evolve as fast as several times a month and forces API clients to upgrade. Also, releases often combine both bug fixes and other breaking changes, forcing the API users to adapt even though they only need the fixes. For practitioners, we recommend that bug fixes and breaking changes be released under separate versions to provide more flexibility to the API clients. Future research needs to focus on finding strategies for evolving Web APIs so that bug fixes and other changes can be easily delivered in separate versions and multiple versions can be made available at the same time.

RQ2 Documentation - We observed a that largely manual process is used for generating API documentation for most RESTful APIs. Manually generating and maintaining such documentation is expensive and error-prone. From our analysis, we showed that even when Web APIs are versioned, documentation for older versions is not always available, and the difference between two versions of a Web API cannot be easily derived from their manually edited API references. We also found that the changelogs and API references are two disconnected documents even though a user needs to read both documents while upgrading. For practitioners, we recommend releasing Web API documentation for each available version that is cross-linked with the changelog. Future

---

[15]http://semver.org/

research on tool development can be aimed to support automated, version-aware documentation needs of evolving Web APIs.

Live Web API explorers are used by API users to try the APIs with little effort. However, there is a lack of reusable approach to create Live Web API explorers and bespoke implementation of the live API explorers are expensive since they need additional software development. We have seen only two of the nine studied APIs to provide a live API explorer. Moreover, we found that live Web API explorers are only offered for latest versions. Future research can focus on the implementation of a reusable, version-aware live Web API explorer for evolving Web APIs.

Self documenting Web APIs can be used to automatically determine compatibility between two versions of a Web API. We found Salesforce REST API to be the only example from our case studies that implemented self documenting Web APIs. For practitioners, we recommend providing self-documenting Web APIs. We also identify this as an opportunity for future research so that reusable tool support can be provided to create self-documenting Web APIS.

RQ3 Communication - For communicating API changes, we found a number of different channels that are used in practice. These channels need to be compared so that the effective channels can be used for communicating between the API developers and users.

In the change profiles we found the APIs change frequently and this requires manual effort to understand the impact of a change on a specific API user. This can be simplified by alerting users about an API change with a customized changelog. We also found that API related discussions are carried out external to the API documentation sites. API users need to search multiple disconnected information sources when API related questions arise. For practitioners, we recommend publishing a customized changelog for their Web APIs and providing discussion forums with API documentation. Previous research showed a solution based on WSDL files [18]. Future research is needed to solve this problem for RESTful Web APIs.

The open source Web APIs (Wordpress and OpenStreetMap) had limited documentation and change log information compared to the proprietary Web APIs. This presents an opportunity for

29

researchers and industry practitioners to create reusable open source versioning and documentation tools for evolving Web APIs.

Limitations - We recognize the number of Web APIs studied to be a threat to the construct validity. Although our selection involves a diverse set of Web APIs, the findings may not be generalized for all evolving Web APIs. Our selection criteria included different industry domains, API sizes, popularity and maturity levels to minimize selection bias. The selected APIs represent only publicly available evolving Web APIs because only publicly available data is used for the case study. Future work need to be carried out to compare the applicability of our findings and related implications on privately maintained evolving Web APIs. To mitigate the interpretation bias of the coding scheme, the first two authors of this paper independently annotated randomly selected snippets of change logs from different Web APIs and reached the same conclusion indicating the replicability of the given coding scheme.

Given the state of practice we found a wide variety of ways to evolve Web APIs in terms of versioning, documentation and communication of changes and no consistent way to deal with the changes. This indicates an immature area where cost-effective solutions that are acceptable for both API developers and users are still missing - as discussed. Further research needs to be carried out to validate the usefulness of the aforementioned recommendations and solve the new research problems as identified by this case study.

2.7 Conclusion

Web API evolution poses challenges since a change may break applications that are developed by different teams and organizations. We presented a case study of Web API evolution focusing on the challenges involved with versioning, documentation and communication of API changes. Recommendations for practitioners from our analysis include the use of semantic versioning, separate releases for bug fixes and new features, auto generated API documentation cross-linked with changelogs and providing live API explorers. A list of open research problems are discussed re-

lated to Web API evolution that we want to solve in our future work.

2.8   References

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

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

[3] D. Romano and M. Pinzger, "Analyzing the evolution of web services using fine-grained changes," in *Proc. of International Conference on Web Services (ICWS)*.   IEEE, 2012, pp. 392–399.

[4] J. Li, Y. Xiong, X. Liu, and L. Zhang, "How does web service api evolution affect clients?" in *Proc. of International Conference on Web Services (ICWS)*.   IEEE, 2013, pp. 300–307.

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

[6] P. Kaminski, M. Litoiu, and H. Müller, "A design technique for evolving web services," in *Proc. of Conference on Center for Advanced Studies on Collaborative Research (CASCON)*.   IBM Corp., 2006, p. 23.

[7] K. Laskey, "Considerations for soa versioning," in *Proc. of Enterprise Distributed Object Computing Conference (EDOC) Workshops*.   IEEE, 2008, pp. 333–337.

[8] M. Treiber, H.-L. Truong, and S. Dustdar, "On analyzing evolutionary changes of web services," in *Proc. of Service-Oriented Computing Workshops (ICSOC)*.   Springer, 2009, pp. 284–297.

[9] L. Aversano, M. Bruno, M. Di Penta, A. Falanga, and R. Scognamiglio, "Visualizing the evolution of web services using formal concept analysis," in *Proc. of International Workshop on Principles of Software Evolution*.   IEEE, 2005, pp. 57–60.

[10] E. Wilde, "Semantically extensible schemas for web service evolution," in *Web Services*. Springer, 2004, pp. 30–45.

[11] R. Fang, L. Lam, L. Fong, D. Frank, C. Vignola, Y. Chen, and N. Du, "A version-aware approach for web service directory," in *Proc. of International Conference on Web Services (ICWS)*.   IEEE, 2007, pp. 406–413.

[12] M. B. Juric, A. Sasa, B. Brumen, and I. Rozman, "WSDL and UDDI extensions for version support in web services," *Journal of Systems and Software*, vol. 82, no. 8, pp. 1326–1343, 2009, sI: Architectural Decisions and Rationale.

[13] W. Zuo, A. Benharkat, and Y. Amghar, "Change-centric model for web service evolution," in *Proc. of International Conference on Web Services (ICWS)*.   IEEE, 2014, pp. 712–713.

[14] J. Mangler, P. P. Beran, and E. Schikuta, "On the origin of services using riddl for description, evolution and composition of restful services," in *Proc. of International Conference on Cluster, Cloud and Grid Computing (CCGrid)*.   IEEE/ACM, 2010, pp. 505–508.

[15] P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar, "End-to-end versioning support for web services," in *Proc. of International Conference on Services Computing (SCC)*, vol. 1. IEEE, 2008, pp. 59–66.

[16] R. Meng and C. He, "A comparison of approaches to web service evolution," in *Proc. of International Conference on Computer Sciences and Applications (CSA)*, 2013, pp. 138–141.

[17] M. Fokaefs and E. Stroulia, "Wsdarwin: Studying the evolution of web service systems," in *Advanced Web Services*, A. Bouguettaya, Q. Z. Sheng, and F. Daniel, Eds.   Springer, 2014, pp. 199–223.

[18] Z. Le Zou, R. Fang, L. Liu, Q. B. Wang, and H. Wang, "On synchronizing with web service evolution," in *Proc. of International Conference on Web Services (ICWS)*.   IEEE, 2008, pp. 329–336.

[19] J. Kopecky, K. Gomadam, and T. Vitvar, "hrests: An html microformat for describing restful web services," in *Proc. of International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, vol. 1.   IEEE, 2008, pp. 619–625.

[20] R. Verborgh, T. Steiner, D. Van Deursen, J. De Roo, R. d. Walle, and J. GabarrÃş VallÃľs, "Capturing the functionality of web services with functional descriptions," *Multimedia Tools and Applications*, vol. 64, no. 2, pp. 365–387, 2013.

Chapter 3

SpyREST: Automated RESTful API Documentation using an HTTP Proxy Server

3.1 Abstract

RESTful API documentation is expensive to produce and maintain due to the lack of reusable tools and automated solutions. Most RESTful APIs are documented manually and the API developers are responsible for keeping the documentation up to date as the API evolves making the process both costly and error-prone. In this paper we introduce a novel technique using an HTTP proxy server that can be used to automatically generate RESTful API documentation and demonstrate SpyREST, an example implementation of the proposed technique. SpyREST uses a proxy to intercept example API calls and intelligently produces API documentation for RESTful Web APIs by processing the request and response data. Using the proposed HTTP proxy server based technique, RESTful API developers can significantly reduce the cost of producing and maintaining API documentation by replacing a large manual process with an automated process.

3.2 Introduction

RESTful APIs, introduced by Fielding, are used as a primary interconnection mechanism among modern day web based systems [1]. For example, the website of a restaurant can use the RESTful API from Twitter to show the latest tweets mentioning the restaurant so that prospective customers can read the experience shared by others. To allow others to use their APIs, Twitter and other RESTful API developers publish documentation describing different features of their RESTful API. The documentation of such RESTful APIs are often produced and maintained using a manual process that is expensive and error-prone.

34

API documentation for library APIs, such as Java Standard Edition, commonly leverage reusable tools such as JavaDoc. The documentation produced by such tools include description of objects and methods, with custom texts primarily sourced via comments in the source code. On the other hand, RESTful API documentation includes additional information such as HTTP headers, request parameters, request and response data in serialized formats (e.g. JSON, XML). Using comments for these additional information requires significant manual effort. There is a lack of reusable tools to automate the documentation process. This makes the task of producing RESTful API documentation a costly and error-prone one. API developers also need to publish and often maintain the documentation for multiple versions as the RESTful API evolves. This requires further manual effort.

To produce RESTful API documentation, the manual process used can be described as a six-step process as follows: 1) craft an example call to an API endpoint with required headers, URL parameters and request body, 2) make the call, 3) capture the response headers and data, 4) strip any confidential data from the captured information, 5) add custom descriptions to the captured data and 6) publish the API documentation. This six-step process is essentially repeated for all API endpoints that are documented. With *SpyREST*, we have implemented an innovative solution to largely automate the aforementioned manual process of RESTful API documentation so that all but steps 1 and 5 are automated. Steps 1 and 5 are left to a manual process to allow for a pragmatic solution. Our solution relies on an HTTP proxy server to intercept example API calls to automatically capture all HTTP traffic. The collected data is then synthesized to present the documentation for RESTful APIs. SpyREST can generate documentation for all RESTful APIs irrespective of the technology used to implement the API since it leverages an HTTP proxy server.

The key contributions of our research are as follows: we discuss a list of requirements for tool development by analyzing related work to automate the RESTful API documentation process and present a new technique based on an HTTP proxy server to meet the requirements. We demonstrate an example implementation of the proposed technique that generates automated yet customizable,

version-aware, collaboration enabled and reusable API documentation software as a service platform that can be used to generate and maintain documentation for any RESTful API. We present a case study of using SpyREST to compare the advantages of our proposed technique over existing solutions for RESTful API documentation.

The remainder of this paper is organized as follows: in the next section we discuss related work on API documentation. The requirements and implementation details of SpyREST is provided next. Then, we present a case study to demonstrate SpyREST in action. Finally, we discuss the implications and limitations of our work in the discussion section.

3.3 Related Work

3.3.1 General API Documentation

Several papers have studied the documentation of APIs to recommend best practices that are also applicable to RESTful API documentation. Robillard et al. found that developers faced severe obstacles learning new APIs due to inappropriate documentation and other learning resources [2, 3]. Robillard recommended the following for API documentation: include good examples, be complete, support many complex usage scenarios, be conveniently organized, and include relevant design elements. Kuhn et al. discussed the importance of examples in API documentation [4]. Hoffman et al. recommended making the API example scenarios to be executable test cases so that a user can execute an API [5].

Nasehi et al. performed a case study based on StackOverflow discussions to find out what makes good code examples [6]. They recommended API developers to include examples in the API documentation and the use of wiki-like collaborative tools with online API documentation. Parnin et al. found that using social media gives API documentation readers a chance to engage with authors of the APIs [7]. Chen et al. recommended integrating crowdsourced frequently asked questions into API documents so that users can easily find relevant discussions when questions arise [8]. Subramanian et al. presented an automated approach to link API documentation of

36

different Java and JavaScript libraries with code examples that are shared on StackOverflow [9].

Stepalina identified several advantages of software as a service (SaaS) based solutions for API documentation systems such as, cost effective yet powerful, platform agnostic and high accessibility, improved document quality, content re-use, automated tools, and organization of robust and scalable documentation process [10]. Several tools exist that help automatic generation of API documentation for local APIs such as JavaDoc[1], RDoc[2], Jadeite [11]. While these tools have been proven to work for local library APIs, they have limited applicability for documenting RESTful APIs because HTTP specific information are not natively supported by these tools.

3.3.2 RESTful API Documentation

Several related work proposed specifications for RESTful APIs. Maleshkova et al. found that a lack of a standard format to document Web APIs and manual documentation causes confusions about how to use the APIs for different use cases [12]. Espinha et al. observed that most RESTful APIs are documented manually by API developers making the documentation a less reliable one [13]. Danielsen et al. presented a vocabulary for documenting RESTful Web APIs called Web Interface Language (WIfL) [14]. Verborgh et at. presented RESTdesc, a Resource oriented and Hyper-link based specification for describing RESTful APIs [15]. Mangler et al. presented RDDL, an XML based specification for describing RESTful APIs [16]. KopeckyÂt' et al. presented hRESTS, a machine readable micro-format to describe RESTful APIs that use an alternate representation compared to WIfL [17]. Ning et al. presented OmniVoke, a RESTful API based invocation engine that provides an abstraction layer for RESTful API calls to multiple APIs that follow different conventions [18]. Manual configuration is required to generate these aforementioned specifications for RESTful APIs. Myers et al. recommended providing a consistent look-and-feel with explanation for the starting points and an overall map comprising of both text and diagrams, providing a browsing experience with breadcrumb trail following a hierarchy, an effective search

---

[1]http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html
[2]http://rdoc.sourceforge.net/

37

interface, providing example code and a way to exercise the examples online without writing code [19].

In addition to the research community, there are several custom RESTful API specification formats that are observed in the industry such as, Swagger[3], RAML[4], and Blueprint[5]. In addition to producing RESTful API documentation, there are SaaS based tools providers that can be used to publish the documentation and auto generate API client code for RESTful APIs that are described using one of these formats. To use these tools to generate HTML based RESTful API documentation, API developers need to manually construct the intermediate documentation format such as Swagger, Blueprint, RAML, etc. since there is no automated tool to produce this.

3.4 SpyREST

3.4.1 Requirements

The following list of requirements for SpyREST, R1-7, in priority order, is derived from analyzing the aforementioned related work and current API documentation practices as observed in the industry:

R1 - Automated RESTful API documentation: RESTful APIs are either documented manually or using custom tools to partially automate the process. The primary requirement for SpyREST is to find a cost-effective approach to automate RESTful API documentation.

R2 - Example based: As discussed in the related work section, several authors have emphasized including example scenarios with API documentation can help users understand how to use an API [2, 4, 5, 6]. SpyREST generated API documentation needs to include example scenarios.

R3 - Executable documentation: In addition to including examples, SpyREST also needs to allow users to execute the example scenarios so that they can try the API features without having to write code [5, 19].

---

[3]https://github.com/swagger-api/swagger-spec/blob/master/versions/2.0.md
[4]http://raml.org/spec.html
[5]https://github.com/apiaryio/api-blueprint

R4 - Version awareness: SpyREST needs to allow API developers to publish documentation for multiple versions of a Web API as it evolves. The following comment from an API client developer of Stripe, an online payment processing company, shows the importance of a version-aware documentation tool[6] :

> "Does the full API documentation only reflect the current version of the API? Is there a way to access the API docs for outdated versions? ...That would be very helpful. When you are trying to upgrade from one version to another it's impossible to know the implementation differences..."

R5 - Customizable: It is important to customize the overview information about how to use an API to reduce API learning obstacles and describe complicated business rules[2]. SpyREST needs to allow API developers to add customized content to auto generated RESTful API documentation.

R6 - Reusable: SpyREST needs to work as a reusable platform so that multiple REST API documentations can be generated and published on a single platform to get the advantages of a SaaS platform [10].

R7 - Collaborative: SpyREST needs to allow people to collaborate on the API documentation so that questions and answers about APIs can coexist with the API documentation to to overcome the knowledge fragmentation [8].

3.4.2 Design

From our research, we identified an HTTP proxy server based solution as a novel approach for generating RESTful API documentation meeting requirements R1-7. Our approach works as follows: API developers make example API calls using a proxy $\longrightarrow$ the proxy records and processes HTTP traffic $\longrightarrow$ a Web App generates RESTful API documentation. SpyREST, an example implementation of this approach[7], is composed of three main components as follows:

---

[6]https://groups.google.com/a/lists.stripe.com/forum/#!searchin/api-discuss/version/api-discuss/li4PyVcweiw/NT9SFTtF-vQJ
[7]https://github.com/smsohan/demo_paper/raw/master/SpyREST_tool_demo.pdf

The Spy: The Spy component in SpyREST is an HTTP proxy server. The Spy is named so because it records the HTTP traffic for example API calls that are made using it as a proxy server. For example, when using Spy as an HTTP proxy to make the following HTTP request:

Method GET

URL https://api.github.com/repositories?since=100

Headers accept: application/vnd.github.v3+json

that produces the following response:

Headers status: 200 OK

content-type: application/json; charset=utf-8

...

Body [
  {
    "id": 1,
    "name": "grit",
    "full_name": "mojombo/grit",
    ...
  }
]

Spy automatically saves the raw request and response data with HTTP headers. Additionally, the Spy synthesizes the data and saves the following meta data about this example API call:

version	v3
resource	repositories
action	GET /repositories
query	since: 100
strippedResponseBody	a subset of the response body
description	blank
digest	base64 hash value of the version, resource, url and description
requiresAuth	false
apiToken	blank, used when a SpyREST API token is provided
userId	blank, used when a user is found for given apiToken

As shown in this example, the version "v3" is automatically detected by parsing the accept request header. Next, the resource field is also auto detected as "repositories" by parsing the request URL. The action "GET /repositories" is automatically detected by combining the request HTTP method with request path. The strippedResponseBody field automatically saves a shorter version of the actual response where large response body is truncated to a smaller one to reduce noise from the generated documentation. These meta fields allow SpyREST to structure the example API calls for a given API host in a hierarchical model as follows: an APIHost (e.g. GitHub.com) has many APIVersions (e.g. v3), each APIVersion has many APIResources (e.g. repositories), each APIResource has many APIActions (e.g. GET /repositories), and each APIAction has many APIExamples.

The Database: The database component saves the data that is captured by the Spy and the Web App. The Spy saves the recorded data about the API examples and the auto-computed meta data in the database. The Web App saves data about SpyREST users and custom modifications on auto-generated API documentation on the same database.

The Web App: The Web App component further processes and presents auto generated documentation that is captured by Spy in the database. The Web App also allows the API developers to edit or add custom content to the auto generated documentation. The source code for both the Spy and Web App components are released as open source projects.

41

(a) API documentation summary section   (b) API documentation examples section

Figure 3.1: SpyREST Screenshots Showing Auto Generated API Documentation

3.4.3 Features

Now that we have explained the design and implementation of SpyREST, we discuss how SpyREST meets the aforementioned requirements R1-7.

R1 - Automated RESTful API documentation: The work flow for automated RESTful API documentation can be explained by the following steps: 1) API developer uses the Spy HTTP proxy to run example API calls, 2) The Spy records example API calls, 3) The Spy extracts meta information about the API call, 4) The Web App displays the auto generated documentation, 5) API developers can optionally customize the auto generated documentation.

Fig 4.2 shows screenshots (shortened) from SpyREST auto generated documentation that is solely based on an example API call to URL `https://api.github.com/search/repositories?q=tetris+language:assembly&sort=stars&order=desc` with a custom header "x-

42

spy-rest-description" to provide the short description for the example.

The auto generated API documentation features two sections, a summary section as shown in Fig. 4.2a and an examples section as shown in Fig. 4.2b. The summary section includes breadcrumbs to show hierarchy of the API objects related to each API action to help API client developers easily navigate the API documentation. This section also includes three tables that display the structure of query parameters, and request and response payloads. In addition to the structure, automated type detection is used to show the data type and example values for each field in these tables. The examples section on the API documentation shows all the recorded API examples for a given API action. For each example, it shows a description, the request URL, query parameters, and request and response headers and bodies. Spy filters out the "authorization" request header before saving the examples in the database and the documentation rendered by the Web App displays this header with a placeholder text as "FILTERED".

R2 - Example based: SpyREST generated RESTful API documentation includes both the structure of the API objects and concrete examples for different use cases. The API examples are annotated with user provided descriptions through a custom HTTP request header "x-spy-rest-description" that is otherwise hard to automatically infer. To record the examples in SpyREST, any REST API client can be used as long as it supports an HTTP proxy to call the API.

R3 - Executable documentation: SpyREST keeps a copy of the API request information including URL, headers and request body. As a result, SpyREST can also recreate the example API calls that can be executed by users of the generated API documentation. SpyREST automatically generates executable test cases that can be run using cURL[8].

R4 - Version awareness: SpyREST allows API developers to publish the documentation for multiple versions of their RESTful APIs. To organize API documentation and examples under multiple versions, SpyREST has an automatic version detection algorithm that parses the "accept" request header or the URL. The auto version detection algorithm can be suppressed by specifying the version in the custom "x-spy-rest-version" header when running example API calls through the

---

[8]http://curl.haxx.se/

Spy.

R5 - Customizable: SpyREST allows API developers to modify and add custom free-form contents to the auto generated summary section of the documentation using Markdown[9] syntax. Custom content can be used to provide overview information and explain complicated business rules about the APIs that are not derivable from simply synthesizing the examples. We consider this as a pragmatic solution to augment manual effort with the largely automated solution to the RESTful API documentation effort. The custom edits are persisted in the database and are not overridden when the API examples are replayed unless the user decides to revert back to automated summaries.

R6 - Reusable: SpyREST relies on an HTTP proxy to auto generate RESTful API documentation. To feed SpyREST with data, any REST API client can be used as long as it supports using a proxy server. This technology agnostic feature allows SpyREST to be a reusable RESTful API documentation tool. SpyREST is offered as a SaaS tool at `http://spyrest.com` that can be used to auto-generate and publish documentation of multiple RESTful APIs. For APIs where a SaaS solution is not acceptable, SpyREST can also be installed and used as a self-hosted solution. Because SpyREST is an open-source application, users can modify the source to support any unique requirements that are not supported by SpyREST.

R7 - Collaborative: SpyREST allows the API developers to comment and discuss API related questions with API client developers right next to the documentation so that crowdsourced documentations can coexist with the officially published API documentation. Each API documentation page on SpyREST features its own discussion thread to help users easily locate relevant information.

---

[9]`http://en.wikipedia.org/wiki/Markdown`

3.5 SpyREST Case Study

In this section we discuss a case study of using SpyREST to auto generate documentation for 25 API actions randomly sampled from three RESTful API providers. These 25 API APIs are documented using 272 lines of code[10]. These three API providers are: GitHub.com, KISSMetrics.com (Online analytics tool), and LiquidPlanner.com (Online project management tool).

3.5.1 SpyREST Documentation vs. Official Documentation

We found SpyREST generated documentation for 5 of the 25 API actions from the case study included fields that are found from actual API responses but not included in their official documentation. For example, the SpyREST generated documentation for Github `GET /notifications` API action included 34 additional fields, such as: `forks_url`, `keys_url`, `collaborators_url`, and 29 more that were not mentioned in the official API documentation even though actually returned as API response. Similarly, the official documentation for GitHub.com `GET /search/code` API action did not include the `releases_url` API response field. Official documentation of KISSMetrics.com `GET core/accounts` and `GET core/accounts/:account_id` API actions did not mention the `data` field that is found in the actual response as documented by SpyREST. The official documentation for `GET /api/account` action on LiquidPlanner.com did not mention the fields `workspaces`, `last_workspace_id`, and `disabled_workspaces_cour` that were included in the SpyREST documentation. These examples show the error-prone nature of a manual process that requires API developers to ensure the documentation is updated to reflect any change in the API. We consider the proxy based solution to solve this problem since the API documentation can be updated by replaying the example API calls.

The official documentation of these three API providers did not include integrated collaboration features. We found 172 unanswered questions out of 662 questions with the tag `github-api` on StackOverflow.com. We found 123 questions about LiquidPlanner API on their developer forum

---

[10]`https://github.com/smsohan/spyrest_examples`

that are not linked to officially generated documentation published as a PDF file. On StackOver-flow.com, we found 4 unanswered questions out of 9 questions on KISSMetrics API. SpyREST provides the collaboration features with auto generated API documentation so that API developers and users can discuss and locate related discussions when browsing RESTful API documentation on a single web interface. SpyREST includes executable API examples, support for multiple versions and presents the auto generated documentation for different RESTful APIs using a consistent look and feel that are not provided by these studied APIs.

3.6   Discussion

RESTful API documentation is expensive, error-prone, and often incomplete because of the manual effort involved in the process. Our core contribution is a novel approach where an HTTP proxy server is used to largely automate the process for RESTful API documentation. There is a lack of tool support for automatically generating documentation of RESTful APIs. We presented SpyREST as an example implementation of the HTTP proxy server based solution to fill this need. The requirements for SpyREST are derived from analyzing the existing literature and industry practices as well as our own professional experiences developing RESTful APIs and their documentation. In the literature, several papers provided lists of recommendations for API documentation based on studying the existing tools and techniques and feedback collected from API developers. We found a lack of available tool support to generate RESTful API documentation following those recommendations and designed SpyREST to provide a solution. As a result, SpyREST features automated documentation with executable examples, customizable contents, collaboration, version awareness and reusability to provide a ready to use tool support for RESTful API documentation following the recommendations.

The novel approach of using an HTTP proxy sever offers some unique benefits over existing tools from academia and industry. For example, tools that rely on user provided comments on source code and code inspection, such as JavaDoc, require the user to write formatted comments

that are often applicable to a single programming language. The comments are not executable, and manual effort is required to ensure the comments and the API that it describes are kept in sync. SpyREST generated documentation can reflect the latest information since the documentation is generated after executing the example API calls. The verbose nature of RESTful API documentation makes it difficult to write and maintain the documentation as comments in source code. Using an HTTP proxy server, SpyREST offers a language agnostic solution so that RESTful APIs written in any programming language can be documented using a single tool.

Another key feature of SpyREST is the shared platform for publishing RESTful API documentation. A similar shared platform is provided by several other tools such as Swagger, Blueprint. We identify the key advantage of SpyREST over these tools to be the fact that SpyREST relies on example API calls instead of relying on a custom API specification that are required by these tools. The suggested API specifications can be used to define the structure of API objects but do not capture examples of API usage. The work flow for using these API specification formats can be described as follows: (1) execute example API calls $\longrightarrow$ (2) manually generate API specification $\longrightarrow$ (3) automatically generate API documentation. SpyREST does not rely on any custom API specification, so the associated manual effort can be largely avoided resulting in a smaller work flow as follows: (1) execute example API calls $\longrightarrow$ (2) automatically generate API documentation. We consider this process to be a cost effective approach for APIs with several endpoints or APIs that evolve. Unlike other available SaaS only solutions such as Swagger, and Blueprint, SpyREST can be used both as a SaaS and a self-hosted platform. The self-hosted mode can be used for documenting internal RESTful APIs that cannot be released on the internet or modifying the open-sourced code of SpyREST to fit unique API specific needs. We have demonstrated using SpyREST with automated test code. Using the test code helps automatically testing different use-case scenarios of the API actions in addition to generating the API documentation.

While SpyREST offers an automated solution for documenting RESTful APIs, API developers can customize the auto generated content. Several researchers have identified the need for rich con-

tent comprising of text and diagrams to explain complex business rules about APIs. To achieve this goal, we consider augmenting auto generated documentation with user contributed documentation, when necessary, to provide a pragmatic solution. To summarize, from our example implementation of SpyREST and the case-study, we have shown the advantages of the proposed HTTP proxy server based solution over the existing techniques for RESTful API documentation.

3.6.1 Threats to Validity

Internal Threats. Our proposed technique uses an HTTP proxy server that can only intercept and record the data when it is either in clear text or trusted to be decrypted by SpyREST. Because SpyREST can be used as a shared platform, any secrets used in the example API calls get decrypted in memory, even though not saved by SpyREST. To overcome these threat, we recommend API developers to use disposable secrets so that the secrets used to run an API call are valid for a single API call. For use cases where this is unacceptable, the self-hosted mode can be used as an alternative. SpyREST currently only supports JSON based APIs for their widespread use among popular RESTful APIs such as Twitter, Facebook, Google Maps.

External Threats. We used SpyREST to auto generate documentation for APIs randomly sampled from three RESTful API providers as a proof of concept and found the proxy server based solution met the requirements R1-7. Further evaluation is needed involving a larger set of RESTful APIs to evaluate the effectiveness of the proposed technique of using an HTTP proxy server for RESTful API documentation.

3.7   Conclusion

In this paper, we have presented a novel technique of using an HTTP proxy server to automate the RESTful API documentation process that otherwise requires a largely manual process. The proxy server based solution supports integrated features such as automated RESTful API documentation with executable example API calls, support for multiple versions, customization and collaboration

48

that are offered both as a SaaS and self-hosted platforms. These features have been recommended by existing research on the documentation of APIs. Overall, we conclude that a proxy server based approach shows a pragmatic solution to the RESTful API documentation problem. In the future, we will perform a qualitative user study involving RESTful API developers to evaluate the effectiveness about the aforementioned HTTP proxy based RESTful API documentation approach.

3.8 References

[1] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[2] M. Robillard, "What makes APIs hard to learn? the answers of developers," *Software, IEEE*, vol. PP, no. 99, pp. 1–1, 2011.

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

[4] A. Kuhn and R. DeLine, "On designing better tools for learning APIs," in *Search-Driven Development - Users, Infrastructure, Tools and Evaluation (SUITE), 2012 ICSE Workshop on*, 2012, pp. 27–30.

[5] D. Hoffman and P. Strooper, "API documentation with executable examples," *Journal of Systems and Software*, vol. 66, no. 2, pp. 143 – 156, 2003.

[6] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns, "What makes a good code example?: A study of programming Q&A in StackOverflow," in *IEEE International Conference on Software Maintenance*, 2012, pp. 25–34.

[7] C. Parnin and C. Treude, "Measuring API documentation on the web," in *Proceedings of the International Workshop on Web 2.0 for Software Engineering*, ser. Web2SE '11. ACM, 2011, pp. 25–30.

[8] C. Chen and K. Zhang, "Who asked what: Integrating crowdsourced faqs into api documentation," in *Companion Proceedings of the International Conference on Software Engineering*, ser. ICSE Companion 2014.   ACM, 2014, pp. 456–459.

[9] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live api documentation," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE 2014.   ACM, 2014, pp. 643–652.

[10] E. Stepalina, "SaaS support in software documentation systems," in *Software Engineering Conference (CEE-SECR), 2010 6th Central and Eastern European*, 2010, pp. 192–197.

[11] J. Stylos, B. A. Myers, and Z. Yang, "Jadeite: Improving API documentation using usage information," in *CHI '09 Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '09.   ACM, 2009, pp. 4429–4434.

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

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

[14] P. Danielsen and A. Jeffrey, "Validation and interactivity of web API documentation," in *International Conference on Web Services*.   IEEE, 2013, pp. 523–530.

[15] R. Verborgh, T. Steiner, D. Van Deursen, S. Coppens, J. G. Vallés, and R. Van de Walle, "Functional descriptions as the bridge between hypermedia APIs and the semantic web," in *Proceedings of the Third International Workshop on RESTful Design*, ser. WS-REST '12. ACM, 2012, pp. 33–40.

[16] J. Mangler, P. Beran, and E. Schikuta, "On the origin of services using riddl for description, evolution and composition of restful services," in *Cluster, Cloud and Grid Computing (CCGrid), IEEE/ACM International Conference on*, 2010, pp. 505–508.

[17] J. Kopecky, K. Gomadam, and T. Vitvar, "hrests: An html microformat for describing restful web services," in *Proc. of International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, vol. 1.   IEEE, 2008, pp. 619–625.

[18] N. Li, C. Pedrinaci, M. Maleshkova, J. Kopecky, and J. Domingue, "Omnivoke: A framework for automating the invocation of web apis," in *IEEE International Conference on Semantic Computing*, 2011, pp. 39–46.

[19] 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," *J. Organ. End User Comput.*, vol. 22, no. 1, pp. 23–51, 2010.

Chapter 4

SpyREST in Action: An Automated RESTful API Documentation Tool

4.1 Abstract

RESTful APIs are often manually documented. As a result, the process of maintaining the documentation of RESTful APIs is both expensive and error-prone. In this demonstration paper, we present SpyREST as an automated software as a service tool that can be used to document RESTful APIs. SpyREST leverages an HTTP Proxy server to intercept real API calls to automatically collect and generate RESTful API documentation by processing HTTP traffic involved in API calls. SpyREST provides an automated yet customizable example based documentation solution for RESTful APIs. RESTful API developers can use SpyREST to automatically generate and maintain updated API documentation.

4.2 Introduction

Fielding introduced RESTful APIs as a versatile mechanism for connecting internet based applications [1]. For example, a hotel web site may use the Google Maps API to provide driving directions to the hotel. RESTful API developers need to provide and maintain API documentations when releasing the APIs so that the APIs can be used. Maintaining API documentation manually is less efficient and often less reliable than automated documentation [2].

Automated solutions exist for documentation of local APIs within the context of classes, and methods. For example, JavaDoc can automatically generate the documentation for Java APIs. For RESTful APIs, instead of classes and objects, the context of APIs are defined by API endpoints, resources, actions, and example API calls in terms of the HTTP headers, request and response

52

payload formats [3]. The existing tools for documenting local APIs do not natively support the different context for RESTful APIs and cannot be readily used to document RESTful APIs.

In this paper we present SpyREST which provides the necessary tool support to address the unique needs for RESTful API documentation. To this regard, in the next section we have discussed the design goals followed by a HTTP proxy server based prototype implementation to demonstrate an automated RESTful API documentation tool. We then present two use cases to show how API developers can use SpyREST to automatically generate and maintain RESTful API documentation. In the discussion, we compare SpyREST against other tools. We provide an analysis of how the requirements for SpyREST are derived in the related work section.

4.3 SpyREST

4.3.1 Design

SpyREST is a web-based software as a service tool[1] [4]. The design goals of SpyREST is to provide an automated, technology agnostic, and example driven solution for documenting RESTful APIs so that SpyREST can be used a reusable tool to document multiple RESTful APIs . To auto-generate documentation for RESTful APIs, SpyREST relies on HTTP traffic information captured while example API calls are made using a HTTP proxy. The high level process can be described as a three step process: API developers make example API calls using a HTTP proxy server, the HTTP proxy server captures the HTTP headers, request and response data, the collected data is synthesized and presented on a web application.

The proxy server used in SpyREST is a specialized server for RESTful API documentation. A typical HTTP proxy server can intercept and record the HTTP traffic for example API calls but further processing is required to generate a usable RESTful API documentation out of the raw HTTP data. For example, API reference documentations need to be version aware so that the documentation can clearly refer to the relevant API versions. To provide human readable

---

[1]`http://SpyREST.com`

descriptions to the example API calls, relevant information needs to be captured in addition to the raw HTTP traffic. Post processing is required to generate the structure of request and response payloads by parsing and consolidating raw HTTP data from multiple example API calls since the optional fields may not be obvious when looked at individual example calls.

To generate usable documentation, SpyREST automatically parses the captured HTTP request and response data and presents summary tables showing the structure for HTTP query parameters, headers, request and response data. For each field, SpyREST displays the name of the field, example values and automatically inferred data type information such as integer, date time, boolean, and string. To reduce verbosity from large API responses, SpyREST automatically strips large responses to only show representative samples from repeated values in arrays. SpyREST does not capture and strip off any authorization headers used in the example API calls to prevent confidential information from being part of the documentation.
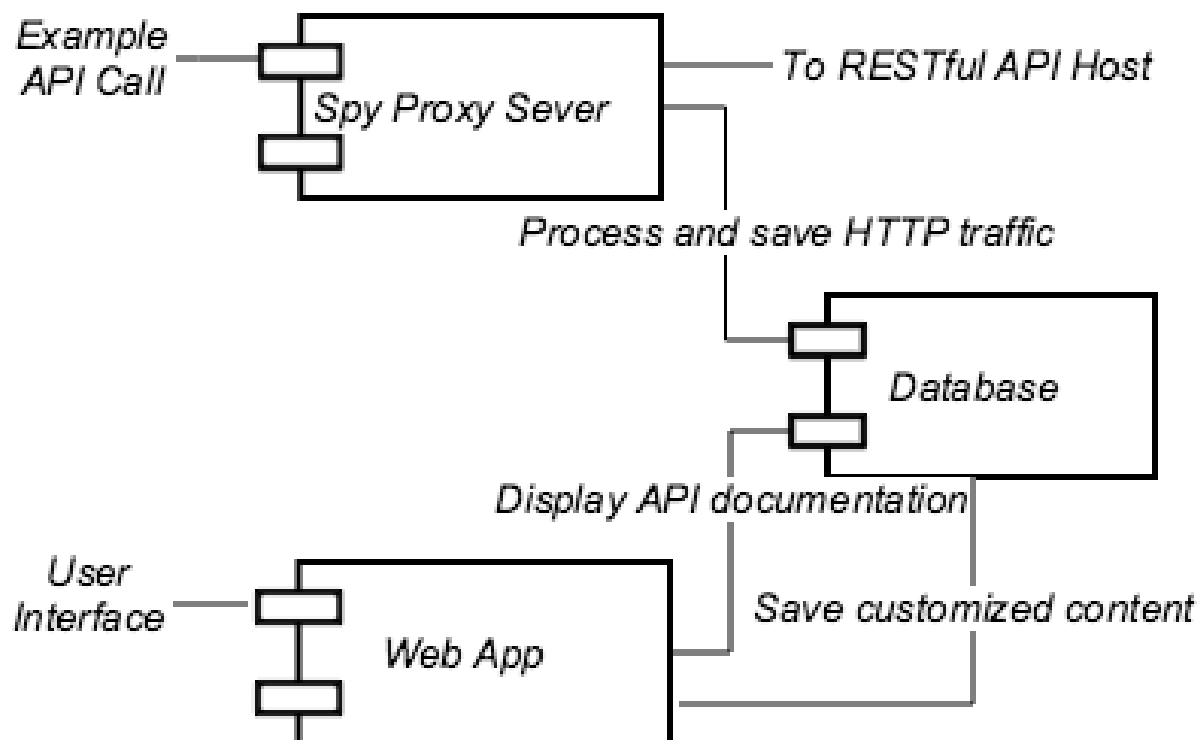

Figure 4.1: SpyREST Design

By using an HTTP proxy server, SpyREST enables the documentation of different RESTful

APIs independent of the implementation technology behind the APIs. SpyREST can be used both as a software as a service tool or as a self-hosted tool when isolation is desired. SpyREST is released as an open-source product and for self-hosted solutions it can be customized to fit unique requirements.

The web interface of SpyREST provides a hierarchical navigation for RESTful API documentation as follows: each API host has one or more API versions, each API version has one or more resources, each API resource has one or more API actions, each API action may have a structure of HTTP query parameters, headers, request and response payloads as well as many API examples, each API example may have HTTP query parameters, headers, request and response payloads. The web interface also provides a wiki-like editor to allow API developers to override auto generated API documentation. On each page, the web interface includes collaboration features so that API developers can discuss questions and collect feedback about specific API documentation pages. For each API example, SpyREST generates a cURL command that can be run to execute the API examples without writing any code.

4.3.2 Implementation

SpyREST is composed of three main components as shown in Fig. 4.1. The "Spy" component is the customized HTTP proxy server. This component is written using NodeJS. The Spy has internal logic to decode HTTP request URL and headers and auto-detect API versions for the commonly used version identifier patterns. Table 4.1 shows examples of auto detected versions from accept HTTP header and URLs.

The Spy also parses the URL to auto detect the API resources, API actions and query parameters for the example API calls. For example, given an example API call made to `GET https://api.github.com/repositories?since=100`, the Spy automatically detects the API resource as `repositories`, the API action as `GET /repositories`, and a query parameter `since` with an example value of `100`.

Even though the auto detected API version, resource and actions cover the commonly ob-

Table 4.1: Examples of Auto Detected Versions

Type	Example Value	Detected version
"accept" header	application/vnd.ex.ca.v3+json	v3
"accept" header	application/vnd.ex.ca.v3.1+json	v3.1
URL	`/v2/x`	v2
URL	`/v2.1/x`	v2.1
URL	`/v2.1-pre/users`	v2.1-pre

served patterns, API developers can override the auto-detections of any of these fields by using custom Spy specific HTTP headers when making the example API calls. The HTTP headers `x-spy-rest-version`, `x-spy-rest-resource`, and `x-spy-rest-action` can be used to override auto-detection of these respective fields. Additionally, API developers can use `x-spy-rest-desc` header to attach human readable descriptions for each API example so that the web interface can tag the examples against meaningful descriptions.

The "Web App" component is implemented using Ruby on Rails. To allow RESTful API developers to edit auto generated documentation, the Web App uses the Markdown Syntax. To facilitate collaboration across all the API documentation pages, the Web App uses Disqus, a popular third-party collaboration service, for commenting.

The Spy writes the captured and synthesized API example data into a MongoDB "Database". The Web App reads and saves custom edits on the same database.

All three SpyREST components are released as Docker containers. Docker is a lightweight virtualization solution that can either be hosted in-house or using many of the popular cloud hosting providers. The public instance of SpyREST has been tested with the three components in three separate docker containers running on a single Linux server with 512MB memory, 1 core processor, and 20GB disk space.

(a) API documentation summary section   (b) API documentation examples section

Figure 4.2: SpyREST Screenshots Showing Auto Generated API Documentation
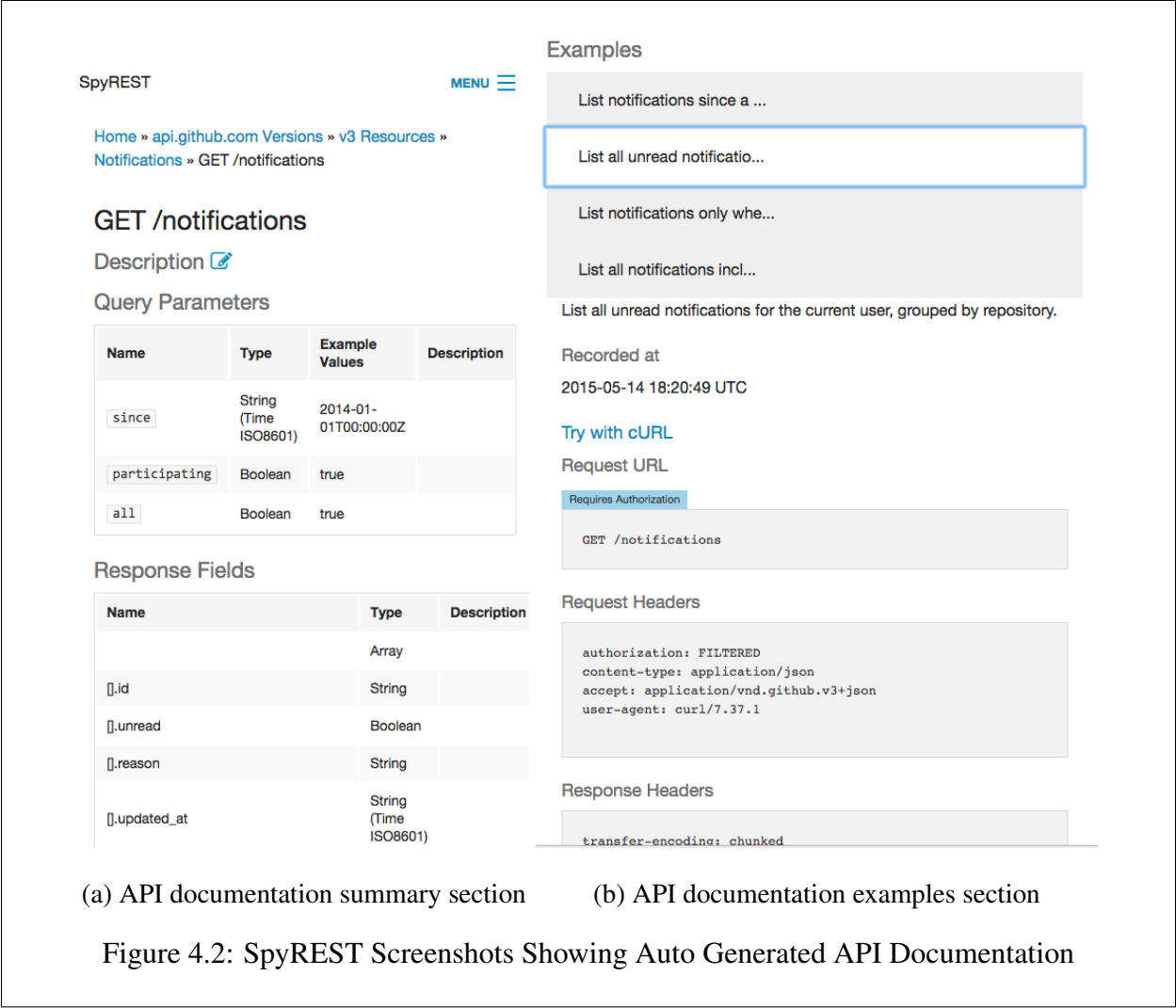
4.4 Use Cases

In this section, we describe two different use cases from the perspective of initially generating a RESTful API documentation for the first time and maintaining the documentation as the API evolves over time.

4.4.1 Initial API Documentation

When releasing a RESTful API, API designers need to publish the API documentation so that other developers can use the APIs. For demonstration, consider generating documentation for the

"GET /notifications (List your notifications)" API that is exposed by GitHub[2]. Using this API a developer can paginate through the list of all public repositories within GitHub. This API action takes an optional parameter "since" that can be used to specify a time filter.

To manually generate documentation for the GET /notifications API, a developer needs to follow these steps: S1) make example authenticated API calls to *GET/notifications*, S2) record the HTTP headers with request and response data, S3) remove duplicate information, S4) remove authentication information, S5) format and stylize the reduced information, S6) add custom content, S7) organize multiple formatted documents together on a web site with navigation, and S8) publish the documentation website. Steps S1-S7 need to be repeated for each of the API endpoints. Since these steps are manually performed, there is also a room for human errors in the aforementioned steps. SpyREST aims to minimize the potential for human errors by automating steps S2-S5 and S7-S8.

To use SpyREST for auto-generating documentation of an API, a developer only needs to make the example API calls via the Spy proxy server with custom HTTP headers for human readable descriptions for the API example. Custom edits to the auto generated documentation may be made by using the Markdown editor. Fig 4.2 shows a partial screen shot of SpyREST generated documentation for this API given four example API calls. SpyREST fully automates the steps S2-S5, and S7-S8 as follows: S2) the Spy proxy server records the HTTP traffic, S3) the Spy automatically strips off repeated items from the arrays in the response body, S4) the Spy does not save any Authorization header, S5) the Web App renders the documentation for all APIs following a consistent look and feel, S7) the Web App provides breadcrumbs for hierarchical navigation of the API objects, and S8) the recorded data is automatically published by the Web App in real-time. The API developers only need to focus on finding good example API calls for S1 and describing the complex rules about APIs that are not otherwise derivable from looking at the request and response data alone for S7. In addition to automating the RESTful API documentation process,

---

[2]https://developer.github.com/v3/activity/notifications/
#list-your-notifications

58

SpyREST also displays a cURL command for each recorded API example so that developers can try the API examples without having to write any code. Using in page collaboration developers can discuss API related questions and find past conversation all in one place. In this specific example, we see the official documentation of $GET/notifications$ provided by GitHub was outdated and missing several fields (e.g. $subscription\_url$, $repository.releases\_url$, $repository.labels\_url$, $repository.notifications\_url$ and 28 more) that are automatically documented by SpyREST. API developers can also customize the auto generated summary information using a wiki-like editor on SpyREST. This allows API developers to explain behaviors that cannot be derived from analyzing the individual example API calls. For example, users can edit the table of Query Parameters to explain any parameter that is not used in the example API calls or explain validation rules (such as required, interdependent fields) about the query parameters.

4.4.2   Maintaining API Documentation for Evolving APIs

Maintenance of RESTful API documentation is another use case of interest since the documentation for RESTful APIs need to evolve with the APIs to reflect the updated information. The 8 steps mentioned previously need to be repeated every time any of the API objects changes when a manual process is followed. Since APIs continue to evolve throughout their lifetime, manually maintaining their documentation incurs further costs or documentation can quickly become outdated.

SpyREST automatically updates the recorded information for each example API call, so re-running the API examples automatically updates the auto-generated documentation. To uniquely identify each example API call, the Spy computes a digital signature of each example API call based on its API version, resource, URL and custom description. To replay the example API calls, API developers can use automated scripts so that once an API example is scripted, it can be run repeatedly. Moreover, the automated scripts can be written as functional tests where the generation of API documentation becomes a side-benefit of running the tests. Running the automated scripts to exercise the API examples frequently can prevent out of date API documentation as shown on

59

the previous use case.

Listing 4.1 shows an example script written using the Ruby based test framework Minitest. In this example script, on line 1 the class `Github` is setup so that the Spy proxy server is used to make example API calls. `HTTParty` used on line 3 is a third-party library for making RESTful API calls. Then the required headers are setup for using GitHub API on line 6. Finally, we setup the proxy connection to SpyREST server. Then, on lines 17-25, we define an example API call as an automated test script to generate documentation for the `GET /notifications` API action with three query parameters `all,` `participating`, and `since`. The results of the API call can be used to test against expected results. Using this script SpyREST will generate the documentation for this API and auto-update the documentation by rerunning the script anytime the API changes while still providing test coverage. If custom edits are made to the summary section of the SpyREST generated documentation, then a re-run of the same API example will only update the example section of the documentation with new data leaving the summary section unchanged. Thus, the custom edited content sustains over multiple runs of the API examples.

4.5   Discussion

With SpyREST we have aimed to provide a solution to RESTful API documentation. A manual approach is both expensive and susceptible to human errors. SpyREST offers several benefits over existing API documentation tools. We performed a preliminary evaluation of SpyREST by generating documentation of 25 RESTful API actions from 3 different APIs providers (Github.com, KISSMetrics.com, LiquidPlanner.com) using 272 lines of test code[3]. Comparing SpyREST generated documents against the official documentation (manually generated) we found that the official documentation for 5 of the 25 API actions had outdated response information that did not match with the actual API responses as captured by SpyREST. This is an expected advantage of automation, since updated documentation can be produced by replaying existing scripts instead of

---

[3]`https://github.com/smsohan/spyrest_examples`

Listing 4.1: Example API call using SpyREST

```ruby
class Github

  include HTTParty

  base_uri 'https://api.github.com'
  headers('Accept' => 'application/vnd.github.v3+json',
    'User-Agent' => 'curl/7.37.1',
    'content-type' => 'application/json',
    'Authorization' => "token GITHUB_API_TOKEN'"
  )

  host, port = 'spyrest.com', 9081
  http_proxy host, port

end

describe 'Notifications' do

  it 'List all notifications for the current user, where they are
    participating, since a time' do

    response = Github.get '/notifications?all=true&participating=
      true&since=2014-01-01T00:00:00Z'
    assert_equal 200, response.code
  end

end
```

requiring any manual efforts.

Using automated tests to generate documentation serves a dual purpose for maintaining a test suite since the documentation of the RESTful API becomes an artifact from the execution of tests. This is a unique advantage of SpyREST over other API documentation tools that translate manually edited API descriptions into API documents instead of executing actual API calls.

Most general purpose API documentation tools such as JavaDoc extract documentation based on formatted comments from the source code. Since comments are not executable code, manual effort is required to keep the comments updated as the underlying API changes. RESTful API documentation typically includes additional information about HTTP headers, request and response payloads as well as example API calls that are not natively supported by general purpose API documentation tools. Also, API documentation tools for library APIs are often programming language specific. SpyREST overcomes these limitations of code comments based API documentation tools by leveraging a HTTP proxy server where executable API examples are converted into API documentation that can be used to document RESTful APIs independent of independent of what programming languages are used to implement the RESTful APIs..

SpyREST offers unique features compared to other RESTful API documentation services. Swagger[4] and Blueprint[5] are two software as a service tools for RESTful API documentation. The key difference between SpyREST and these tools is in the underlying approach used. These tools require the API developers to follow custom vendor specific API specification formats to describe the API objects. Once the APIs are described, the tools can auto-generate the documentations for the APIs. There is no automated tool support to generate the required API description in the vendor specific formats. Thus manual effort is required to produce the required API specifications and maintain it over time. SpyREST does not require any intermediate API specification as data is sourced for the documentation based on live API calls. Swagger and Blueprint do not support a self-hosted service to document RESTful APIs whereas SpyREST allows a self-hosted

---

[4]https://github.com/swagger-api/swagger-spec/blob/master/versions/2.0.md
[5]https://github.com/apiaryio/api-blueprint

service to document RESTful APIs that cannot be externally shared for confidentiality issues.

The technology choice behind for our implementation of SpyREST components is based on our past experience of using these tools but the concepts behind SpyREST is not dependent on the aforementioned technology choice. As a proof of concept, SpyREST only analyzes JSON based API request and response payloads.

SpyREST is a proof of concept implementation for a novel RESTful API documentation technique. We identify the following limitations. SpyREST has yet to be evaluated. We plan to evaluate SpyREST with a large number of RESTful APIs as well as conduct a user evaluation involving API developers to understand the strengths and weaknesses of our tool. The current implementation of SpyREST only works on JSON based RESTful APIs. SpyREST needs to be enhanced to support other data types such as XML, and CSV. SpyREST, like other HTTP proxy servers, performs an intentional "man in the middle attack" to intercept HTTP traffic for example API calls over SSL/TLS. We consider the security impact of this approach to be low since the data is commonly meant to be included in documentation for public use. Otherwise, API developers can use the self-hosted mode and provide SpyREST with the required SSL keys. Web APIs often include a set of commonly used error responses across a variety of API endpoints. Additional work is required on SpyREST to automatically provide a summary of error responses based on example API calls. SpyREST also needs to be extended to support auto-linking between different API endpoints so that API developers can navigate between related API documentations.

4.6   Related Work

Several papers have studied API documentation to understand and recommend requirements for effective API documentation tools. Robillard recommended the following requirements for API documentation: include good examples, be complete, support many complex usage scenarios, be conveniently organized, and include relevant design elements [5, 6]. Kuhn et al. discussed the importance of examples in API documentation as a key recommendation based on a survey of

software developers using APIs [7]. They identified trustworthiness, confidentiality, and limiting information overload as other key recommendations for API documentation. Hoffman et al. recommended providing executable test cases with API documentation [8]. Nasehi et al. recommended the use of wiki-like editing features for online API documentation to foster collaboration [9]. Parnin et al. and Chen et al. also recommended incorporating collaborative features with API documentations [10, 11]. Stepalina identified several benefits of using software as a service model for API documentations as follows: cost effective yet powerful, platform agnostic and high accessibility, improved document quality, content re-use, automated tools, and organization of robust and scalable documentation process [12].

Several papers have discussed the topic of RESTful API documentation. Espinha et al. found most RESTful API documentations to be less reliable because they are manually generated [2]. Danielsen et al. proposed a vocabulary called Web Interface Language (WIfL) for documenting RESTful APIs so that different APIs can be described using a standard terminology [3]. RESTdesc, RDDL and hRESTS have been proposed as custom specifications to describe RESTful APIs [13, 14, 15]. For RESTful API documentation, Myers et al. recommended providing a consistent look-and-feel and an overall map comprising of both text and diagrams, providing a browsing experience with breadcrumb trail following a hierarchy, an effective search interface, providing example code and a way to exercise the examples without writing code [16].

Our work on SpyREST is a result of following the recommendations as found on the aforementioned related work as well as critically analyzing existing API documentation tools such as JavaDoc, Swagger, and Blueprint, to identify the unmet requirements. Based on our analysis, we incorporated the following recommendations as requirements for SpyREST: automated RESTful API documentation, executable examples, consistent navigation and look and feel, software as a service model, wiki-like editing, automatic stripping of confidential data, reducing information overload, and in place collaboration.

4.7   Conclusion

In this paper we have presented SpyREST as a prototype tool to demonstrate an automated solution to RESTful API documentation. Our main contribution is a novel tool that leverages a HTTP proxy server to intercept and automatically extract API documentation from example API calls. SpyREST provides an integrated tool for RESTful API documentation by providing features for automated generation, customization, maintenance, collaboration and executable examples under a single cloud based software as a service platform. By automating the repeated parts of the RESTful API documentation process, SpyREST provides a more effective alternative to the manual process. API developers can use SpyREST to save time and costs for generating and maintaining RESTful API documentation. In the future we plan to conduct a quantitative evaluation by using SpyREST to auto generate documentation for a large set of RESTful APIs. We will also conduct a qualitative user study involving API developers to collect feedback about the strengths and weaknesses of SpyREST.

4.8   References

[1] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

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

[3] P. Danielsen and A. Jeffrey, "Validation and interactivity of web API documentation," in *Proc. of International Conference on Web Services (ICWS)*.   IEEE, 2013, pp. 523–530.

[4] S. Sohan, C. Anslow, and M. Frank, "SpyREST: Automated RESTful API documentation using HTTP proxy server," in *Proc. of IEEE/ACM International Conference on Automated Software Engineering*, 2015.

[5] M. Robillard, "What makes APIs hard to learn? the answers of developers," *Software, IEEE*, vol. PP, no. 99, pp. 1–1, 2011.

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

[7] A. Kuhn and R. DeLine, "On designing better tools for learning APIs," in *Search-Driven Development - Users, Infrastructure, Tools and Evaluation (SUITE), 2012 ICSE Workshop on*, 2012, pp. 27–30.

[8] D. Hoffman and P. Strooper, "API documentation with executable examples," *Journal of Systems and Software*, vol. 66, no. 2, pp. 143 – 156, 2003.

[9] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns, "What makes a good code example?: A study of programming Q&A in StackOverflow," in *IEEE Proc. of International Conference on Software Maintenance (ICSM)*, 2012, pp. 25–34.

[10] C. Parnin and C. Treude, "Measuring API documentation on the web," in *Proc. of the International Workshop on Web 2.0 for Software Engineering*, ser. Web2SE '11. ACM, 2011, pp. 25–30.

[11] C. Chen and K. Zhang, "Who asked what: Integrating crowdsourced faqs into api documentation," in *Proc. of the International Conference on Software Engineering Companion*, ser. ICSE Companion 2014. ACM, 2014, pp. 456–459.

[12] E. Stepalina, "SaaS support in software documentation systems," in *Proc. of Central and Eastern European Software Engineering Conference (CEE-SECR)*, 2010, pp. 192–197.

[13] R. Verborgh, T. Steiner, D. Van Deursen, S. Coppens, J. G. Vallés, and R. Van de Walle, "Functional descriptions as the bridge between hypermedia APIs and the semantic web," in *Proc. of the Third International Workshop on RESTful Design*, ser. WS-REST '12. ACM, 2012, pp. 33–40.

[14] J. Mangler, P. Beran, and E. Schikuta, "On the origin of services using riddl for description, evolution and composition of restful services," in *Proc. of IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, 2010, pp. 505–508.

[15] J. Kopecky, K. Gomadam, and T. Vitvar, "hrests: An HTML microformat for describing restful web services," in *Proc. of International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*, vol. 1. IEEE, 2008, pp. 619–625.

[16] 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," *J. Organ. End User Comput.*, vol. 22, no. 1, pp. 23–51, 2010.

Chapter 5

Automated Example Oriented REST API Documentation at Cisco

5.1 Abstract

Generating and maintaining an up-to-date API documentation is a challenging problem for evolving REST APIs. At Cisco, we've used SpyREST, an automated REST API documentation tool, via our functional tests to solve this problem with one of our APIs for a cyber security application over the past eighteen months. Using this approach, we've avoided the need for extensive manual effort by leveraging our test code to also generate a continuously updated API documentation as the API evolved. Our always-updated API documentation has helped creating a fast feedback loop between the developers and QA engineers. The findings from this paper can be used by practitioners to introduce automation to reduce the manual effort associated to their REST API documentation process.

5.2 Introduction

Researchers identified the documentation of APIs as both the primary source of information as well as the key obstacle for API usability [1]. To this regard, researchers have identified the qualities of "good API documentation" as follows: complete, correct, includes thorough explanations and code examples, provides consistent presentation and organization [1, 2]. In our previous work, we introduced a novel technique and SpyREST, an implementation, based on an HTTP proxy server to automatically intercept example REST API calls and synthesize the data to produce REST API documentation to meet the aforementioned qualities [3].

The primary motivation of this paper is to share the results of an evaluation of SpyREST and

it's underlying REST API documentation technique based on its production use at Cisco. While REST APIs are published and documented by a large number of today's Internet based software companies, we see a lack of published literature on the topic of reusable and automated documentation techniques for evolving APIs. By sharing our lessons learned from a production use of a new technique in an industry setting we aim to help practitioners and researchers with a practical approach to generate and maintain example oriented REST API documentation.

The team at Cisco was tasked to develop a REST API such that it could frequently evolve to support incremental releases and addition of new features with backward compatibility as needed by the business. We compared several existing REST API documentation tools to evaluate the features against the documentation requirements for this evolving API. SpyREST was selected because the team found it to provide better automation support to achieve the requirements compared to the alternatives. We expect the information on this paper to be of use for other evolving REST APIs that need to maintain an up-to-date documentation without duplicating developer effort.

SpyREST is being used in production at Cisco for the documentation of a commercial REST API of a cloud based Cyber security product. The first author of this paper is affiliated with the team behind the product. It provides us with a unique opportunity to analyze the impact of the industry adoption of a tool developed in research. Production usage over an eighteen month period also allows us to understand the problem and opportunities presented by SpyREST in depth.

Our core contributions from this case study are as follows:

- **Test driven REST API documentation**. For practitioners, we discuss a reusable technique for producing example oriented REST API documentation as a byproduct from automated API test code.

- **Evolution of API documentation**. For practitioners, we discuss a viable process to support evolution of API documentation as the API evolves without duplicating effort.

- **Evaluation of SpyREST at Cisco**. We present an industrial evaluation of SpyREST

69

and its underlying technique of generating API documentation by intercepting API examples based on our case study at Cisco. Using this as evidence, researchers can extend the interception based the API documentation technique to improve tool support for the documentation of other forms of APIs beyond REST.

The remainder of this paper is organized as follows: in the following section we present related work to discuss the current state of research on REST API documentation. Then, we provide a brief overview of our REST API documentation technique and the tool, SpyREST, followed by a case study of using SpyREST at Cisco. Then, we discuss our lessons learned and the limitations of this case study.

5.3 Related Work

5.3.1 API Usability and Documentation

Several papers in the existing literature have focused on identifying the characteristics that make an API usable based on case studies. Robillard studied API usability by surveying 83 software developers at Microsoft [4]. They found that 78% of the survey participants read API documentation to learn the APIs, 55% used code examples, 34% experimented with the APIs, 30% read articles, and 29% asked colleagues. Robillard et al. found that the most severe API learning obstacles are related to the API documentation. They suggested the following requirements as must-have for API documentation: include good examples, be complete, support many example usage scenarios, be conveniently organized, and include relevant design elements. Myers et al. also recognized the documentation as a key component for API usability and suggested using examples in the documentation to answer API related questions [5]. Zibran et al. found that 27.3% of the reported bugs are API documentation bugs studying repositories for 562 API usability related bugs from five different projects [6]. Scheller et al. provided a framework for measuring API usability based on the number and types of different objects and methods that the API provides [7].

Kuhn et al. performed a user study with 19 professional software developers to understand re-

quirements for tool development to support API learnability [8]. They recommended the following as requirements for API documentation: trustworthiness, confidentiality, lack of information overload and the need for code examples as first-class documentation artifacts. Shi et al. recommended API documentation tools to support editors for custom content to provide simple ways to include API usage examples without syntax errors [9]. Ko et al. found that thorough introductions to the concepts, standards and ideas in API documentation are a prerequisite for API usability [10].

Because REST APIs are a category of APIs, the strong relationship between the documentation and usability of the API as discussed in the aforementioned papers also applies to the context of REST APIs.

5.3.2   Usage Examples in API Documentation

Several authors introduced tool support for including usage examples with API documentation. Hoffman et al. recommended using executable examples in API documentation [11]. They introduced the Roast test as tool support to combine prosaic descriptions of Java APIs along with executable code examples. Montandon et al. developed APIMiner as a search tool for Java APIs and recommended providing production-like API usage examples in the API documentation [12]. Zhu et al. developed UsETeC to extract API usage examples by automatically synthesizing JUnit test code of the APIs [13]. Stylos et al. presented Jadeite to provide developers with faster access to relevant API documentation by using placeholders for API methods [14].

Several authors presented techniques for linking official API documentation with crowd-sourced API usage examples that is otherwise fragmented. Nasehi et al. recommended mining knowledge repositories such as StackOverflow and developer forums should be considered for retrieving useful code examples [15]. Parnin et al. found that examples of 87.9% of all jQuery API methods are found by searching software development blogs and forums [16]. Wu et al. presented an Eclipse plugin called CoDocent that can automatically find code examples using various online code search engines and link with the relevant official API documentation [17]. Chen et al. presented a technique to automatically link official documentation with crowd-sourced documentation

by recording the API related web searches that are performed by developers [18]. Dagenais et al. presented a tool called RecoDoc to link code-like elements from API mailing lists and developer forums with their corresponding code elements [19]. Treude et al. presented a machine learning based technique called SISE to augment useful information from StackOverflow to API documentation by using text similarity of API elements and StackOverflow content [20].

We observed that the research on usage examples in API documentation related tools have focused on local APIs such as Java library APIs. While local APIs are defined by programming language specific constructs such as classes, and methods, REST APIS are defined by HTTP terminology such as HTTP request, response, and headers. We found a lack of published work on the tool support for including usage examples with REST API documentation.

5.3.3 REST API Documentation

The concept behind RESTful API was introduced by Fielding [21]. Maleshkova found that most REST APIs are manually documented which results in API underspecification, and a lack of support for common tasks and reusable tools [22]. Myers et al. performed a user study on the usability of a complex API for enterprise SOA [2]. They recommended providing a consistent look-and-feel with explanation for the starting points and an overall map comprising of both text and diagrams, providing a browsing experience with breadcrumb trail following a hierarchy, an effective search interface, providing example code and a way to exercise the examples online without writing code. In a previous case study, we found the documentation of REST APIs are generated manually or using bespoke tools [23].

Several authors have suggested machine readable specification languages for REST APIs that can be used to transform into API documentation and auto generated API client code such as WADL (Web Application Description Language), RIDDL, hRESTS, RESTdesc, and WIfL (Web Interface Language), OmniVoke [24, 25, 26, 27, 28, 29]. Polak proposed a specification format for REST API using the Model-Driven Architectural principle [30]. In addition to the existing literature, several REST API description languages have been proposed by industry practitioners such

as RAML, API Blueprint, and Swagger[1]. Listing 5.1 shows a fragment of REST API specification following swagger [2].

Listing 5.1: Example Swagger REST API Specification

```
1  basePath: /v1
2  schemes:
3    - http
4  consumes:
5    - application/json
6  produces:
7    - application/json
8  paths:
9    /pets:
10     get:
11       summary: List all pets
12       operationId: listPets
13       tags:
14         - pets
15       parameters:
16         - name: limit
17           in: query
```

The primary advantage of these specification languages is code generation and automatic transformation into REST API documentation. For some strongly typed programming languages, such as Java, it is possible to extract a schema representation of the API objects from the code. However, for dynamically typed languages, such as Ruby, the lack of type information prevents automated API specification. As a result, manual effort required from the developers to maintain a separate API specification. Even when API schema automatically extracted, API developers need to manually maintain usage examples because the examples don't exist in the API code.

---

[1] http://raml.org/ https://apiblueprint.org/ http://swagger.io/
[2] https://github.com/OAI/OpenAPI-Specification

Table 5.1: REST API Documentation

Desirable Property	Current State
Detailed introduction	Manually edited contents are commonly used.
Includes Examples	Commonly include manually generated API examples.
Executable Examples	Bespoke tooling is used to provide API explorers.
Automated	Tools rely on manually written specifications.
Consistent Presentation	Includes access information, resources, actions, request and response structures and API examples.

Table 5.1 contrasts the current state of tool support for REST API documentation against a set of properties that researchers identified as required for API usability. In summary, practitioners and researchers have attempted to solve the problem of REST API documentation by proposing specifications for describing REST APIs. Manual work is needed by REST API developers to generate and maintain the API specifications. These specification formats describe the structure of different API elements (the syntax), but lacks support for automatically generated executable API usage examples with realistic data (the semantics). We found a lack of published papers on the effectiveness of the aforementioned specification languages in an industry setting. In this research, we aim to fill this gap by sharing our experience of using an automated REST API documentation technique in the industry.

5.4 REST API Documentation Tool Selection

5.4.1 Requirements

Tulach summarized developer feelings about writing API documentation as follows: **"Everyone who ever tried to convince a developer to write documentation knows how hard that is. As a result, the general feeling is that developers don't like and are not able to provide documentation for their own code."** [31]. At Cisco, a focused group was formed involving five API

74

developers and two QA engineers to select an appropriate REST API documentation tool that they would actually use. The group collected the following requirements: **R1.** support multiple versions, **R2.** support multiple environments, **R3.** easy to maintain, **R4.** include executable API usage examples, **R5.** use familiar tools, and **R6.** allow customization.

5.4.2 Evaluation of Tools

A list of REST API documentation tools were selected for "spikes", a practice used by the software engineers to better understand how a tool fits a problem in context. The API developers and QA engineers shared opinions based on their spikes about three different tools, Swagger, API Blueprint, and SpyREST. Swagger and Blueprint were researched because of their maturity and industry adoption information as found through online searches. The authors of this paper previously developed SpyREST based on their past experience outside the context of this specific Cisco API. The focused group participated in the "spike" and discussed the pros and cons of each selected tool. Table 5.2 summarizes the findings:

As shown in Table 5.2, the verdict to use SpyREST was based on the observation that using Swagger or API blueprint required us to manually maintain the REST API documentation as a separate static artifact from the API related code. SpyREST was found to provide better support for R3, R4, and R5 over Swagger or API Blueprint as the developers could auto-generate the documentation from the functional test suite instead of "writing separate documentation" that are required to be written and maintained with the code. A round of informal feedback on SpyREST was collected after the first production release of the API and the team decided to continue using SpyREST.

5.5 SpyREST – Overview

We provide a brief overview of SpyREST, the REST API documentation tool and the underlying technique used in this case study [3, 32]. At the heart of the technique is a pass-through HTTP

Table 5.2: Comparison of Tools against the Requirements

Req.	Swagger	API Blueprint	SpyREST
R1 (version)	Separate file / version	Separate file / version	API example / version
R2 (env.)	Separate file / environment	Separate file / environment	Automated
R3 (maint.)	Manually maintain separate artifact from code	Manually maintain separate artifact from code	Derived from automated tests
R4 (exe. exampl.)	Automated (Web UI and cURL)	Manual	Automated (cURL)
R5 (familiarity)	Swagger API Spec and code inspection	Markdown	Automated tests
R6 (custom.)	Manual	Manual	Automated tests and Manual

proxy server which acts as an interceptor between an API client and the API. This allows the proxy server to inspect the raw HTTP request and response data from the example API calls. However, for usable API documentation the raw HTTP data needs to be further processed and enriched with meta data. For example, given the following HTTP request and response data from an example API call to create a Blog post:

```
Request Verb: POST
Request URL: /v2/posts
Request Headers:
  Content-Type: application/json
  Authorization: Basic dXNlcjpwYXNzd29yZA==
Request Body:
  {
    "title":"My New Blog post",
```

```
    "content": "This is a new blog post"
  }


Response Headers:
  Location: "/v2/posts/1"
  host: "blog.example.com"
```

A series of transformation needs to take place to produce usable API documentation. The proxy server used by SpyREST is customized to record and synthesize such example API calls. The transformation process involves the following analyzers:

- **API version analyzer**. The version analyzer automatically infers the API version used by the example API call. From the aforementioned example, the version analyzer auto-detects the API version as "v2" based on the URL.

- **API resource analyzer**. To generate a hierarchical representation of the API elements, it's important to group multiple API actions that correspond to a single API resource under a single hierarchy. API resource analyzer automatically detects *post* as the API resource for this example.

- **API action analyzer**. The API action analyzer automatically infers the API action on a resource from example API calls. For the given example, the action analyzer identifies $POST/v2/posts$ as the API action.

- **API query parameter analyzer:**. The query parameter analyzer records each query parameter that is used by the example API calls and automatically infers the data types such as integer, string, timestamp, etc.

- **API request header analyzer**: The request header analyzer removes standard authorization credentials from the API documentation. For the given example, the API requests header analyzer transforms the *Authorization* header's value as $Authorization : Basic FILTERED$.

- **API body analyzer**: The body analyzer captures the request and response bodies and infers the structure of the body as an object with auto-detected field names and data types. The request body analyzers produces the following information from the given example for API documentation:

Field	Data Type	Example
title	String	My New Blog post
content	String	This is a new blog post

- **API response header analyzer**: The response header analyzer automatically removes noisy headers such as (Server, Pragma, etc.) from the documentation.

- **Custom content analyzer**: SpyREST automatically infers a human readable description for each API action. For the given example, the auto detected description is as follows: *CreateaPost*. The custom content analyzer allows API developers to override the auto-detected attributes such as the API version, resource, action and description by using a set of SpyREST specific HTTP headers.

Fig. 5.1 shows SpyREST screenshots of the production REST API documentation at Cisco. Fig. 5.1a shows a fragment of the manually written overview information juxtaposed to the auto-generated index of the API resources. Fig. 5.1b shows the output of the different analyzers within SpyREST proxy server that automatically detects the version (v1), API resource (Computer), action (PATCH /v1/compouters/...), and the structure of the response fields from the single example API call. Fig. 5.1c shows the transformed example API request headers, body, and an executable cURL command that can be used to exercise the API call. Fig. 5.1d shows the API response headers and body for the example.

To summarize, SpyREST provides tool support to improve the process of REST API documentation with usage examples, automatic updates, executable examples and a consistent hierarchical representation of the API.

Cisco AMP for Endpoints API

Home > Versions > v1 Resources

Overview

Making Requests

Wherever possible, we suggest using the `Accept-Encoding: gzip` header.

All API requests must use HTTP Basic Auth.

Generating Client ID and API Key:

- Log into https://console.amp.cisco.com (N.A.) or
  https://console.eu.amp.cisco.com (E.U.)
- Go to the **API Credentials Page** from the **Accounts** dropdown menu.
- Click on the "New API Credential"
- In the popup, enter an application name and select a scope, either
  'Read-only' or 'Read & Write' and click 'Create' button to generate a
  Client ID and API key.
  - 'Read-only' scope gives you only read access to the API endpoints.
  - 'Read & Write' scope gives both read and write access.
- Be sure to store your API key in a safe place, it is only shown to you
  once.

Resources

Select an API Resource for version `v1` from the following list

- Computer
  GET /v1/computers
  GET /v1/computers/{:connector_guid}
  GET /v1/computers/{:connector_guid}/trajectory
  PATCH /v1/computers/{:connector_guid}
- Computer Activity
  GET /v1/computers/activity
- Event
  GET /v1/events
- Event Type
  GET /v1/event_types
- File List
  GET /v1/file_lists/application_blocking
  GET /v1/file_lists/{:file_list_guid}
  GET /v1/file_lists/simple_custom_detections
- File List Item
  GET /v1/file_lists/{:file_list_guid}/files
  GET /v1/file_lists/{:file_list_guid}/files/{:sha256}

(a) API Overview in SpyREST

Home > Versions > v1 Resources > Computer > PATCH /v1/computers/{:connecto

PATCH /v1/computers/{:connector_guid}

Description

Hide Response Fields

Name	Type	Description
version	String	
metadata.links.self	String	
data.connector_guid	String	
data.hostname	String	
data.active	Boolean	
data.links.computer	String	
data.links.trajectory	String	
data.links.group	String	
data.connector_version	String	
data.operating_system	String	
data.internal_ips	Array	
data.internal_ips[]	String	
data.external_ip	String	
data.group_guid	String	
data.network_addresses	Array	
data.network_addresses[].ip	String	
data.policy.guid	String	
data.policy.name	String	

(b) SpyREST Analyzed API Structure

79

Examples

Moves computer to a group with given connector_guid and group_guid

Request

Requires Authorization

PATCH /v1/computers/ad29d359-dac9-4940-9c7e-c50€

Headers

accept: application/json
content-type: application/json
authorization: Basic FILTERED
content-length: 53

cURL Edit, then copy and paste on your terminal

curl -X PATCH \
-H 'accept: application/json' \
-H 'content-type: application/json' \
-H 'content-length: 53' \
--compressed -H 'Accept-Encoding: gzip, deflate'
-d '{"group_guid":"b077d6bc-bbdf-42f7-8838-a0605
-u YOUR_API_CLIENT_ID \
'https://api.amp.cisco.com/v1/computers/ad29d359

Body

{"group_guid":"b077d6bc-bbdf-42f7-8838-a06053fbc

Response

Shortened for readability

x-ratelimit-limit: 1000
x-ratelimit-reset: 3591
x-ratelimit-remaining: 975
x-frame-options: SAMEORIGIN
x-ratelimit-resetdate: 2016-08-09T05:26:20Z
status: 202 Accepted
transfer-encoding: chunked
content-type: application/json; charset=utf-

{
  "version": "v1.0.0",
  "metadata": {
    "links": {
      "self": "https://api.amp.cisco.com/v1
    }
  },
  "data": {
    "connector_guid": "ad29d359-dac9-4940-9
    "hostname": "Demo_CozyDuke",
    "active": true,
    "links": {
      "computer": "https://api.amp.cisco.co
      "trajectory": "https://api.amp.cisco.
      "group": "https://api.amp.cisco.com/v
    },
    "connector_version": "4.1.7.10201",
    "operating_system": "Windows 7, SP 1.0"
    "internal_ips": [
      "87.27.44.37"
    ],
    "external_ip": "93.111.140.204",

(c) SpyREST Recorded API Request   (d) SpyREST Recorded API Response

Figure 5.1: Screenshots of the REST API Documentation from Cisco using SpyREST

5.6 Case Study: SpyREST at Cisco

5.6.1 Methodology

For this case study, we use a specific REST API within Cisco. SpyREST has been used over the past eighteen months and is being used at the time of writing this paper to document the REST API for a cloud delivered cyber security application at Cisco named Advanced Malware Protection for Endpoints. The API allows the customers of the cyber security application to extract and modify security related data specific to their businesses from the cloud to facilitate automation and custom third-party integration. For example, some customers use the API to automatically create an investigation ticket on their ticketing system for each malware detection event.

The REST API is documented using SpyREST via automated tests. To evaluate the outcome of the documentation process using SpyREST, we've incorporated stakeholder feedback as well as collected metrics from the various log files associated with the development and deployment of API docs and the actual API changes. 25 API stakeholders from Cisco were invited via email to provide free form feedback about the API documentation and its underlying process via emails to the first author of this paper.

5.6.2 Context

The Stakeholders

The REST API has several stakeholders as follows: customers, API client developers, customer support team, product management, API developers, and quality assurance engineers. Customers and potential customers are interested in an API so that custom tooling and business specific integration can be performed. API client developers are often employed by the customers to implement API integrations. At present, over a hundred unique customers use the API at least once a month as per the API usage logs.

Inside Cisco, the stakeholders are the customer support team, product management team, API developers (including the first author) and quality assurance engineers. At present, a total of 25

81

people are involved as the Cisco stakeholders in this case study.

Participants

We received a total of 9 feedback emails comprising of the following: one project manager (PM), one customer support engineer (SE), five API developers (D1-5) and two QA engineers (QA1-2). Developer D2 has 5 years, and the rest of the participants each have more than 10 years of software industry experience.

API Properties

At present, the API is currently available under two different versions, v0 and v1. The API is also served from three different Cisco managed cloud environments for North America, Europe, and Asia Pacific. Additionally, the API is shipped with a private cloud appliance, where customers can host the software on-premise. Even though the environments are designed to eventually serve the API with identical features, the deployment of the API to each environment is independently managed. The underlying application, introduced in 2010, uses Ruby on Rails [3] web framework, and has a total of 120K lines of custom code with a code to test ratio of 1:1.3.

In the latest version, v1, the API has nine different API resources, and a total of twenty API actions, with multiple variants per action, to be performed on those resources. For example, the API has a resource named "Computer," and an action $PATCH/v1/computers/: connector_guid$ for updating a computer. The older version, v0, has 6 resources and 10 API actions. Currently, v1 is backward compatible with v0, but backward compatibility is not guaranteed with future versions.

API Evolution

The API was first launched with version v0 in February 2015, and a fully compatible version v1 was released in May, 2016. The monthly release notes have mentioned at least one API related change for 12 times out of a total of 18 publicly announced releases of the software. Internally, the automated functional tests for the API documentation have changed 95 times by 6 contributors (as
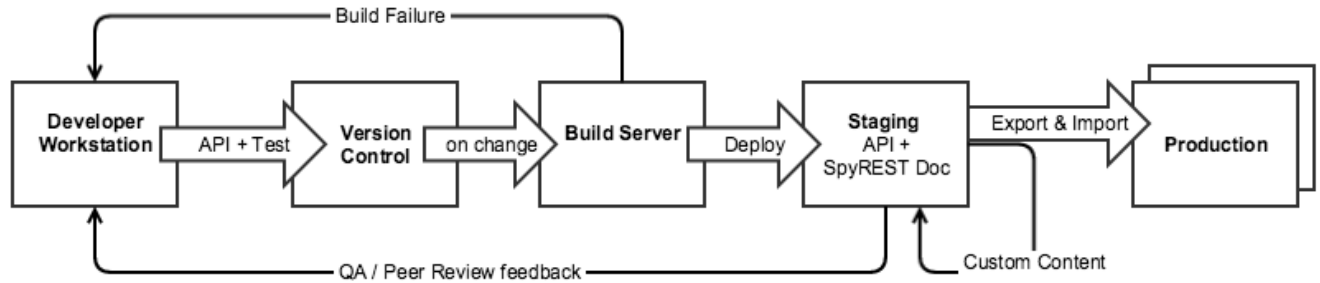
---

[3] http://rubyonrails.org

Figure 5.2: API Development workflow using SpyREST at Cisco

per the version control logs). The actual number of internal API evolution before they are released is unknown since they often happen several times a day, and don't always involve a change in the functional tests. The version numbers v0 and v1 were left unchanged to release backward compatible new features and bug fixes. At present, the API serves 80,000 API calls on an average per day (as per the API call logs) and is used by over a hundred enterprise customers that are globally distributed. The customers use the SpyREST generated documentation as the sole information source for learning the API features. The actual number of API client developers is not known at this time since multiple API client developers can be involved in writing an API client application for the customers.

5.6.3   The API Development Process

Documentation of a new API Action

The REST API developers and quality assurance engineers of the cyber security product at Cisco use SpyREST to generate the API documentation throughout the lifecycle of an API from the developer workstation to a production instance as shown in Fig 5.2. When a new API action is introduced, the API developers write one or more functional tests against the API to show an intended usage example. If the functional test is run using SpyREST proxy server, it automatically generates the API documentation. Listing 5.2 provides an example of an automated test fragment from the case study that renders part of the documentation as shown in Fig. 5.1c 5.1d.

Listing 5.2: Functional Test Code

```
1   context 'v1' do
2     describe 'Computer' do
3       it 'Moves computer to a group with given connector_guid and group_guid'
            do
4         response = API.patch("/v1/computers/#{connector_guid}", {
5           headers: 'x-spy-rest-action' => '/v1/computers/{:connector_guid}',
6           body: { group_guid: group_guid }.to_json
7         })
8         expect(response.code).to eql(202)
9         #more assertions
10      end
11    end
12  end
```

This example code is written using RSpec[4], a Ruby based test framework. Line 1 mentions the API version, line 2 mentions the API resource of interest, and line 3 shows a human readable description of the test. On lines 5-7, an example HTTP patch API request is made using the method *API.patch*. Then, an example assertion is added on line 8. the *API* class has capabilities to make the *patch* HTTP request over an HTTP Proxy server. While using the proxy server, the *API* class also sends the test description "Moves computer to a group with given *connector_guid* and *group_guid*" with a SpyREST specific header to automatically map test description as the description of the API usage example. As discussed earlier, the SpyREST analyzers can inspect the HTTP request and response information with the custom headers to produce usable REST API documentation from this automated test code.

This practice of using tests for documentation was mentioned as an advantage by 4 of the 9 participants. For example, QA2 mentioned the following: **"SpyREST was a useful tool for us... as the (API) endpoints were documented using unit tests, (API) developers would write**

---

[4]http://rspec.info

84

the documentation for a similar audience (API client developers).” The PM mentioned the following feedback about the executable API examples in the documentation: **“The embedded examples are great for allowing users to try API calls quickly and easily to get a feel for how the API works.”**

The next step in the lifecycle of the API documentation is in the automated build server where the test suite is executed against a deployed API on a staging server through a SpyREST proxy. This promotes the REST API documentation from the developer workstation to a shared instance that is used by the quality assurance engineers, product management, and the peer developers to verify against the acceptance criteria. This often results in several loops between the developer workstations and the staging servers until the API and its documentation meets the acceptance criteria. During the team retrospective following the API production release, the following comment was captured in the meeting notes by one of the QA engineers (QA1): **“Up-to-date documentation using SpyREST helped developing and collecting lots of input about the API in small pieces during the weekly meetings before production release”**. The API developers and quality assurance engineers add any custom content as needed to describe complex API concepts and prerequisites that aren’t captured in the automated examples.

The next step in the lifecycle of the API is a release to one or more of the production environments that customers use. With the production release of an updated API, its documentation is automatically promoted from the staging environment to production. The promotion of the documentation is a three-step process. First, the data from the staging server is exported in a portable archive file. Then, a transformation is performed to update staging specific data such as URLs, email domains, and hostnames within the archive to match the desired production environment. Finally, the transformed archive is published to production as a read-only artifact.

Documentation of a new Version for an Existing API Action

When API version v1 is launched, all the functional tests for v0 are run on a loop, once per version. This code reuse minimizes the effort required to document multiple versions of the same API action

when the API versions are backward compatible. The rest of the lifecycle steps follow similar process as the documentation of a new API action.

Documentation of an updated API Action

When an API update requires a new example, a new automated test case is written. For example, an API action was updated to receive a new query parameter to support an additional operation.

When an API update doesn't require a new API example to describe the change, the API documentation is automatically updated by the build server. For example, in the case study an API action was updated to include a new field with the response body and the API documentation was automatically updated without needing a change in the test code.

REST API Evolution

In the case study we have observed the evolution of the API and it's documentation is triggered by the following: new requirements, and internal reviews of the API. For example, the primary difference between v0 and v1 of the API is the addition of a new requirement to allow API client developers the features to modify API objects in v1 that only allowed read-only access on v0.

While the new requirements are developed, before they are released to production, the API undergoes a more frequent evolution, several times a day, triggered by internal reviews of the API by the peer API developers and the quality assurance team. The auto-generated API documentation using SpyREST is used in the internal reviews to suggest alternatives and verify API acceptance criteria.

API evolution before and after the production release of the API has two opposing forces, yet one helps achieve the other. After an API is published to production, we are unable to make any breaking changes without affecting customers. To support this feature, we found it important to be able to evolve the API frequently before the API is published. The always up-to-date API documentation with SpyREST has helped establish a quick feedback loop between the API development and QA teams throughout the life-cycle of the API.

5.7 Discussion

5.7.1 REST API Documentation from Test Code

API developers are required to write the tests for the APIs even if not used for documentation. At Cisco, we have learned that the documentation of the REST API from its functional test code to be a welcome side-effect. Developer D3 mentioned the following feedback: **"Using a set of functional tests to both ensure correctness of the API, as well as a source of documentation is an excellent strategy"**. When the data from the tests are intercepted to generate API documentation, API developers need to actively think about the API usage scenarios against a realistic setup. For example, one of the Cisco developers updated the test name for an API from "Finds computers that have connected with an IP for v0" to "Fetches list of computers that have connected to a given IP address" to better reflect an API action for documentation based on internal review. This enforces the need for a stable test suite since a failing or incomplete test suite also breaks the API documentation. Eight of the nine study participants mentioned the executable API examples as a key feature of SpyREST that helped them to review the details of the API without writing code.

Continuously updated documentation improves its correctness and verification of version compatibility as an API evolves. A breaking change in the API fails the underlying tests. For non-breaking changes, such as addition of new REST API objects, the test suite helps proving backward compatibility.

One limitation of this approach is that not all API tests are suitable to be used in the REST API documentation. For example, even though unhappy paths are commonly written in tests, they are seldom included in the documentation for brevity. API developers may need to select a subset of the tests that are run through SpyREST proxy server that are meaningful for documentation.

Based on our experience at Cisco, we recommend REST API developers to utilize the automated test suite to drive API usage examples in the API documentation using SpyREST. SpyREST can intercept REST API calls irrespective of the technology used to implement the API and the API test client because it works as an HTTP proxy. The approach shared in this paper can be

87

applied to document REST APIs within other organizations where the technology behind the API implementation may differ.

5.7.2 Maintaining Custom Content

Based on the feedback on SpyREST by Cisco developers, we've learned both the benefits and drawbacks of using a wiki-like editor for custom content with the API documentation. The wiki within SpyREST allows the API developers to add rich content comprising of proses, images, tables, code fragments, and web links, etc. to explain concepts that are required to understand the API. One developer feedback (D1) mentioned the following about the wiki: **"Good customization on the overview of each resource."** Developer D3 mentioned the following: **"Being able to edit the documentation in place, similar to a wiki works really well, effectively allowing an organization to crowd-source the job."** For example, at Cisco, we've added an overview explaining API access information, rate-limit, and common approaches to perform pagination within the API actions as custom content.

The custom content using a wiki has a drawback as it's maintained as a separate artifact within SpyREST. Developer D4 mentioned the following feedback: **"Tests are great for examples and showing lists of available endpoints and parameters, but there also needs to be some explanation of how to use the API in general."** Since the wiki-edited custom content lives outside the code repository, manual effort is needed to maintain the evolution of custom content when needed. One of the Cisco developers (D2) mentioned the following in the feedback about SpyREST: **"It is helpful for deployment if the content in the overview section on the home page can be auto-generated through SpyREST code."**

We identify this as an opportunity to continue further research to improve the collaborative editing of rich customized content with auto-generated API documentation such that the custom content can be put into the version control systems.

5.7.3 Handling Flexible API Elements

The API elements within REST are not restricted to follow a strict structure. For example, in the case study, we have a flexible API element of type "Event" to denote a malware detection event. The actual structure of the "Event" API element can be widely different based on the type of the event. For example, a file detection event contains several file specific data such as the fingerprint, name, path on disk, size. On the other hand, a network detection event contains network specific data such as the remote host, URL, IP, protocol. A single API call may return a list of "Events" of such different types. Incomplete list of API response attributes caused a confusion to a customer and the SE mentioned the following feedback: **"The only criticism I have is making sure that all of the returnable data that is available in the Response Fields, because I have found some instances where this data was missing"**. We've incorporated this feedback and changed SpyREST so that the "Event" API element is documented as a union of all the different attributes of the different event types (fingerprint, name, path on disk, size file, remote host, URL, IP, protocol) from the intercepted example API calls.

We have learned the main advantage is, merging the attributes of different types of API elements in an Array provides a list of all possible attributes that are observed in an API element. An API client developer reading the API documentation gets a complete picture of the API element. On the other hand, it may not clearly communicate the fact that only a subset of the documented attributes may be returned by an API call for each API element. In our case-study at Cisco, the API usage examples captured by SpyREST shows specific examples with different "Event" types to provide the context around the event types. We recommend practitioners to follow this pattern since it allows API client developers to understand the implied structure of flexible API elements with specific use cases.

5.7.4 API Documentation Life-cycle

We've learned that it is a critical feature that the API documentation follows the API throughout its life-cycle stages such as development, staging, and production environments for each available API version. In our case study at Cisco, we've achieved this requirement from the test suite that is maintained under the same version control as the REST API code. As a result, for any version of the API, corresponding documentation can be generated from test code. The PM provided the following feedback about the customer impact of this auto-updated API documentation: **"One of the great values of a self-documented API is that it's always current and up to date. This simple, but valuable fact is not lost on our customers who feel like companies often forsake documentation and leave it as an afterthought."**

We've also learned that when REST API documentation is published from a staging environment to production, the data from the API examples may need to be obfuscated and transformed. This need was mentioned on the feedback from developer D5: **"It would be cool if you could dynamically re-write the hostname within the rails app (SpyREST), so you could hit an internal server but show docs for a public one."**. This idea was incorporated within SpyREST to automatically rewrite all URLs and email domains within the documentation during the production release process. However, there may be other confidential information auto-captured by SpyREST that may need to be obfuscated. For example, at Cisco, one of our API examples show the user login name on a computer as an API response. To prevent the leakage of such confidential data, we run the tests against a sandbox API environment that serves dummy data. We suggest practitioners to incorporate this idea of continuous API documentation while working on APIs to improve collaboration and feedback.

5.7.5 API Changelog

We found the manual process of generating API changelog to be both error-prone and time consuming. For an evolving API, a changelog is the primary information source for existing API client

developers to learn about the API changes. To verify the changes during rapid development, we've found a changelog to be a necessary communication artifact between stakeholders. SpyREST can automatically detect API changes when API objects such as query parameters, headers, request, and response objects are introduced or removed between versions because it records the API objects for each version. We have used the data from SpyREST and version control history of the tests as interim changelogs before a production release.

For production release, we manually inspect the API documentation and the interim changelogs to write a new changelog for customers. The manually written changelogs show the high level API changes leaving out the details. For example, the v1 changelog includes the following "Move a computer from one group to another", but further details were absent in the changelog. The underlying data model within SpyREST can be utilized to bridge the API reference documentation with relevant changelog for each version. Future work needs to be done to extend automation support for API changelog generation for evolving REST APIs.

5.7.6 Cross-Referencing API Elements

The primary navigation offered by SpyREST automatically presents a hierarchical view of the API elements comprising of API version, resources, actions on resources, and specific examples for each action. While this allows API client developers to get a quick index into the API elements, it may not provide the conceptual cross-references between API elements. Developer D2 suggested following enhancement about the SpyREST generated API Index: **"Is it possible to manually order the resources on the API doc site? Right now, the ordering is ensured in the SpyREST code."**

We've learned that cross-referencing the API elements may provide useful alternative navigation experiences. For example, the API in the case study has the API resources "Group" and "Computer," where a "Computer" belongs to a "Group". This nesting relationship is not captured in the SpyREST generated navigation. Similarly, the "Group" resource is related to a "Policy" resource since a "Policy" is applied on a "Group." In the API documentation for both "Group"

91

and "Policy" resources there are references to one another in the API response example. This dependency relationship is not captured within the SpyREST generated API navigation. Future work needs to be carried out to automatically surface such dependency relationships among API elements to help the API client developers.

5.7.7 Extending beyond REST API Documentation

From this case study, we've learned that it is feasible to use interception to generate REST API documentation with examples. While an HTTP proxy server cannot intercept non-HTTP APIs, the core concept of interception can be applied to such APIs. For example, to document a local API, an interceptor can be written to intercept example API calls in memory to generate API documentation with examples.

We identify several benefits of this approach over publishing the existing unit test code as documentation. First, organizations may not allow publishing their test code to external API users due to intellectual property issues. Second, even if unit tests are published as API documentation, the API client developers may not be familiar with the unit test framework and external dependencies. Third, unit tests often use test specific code such as complex setup, tear down, stubs and mocks, that may not be useful in the documentation. Fourth, the use of interception technique transforms task of documentation to writing executable code, a welcome change for developers. Researchers can extend the interception based documentation technique on non-REST APIs to document such APIs with executable examples.

5.7.8 Limitations

The first author on this paper is a member of the team at Cisco where the case study is performed. While it provides us with a unique insights, it also introduces a confirmation bias. Our results may include a selection bias since the results are based on a single single set of REST APIs. While the aforementioned feedback shows that REST API documentation process is effective for us, we have a lack understanding of the impact of the documentation on the API client developers. This case

study needs to be replicated against REST APIs that are implemented by other organizations using different technologies to reduce our limitations.

5.8   Conclusion

In this research, we presented the results of using SpyREST at Cisco to maintain an up-to-date documentation with usage examples for an evolving REST API. Our primary findings provide an evidence that SpyREST can be used as a practical REST API documentation tool. By leveraging automated functional tests to document REST APIs, we have reduced human efforts with automation in the API documentation process while improving the test suite. We discussed techniques to deal with custom content, flexible API elements and including the documentation of the REST API with every life-cycle step to establish a quick feedback loop. We presented our case for extending the interception based API documentation technique beyond REST APIs. In future, we plan to improve SpyREST by incorporating the ideas from this case study. We will perform an experiment to evaluate the impact of usage examples on REST API client developers.

Acknowledgments

5.9   References

[1] M. Robillard, "What makes APIs hard to learn? the answers of developers," *Software, IEEE*, vol. PP, no. 99, pp. 1–1, 2011.

[2] B. Myers, S. 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," *J. Organ. End User Comput.*, vol. 22, no. 1, pp. 23–51, 2010.

[3] S. Sohan, C. Anslow, and F. Maurer, "Spyrest: Automated restful API documentation using an HTTP proxy server," in *Proc. of IEEE/ACM International Conference on Automated Software Engineering, ASE, Lincoln, NE, USA*, 2015, pp. 271–276.

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

[5] B. Myers and J. Stylos, "Improving API usability," *Commun. ACM*, vol. 59, no. 6, pp. 62–69, May 2016. [Online]. Available: http://doi.acm.org/10.1145/2896587

[6] M. F. Zibran, F. Z. Eishita, and C. K. Roy, "Useful, but usable? factors affecting the usability of APIs," in *Proc. of IEEE Working Conference on Reverse Engineering (WCRE)*, 2011, pp. 151–155.

[7] T. Scheller and E. Kühn, "Automated measurement of api usability: The API concepts framework," *Information and Software Technology*, vol. 61, pp. 145–162, 2015.

[8] A. Kuhn and R. DeLine, "On designing better tools for learning APIs," in *In Proc. of ICSE Workshop on Search-Driven Development - Users, Infrastructure, Tools and Evaluation (SUITE)*, 2012, pp. 27–30.

[9] L. Shi, H. Zhong, T. Xie, and M. Li, "An empirical study on evolution of API documentation." in *Proc. of Conference on Fundamental Approaches to Software Engineering (FASE)*, vol. 6603. Springer, 2011, pp. 416–431.

[10] A. Ko and Y. Riche, "The role of conceptual knowledge in api usability," in *Proc. of IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2011, pp. 173–176.

[11] D. Hoffman and P. Strooper, "API documentation with executable examples," *J. Syst. Softw.*, vol. 66, no. 2, pp. 143–156, May 2003.

[12] J. E. Montandon, H. Borges, D. Felix, and M. T. Valente, "Documenting APIs with examples: Lessons learned with the APIMiner platform," in *Proc. of IEEE Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 401–408.

[13] Z. Zhu, Y. Zou, B. Xie, Y. Jin, Z. Lin, and L. Zhang, "Mining api usage examples from test code," in *Proc. of 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME),*. IEEE, 2014, pp. 301–310.

[14] J. Stylos, B. Myers, and Z. Yang, "Jadeite: Improving API documentation using usage information," in *Proc. of Extended Abstracts on Human Factors in Computing Systems*, ser. (CHI EA). ACM, 2009, pp. 4429–4434.

[15] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns, "What makes a good code example?: A study of programming Q&A in StackOverflow," in *Proc of. IEEE International Conference on Software Maintenance*, 2012, pp. 25–34.

[16] C. Parnin and C. Treude, "Measuring API documentation on the web," in *Proceedings of the International Workshop on Web 2.0 for Software Engineering*, ser. (Web2SE). ACM, 2011, pp. 25–30.

[17] Y.-C. Wu, L. W. Mar, and H. C. Jiau, "Codocent: Support api usage with code example and api documentation," in *Proc. of IEEE International Conference on Software Engineering Advances (ICSEA)*, 2010, pp. 135–140.

[18] C. Chen and K. Zhang, "Who asked what: Integrating crowdsourced faqs into api documentation," in *Companion Proc. of the ACM International Conference on Software Engineering*, ser. ICSE Companion 2014, 2014, pp. 456–459.

[19] B. Dagenais and M. Robillard, "Recovering traceability links between an api and its learning resources," in *Proc. of 2012 International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 47–57.

[20] C. Treude and M. Robillard, "Augmenting api documentation with insights from stack over-flow," in *Proc. of ACM International Conference on Software Engineering*, ser. ICSE '16, New York, NY, USA, 2016, pp. 392–403.

[21] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[22] M. Maleshkova, C. Pedrinaci, and J. Domingue, "Proc. of ieee investigating web APIs on the world wide web," in *Proc. of European Conference on Web Services (ECOWS)*, 2010, pp. 107–114.

[23] S. Sohan, C. Anslow, and F. Maurer, "A case study of Web API evolution," in *Proc. of IEEE World Congress on Services (SERVICES)*.   IEEE, 2015, pp. 245–252.

[24] M. J. Hadley, "Web application description language (WADL)," 2006.

[25] J. Mangler, P. P. Beran, and E. Schikuta, "On the origin of services using RIDDL for descrip-tion, evolution and composition of RESTful services," in *Proc. of IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, 2010, pp. 505–508.

[26] J. Kopecky, K. Gomadam, and T. Vitvar, "hRESTS: An HTML microformat for describing RESTful web services," in *Proc. of International Conference on Web Intelligence and Intel-ligent Agent Technology (WI-IAT)*, vol. 1.   IEEE, 2008, pp. 619–625.

[27] R. Verborgh, T. Steiner, D. Van Deursen, J. De Roo, R. d. Walle, and J. Gabarro Valles, "Capturing the functionality of web services with functional descriptions," *Multimedia Tools and Applications*, vol. 64, no. 2, pp. 365–387, 2013.

[28] P. Danielsen and A. Jeffrey, "Validation and interactivity of web API documentation," in *International Conference on Web Services*.   IEEE, 2013, pp. 523–530.

[29] N. Li, C. Pedrinaci, M. Maleshkova, J. Kopecky, and J. Domingue, "OmniVoke: A framework for automating the invocation of web APIs," in *Proc. of IEEE International Conference on Semantic Computing*, 2011, pp. 39–46.

[30] M. Polák and I. Holubová, "REST API management and evolution using MDA," in *Proc. of ACM International Conference on Computer Science & Software Engineering*, ser. C3S2E '15, New York, NY, USA, 2008, pp. 102–109.

[31] J. Tulach, "Teamwork," in *Practical API design: Confessions of a java framework architect*, C. Andres, Ed. Berkeley, CA: Apress, 2008, pp. 291–302.

[32] S. Sohan, C. Anslow, and F. Maurer, "Spyrest in action: An automated RESTful API documentation tool," in *Proc. of IEEE/ACM International Conference on Automated Software Engineering, ASE, Lincoln, NE, USA*, 2015, pp. 813–818.

Chapter 6

A Study of the Effectiveness of Usage Examples in REST API Documentation

6.1 Abstract

Generating and maintaining REST API documentation with usage examples can be a time consuming and expensive process for evolving APIs. Most REST API documentation tools focus on automating the documentation of the API objects, but require manual effort for capturing usage examples. Consequently, REST API developers need to know the cost vs. benefit of providing usage examples in the documentation to prioritize the documentation efforts. To this end, we have performed a controlled study with 26 experienced software engineers to understand problems that REST API client developers face while using an API without usage examples. We found that REST API client developers face productivity problems with using correct data types, data formats, required HTTP headers and request body when documentation lacks usage examples. By following the REST API documentation suggestions from this paper, REST API developers can reduce the errors, improve success rate and satisfaction of API client developers.

6.2 Introduction

REST APIs are used as the predominant application integration mechanism over the Internet. Generating and maintaining REST API documentation with usage examples can be an expensive process because most API documentation tools do not support effective usage examples. Researchers have emphasized API documentation as a key factor that impacts API usability both positively and negatively. To improve API documentation, researchers have recommended incorporating usage examples in the API documentation.

98

In a resource constrained environment, it is important to understand the value of usage examples on REST API usability to allocate sufficient attention and efforts towards incorporating examples in the API documentation. While it is expected that examples help API client developers, API developers need to know what to include in the examples and why.

The documentation of REST APIs has distinctive features compared to the documentation of local APIs such as Java libraries. For example, local API documentation commonly comprises the description of classes and interfaces with their methods. In contrast, REST API documentation needs to include information about HTTP headers, request and response body and the data representation formats such as JSON, XML. The existing research on API usability areas have primarily focused on local APIs. In our work we have focused on understanding the impact of usage examples within the realm of the distinctive REST API features.

We designed and performed a controlled study with experienced software engineers to understand how REST API client developers are affected while using an API documentation that describes the API elements but lacks usage examples. Participants were divided into two groups and given the same set of API tasks to complete. One group was given the official WordPress REST API documentation and another group was given an enhanced version of the documentation where three usage examples were added. Using a novel technique, we collected 539 API calls made by the participants. We analyzed the data and observed recurring obstacles faced by the participants while performing the API tasks using the official documentation that lacks usage examples. Our contributions on this paper are as follows:

- We provide a list of obstacles that REST API client developers face while performing API tasks using documentation that lacks usage examples.

- We provide a list of recommendations for REST developers to be used as a guideline to incorporate usage examples in API documentation.

- We provide empirical evidence that usage examples in REST API documentation help API client developers perform API tasks with higher developer satisfaction,

99

less time, and better success rate.

The remainder of this paper is organized as follows: In Section 6.3 we present our research question. In Section 6.4 we discuss our methodology in terms of the study requirements, the selected API of the study and the participant selection, the study setup, data collection and analysis methods. In Section 6.5 we discuss the results of our analysis. We discuss the threats to validity and provide a summary of the related work in Sections 6.6 and 6.7 respectively.

6.3 Research Question

This research is aimed at answering the following question:

- **RQ**. What obstacles do API client developers face when using a REST API documentation that lacks usage examples?

6.4 Methodology

To answer the aforementioned research question, the study has the following requirements:

- **R1. Representative API.** We had to choose an existing REST API that is currently used by API client developers. Selecting a mature REST API for this study reduces the possibility that the obstacles we observe in the study are the results of insignificant accidental problems symptomatic of an immature technology. We selected a familiar domain so that participants are able to relate to the API features without requiring upfront training. In addition to selecting the API, we had to select tasks that are related to the core features provided by the API to represent a common usage area of the API.

- **R2. Open source.** To be able to understand the impact of usage examples, we needed to select an open-source API where we can add new examples to the documentation for performing this study.

- **R3. Time bound.** We applied a maximum time constraint for each participant to measure the rate of success within a limited time frame. As a result, the study needs to be setup such that participants are able to focus on performing the tasks minimizing any overhead.

- **R4. Participant Selection.** Developers with prior experience on REST APIs need to be recruited as study participants to perform the study within a limited amount of time and in a realistic setup. Furthermore, to reduce a learning bias, only participants with no prior experince of using the WordPress REST API V2 are accepted for this study.

6.4.1 Study API

We selected the WordPress REST API V2 for this study. WordPress is a blog-like open-source (R2) framework used by over 409 million people to visit 23.6 billion pages each month [1]. The API allows programmatic access to list, create, update, and delete WordPress data such as blog posts, comments, users, images, and tags.

The WordPress REST API V2 has been published and maintained since May 2015. Before January 2017, the WordPress REST API V2 was distributed as a plug-in where WordPress users could optionally install the API component. The following statistics are for the plug-in installation numbers between May 2015 and October 2016:[2]

- Total installations: $\approx 248K$ installs of the plug-in.

- Average daily installations: ≈ 500.

Starting January 2017, the WordPress REST API V2 is no longer required to be installed as a separate plug-in since it is bundled with WordPress installation. According to the code repository, there are a total of 99 and 46 contributors that had at least one commit to the code repository of the

---

[1]https://wordpress.com/activity/
[2]https://wordpress.org/plugins/rest-api/stats/

API and its documentation, respectively. These properties satisfy R1 AND R2, our requirements for using a representative API. By selecting an open-source project we are able to access the source-code to inspect the implementation and documentation technique of the WordPress REST API. The API implements a self-documenting feature where API developers expose API endpoints over HTTP OPTIONS verb to explain the API elements. To implement this feature, the API developers describe the API elements in the code. For example:

Listing 6.1: Example of self-documenting API Code in PHP

```php
1  public function get_item_schema() {
2    $schema = array(
3      '$schema'    => 'http://json-schema.org/draft-04/schema#',
4      'title'      => $this->post_type,
5      'type'       => 'object',
6      /*
7       * Base properties for every Post.
8       */
9      'properties' => array(
10         'date'             => array(
11             'description' => __( "The date the object was published, in
                  the site's timezone." ),
12             'type'        => 'string',
13             'format'      => 'date-time',
14             'context'     => array( 'view', 'edit', 'embed' ),
15         ),...
```

On Listing 6.1, line 4 specifies that this is a schema definition for the API element `Post` [3]. On Line 10, it defines `date`, one of the properties of `Post`, followed by a human readable description and type information. On line 14, the context of this property is mentioned as `view, edit, embed`, meaning that this property will be returned when the `Post` object is returned, embedded, or can be used as an input for editing.
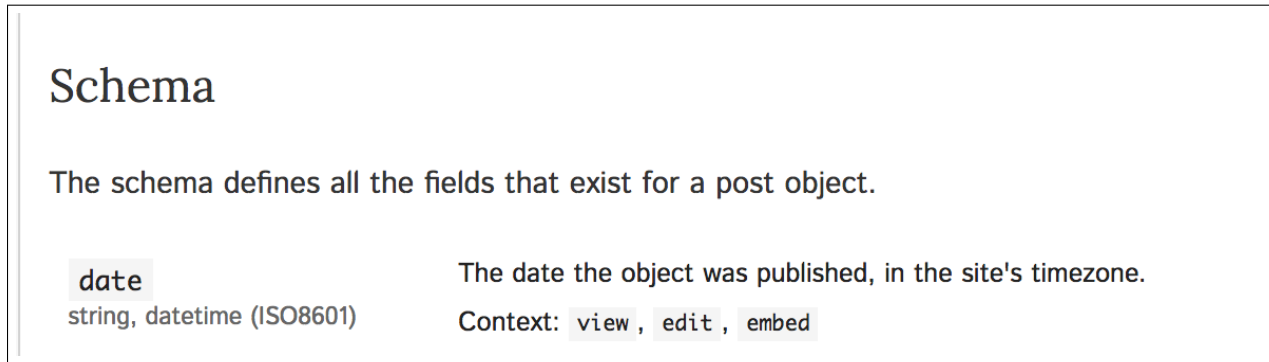
---

[3] https://github.com/WP-API/WP-API

Figure 6.1: Screenshot of Documentation for Post/date

The WordPress team leverages this self-documenting feature to generate and publish the official API documentation as an HTML website. Fig. 6.1 shows a screenshot of the published documentation for the `date` attribute of the `Post` API element.[4]

In addition to the auto-generated documentation of the the schema and API actions, contributors add custom content to provide prosaic overviews and usage examples.

6.4.2 Study Design

Tasks

The participants are requested to perform a total of six tasks using the API including one practice task. All of the tasks are related to a single API element, `Post`[5]. The tasks get progressively more difficult, and all but the last task can be performed independently of each other. Participants are requested to limit the total time on the study to a maximum of one hour (R3). Participants are encouraged to proceed to the next task when they are either satisfied with their answer or feel stuck and unable to make progress.

To use a REST API, the API client developers need to work with the following four inputs over HTTP:

- I1. Request method

---

[4] http://v2.wp-api.org/reference/posts/
[5] A Post identifies a blog post within WordPress

- I2. Request URL

- I3. Request headers

- I4. Request body

To verify the response of an API call, the API client developers can use HTTP response headers and/or HTTP response body. To perform the tasks, the participants are required to use one or more of the inputs I1-4. In the following paragraphs, we describe each task with its description and the study observation goals against the aforementioned API input and output information. For each of the tasks, the participants are required to use the same WordPress REST API and one of the two variants of accompanying documentation.

T1: List all posts task. We asked the participants to use the WordPress REST API to get a list of the blog posts from a given WordPress site. This is the practice task, and the inputs to answer this task are pre-filled for the participants. It allows the participants to understand the tools used for this study as well as get familiarity with the Post API. The answer for this task makes use of I1 and I2.

T2: Filter posts by author task. The participants are asked to use the API to filter the list of posts obtained in T1 by an author given the author's user name. To answer this correctly, the participants are required to first make an API call to get the numeric ID of the author given the string based user name. Then, the ID needs to be used on the `Post` API to filter posts by the author. This task allows us to understand the impact of usage examples on API client developers when multiple API calls need to be made to perform a task using the API. Inputs I1-2 are required to complete this task successfully.

T3: Exclude posts by IDs task. We ask the participants to use the API to get a list of all posts excluding posts with IDs 1 and 4. Participants need to use the inputs I1-2, and use a desired format on I2 to pass an array of IDs as a parameter. This task allows us to understand how API client developers identify the format for using an array within the URL with respect to the usage examples in the API documentation.

104

T4: Find total posts task. This task requires the participants to use the API to find a total number of posts. Participants need to use the inputs I1-2 and inspect the HTTP response headers to successfully complete this task. This task allows us to understand how API client developer productivity is affected with respect to missing examples about HTTP response headers in the API documentation.

T5: Publish post task. We ask the participants to use the API to publish a blog post with a specific title, content, and a published date. To successfully complete this task, the participants are required to use all four input types and inspect both the HTTP response header and the response body. Additionally, the participants are required to use a specific date format that the API accepts as a valid format for date specification. Answers to this task allows us to study API client developer productivity with respect to the usage examples lacking details about the inputs I3-4.

T6: Update post task. We ask the participants to use the API to update a blog `Post` that they published in T5 with a new `excerpt`. Similar to T5, this task requires the use of inputs I1-4, but with different values for the inputs. This task allows us to understand API client developer productivity on inter-dependent tasks with respect to usage examples.

To summarize, the tasks allow us to understand how REST API client developers approach API tasks of different complexity levels involving various input types and available output information with respect to the usage examples in the API documentation.

Participant Selection

To satisfy the requirement of developers with REST API experience (R4), we used the following criteria for recruiting the study participants:

- Currently working as a software engineer.

- At least 1 year of industry experience as a developer.

- At least 1 year of industry experience with REST APIs.

- No prior experience with WordPress REST API.

105

Figure 6.2: Screenshot of the Custom-built Web-based REST API Explorer Used by Study Participants

We recruited a total of 26 participants from sixteen different companies and six different countries (Canada, USA, Germany, Ireland, Brazil, and, Bangladesh) through online announcements posted on Twitter, Facebook, and software developer focused mailing lists. Table 6.1 shows a summary of the experience level of the participants in each group.

Pilot Studies

We conducted two pilot studies to develop a process for performing this study. The first pilot study involved four participants that joined the first author in-person or over video conferencing. The study involved tasks using the WordPress REST API V2 and the GMail REST API. Each participant was given one of the two APIs and and a set of API tasks and online answer forms to complete using the API within an hour. The primary findings from this pilot are as follows: 1)

Create a Post

Arguments

date	The date the object was published, in the site's timezone.
date_gmt	The date the object was published, as GMT.
slug	An alphanumeric identifier for the object unique to its type.
status	A named status for the object. One of: publish , future , draft , pending , private
password	A password to protect access to the content and excerpt.
title	The title for the object.

Definition

POST /wp/v2/posts

(a) Original API Documentation

Definition

POST /wp/v2/posts

Example Request

```
$ curl -X POST http://demo.wp-api.org/wp-json/wp/v2/posts \
 --header "Authorization: Basic c3B5cmVzdDowT1BqIHRCQTcgUUtGUiBNaEFU" \
 --header "Content-Type: application/json" \
 --data '{"title":"My New Title",
"content": "My New Content",
"excerpt": "Sample excerpt",
"date": "2010-01-01T02:00:00-10:00",
"status":"publish",
"comment_status": "open",
"ping_status": "open",
"sticky": false}'
```

Example Response Headers

(b) API Documentation Enhanced with an Example

Figure 6.3: Screenshots of Original vs. Enhanced API Documentation

Table 6.1: Participant Profile

	Group 1	Group 2
Number of Participants	16 (P1.1-P1.16)	10 (P2.1-P2.10)
Industry Experience		
1-5 years	5	1
5-10 years	6	5
10+ years	5	4
Average	9.1	10.6
REST API Experience		
1-3 years	5	3
3-5 years	7	4
5+ years	4	3
Average	4.5	4
Number of Companies	10	8

participants required significant overhead time to setup a development environment with the proper API credentials, 2) the intermediate trial attempts of using the API are potentially more valuable than the final answer as it allows us to study what obstacles the participants face, 3) the number of tasks for the study had to be reduced to fit within the one hour limit, and 4) for the GMail API, participants used up a large portion of their time on learning about how to use OAuth.

To overcome the shortcomings found from the first pilot study, we decided to develop a web-based REST API explorer as shown on Fig. 6.2 that allows participants to use their browser to make the API calls without setting up any development environment. The web-based API explorer records the inputs I1-4, and displays the API response on the click of a button. Thus the participants could focus on using the right input and verifying the output without writing code. The web-base REST API explorer also allowed us to automatically collect all the trial API calls that the participants make for each API task. A second pilot study involving seven new participants was

performed to collect usability related feedback about the web-based REST API explorer. We only used WordPress since it did not require knowledge about OAuth. From this study, we observed patterns of mistakes that API client developers make that can be reduced by adding usage examples in the API documentation.

Protocol

Learning from the pilot studies, we designed the main study protocol as follows: we enhanced the original WordPress REST API documentation and added a total of 3 API usage examples to show listing of blog posts with query parameters for filtering, a request to create a blog post and a request to update a blog post. We used the WordPress API unit tests to find relevant data for these examples. Fig. 6.3a shows a screenshot of the original API documentation related to T4 where the API client developers are provided with a reference table describing the different properties that can be used to create a `Post` object[6]. Fig. 6.3b shows a screenshot of the enhanced API documentation with a cURL[7] based usage example. cURL is used because it is used elsewhere in the original API documentation. In the enhanced API documentation, the example shows one possible API call with realistic values for the data that is described in the reference table and associated API response headers and body.

We divided the participants into two groups, G1 and G2. The participants in the pilot are not counted towards G1 and G2 for the study. G1 participants were provided with a link to the official API documentation on the web-based API explorer, and G2 participants were provided with a link to the enhanced API documentation with usage examples. We designed the the web-based API explorer to allocate more participants to G1 compared to G2 because we wanted to better understand the impact of the lack of usage examples on API client developer productivity. However, each individual participant was randomly assigned to a group by the web-based API explorer. All participants were given the same set of API tasks and were requested to limit their participation time to a maximum of one hour. No task specific time limit was imposed except an

---

[6]http://v2.wp-api.org/reference/posts/
[7]https://curl.haxx.se/

overall limit of one hour for the entire study because we wanted participants to spend sufficient time on each task without forcing them to move to the next one. Participants were allowed to access the internet and external resources alongside the provided API documentation to perform the tasks as they would normally use on a typical work day. We used the web-based API explorer to also collect an experience rating on a scale of 0-10, 10 being the best possible, of using the given REST API documentation and a free-form feedback from each participant as an exit survey.

Data Collection

The data collected by the web-based REST API explorer for each participant is exported into a text file as the raw data artifact. For each API task and each participant that attempted the task, the exported text file contains the request inputs I1-4 and associated API response headers and body for each API call made by the participant. For each participant, we recorded all trial API calls with timestamps. Additionally, the demographic information, experience rating and the free-form feedback for each participant is also included in the artifact.

Data Analysis

We analyzed the data artifact qualitatively to understand the obstacles that REST API client developers face when using an API documentation that lacks usage examples. We exported a table of data from the web-based REST API explorer. For each row, the table contains the following columns: API task, participant identifier, timestamp, API trial number, I1-4, response headers and body. The values for I1-4 used by the participants were manually coded to group the participant responses into categories. The categories help us determine the information type that API client developers need in the API documentation to perform API tasks. The analysis started with an empty set of codes and new codes were introduced to describe scenarios that did not fall under existing codes. The first author applied the codes on the raw data artifact and provided a coding scheme comprising 11 codes to a co-author. The co-author applied the codes to 82/539 API calls (10% confidence interval with 95% confidence level). The two authors validated the codes against each other to evaluate consistency. The initial coding from the co-author had discrepancies in 4 /

110

11 codes for 27 / 82 API call samples. After discussing the coding scheme, the discrepancies were resolved as the definition of the terms became clearer.

For each API task attempted by each participant, we annotated the artifact with one of the following labels: successful, partially successful, and unsuccessful. Task participations are marked successful when I1-4 matches the required values for performing the given task. If a participant is able to use the correct I1-4 for one of the two tasks required to complete a single task (T2), we marked it as partially successful. Otherwise, it's marked as unsuccessful. Based on these annotations, the following formulas were used to compute the quantitative results:

Success Rate (Task, Group) = (No. of participants that successfully completed Task) / (No. of participants in Group)

Average Trial API Calls (Task, Group) = (No. of API calls made by Group on Task) / (No. of participants in Group).

Average Time Spent (Task, Group) = (Time spent by Group on Task) / (No. of participants in Group)

6.5   Results

6.5.1   Quantitative Analysis

We present a summary of the quantitative results as found by analyzing a total of 539 API calls (385 from G1, 152 from G2) made by the participants from this study in Fig. 6.4. Fig. 6.4a juxtaposes the average number of trial API calls made by each group against their rate of success in performing the given tasks T2-T6. The average number of trial API calls and success rate for each group is computed using the aforementioned formulas.

Even though it is expected that G2 would outperform G1 with an enhanced documentation, we were surprised to see the extent of the difference in effort vs. success of the two groups. In Fig. 6.4a and 6.4b we see that, for the simpler tasks, T2 and T3, G1 required more time and more trial API calls and reached successful answers with a lower percentage compared to G2. For T4,
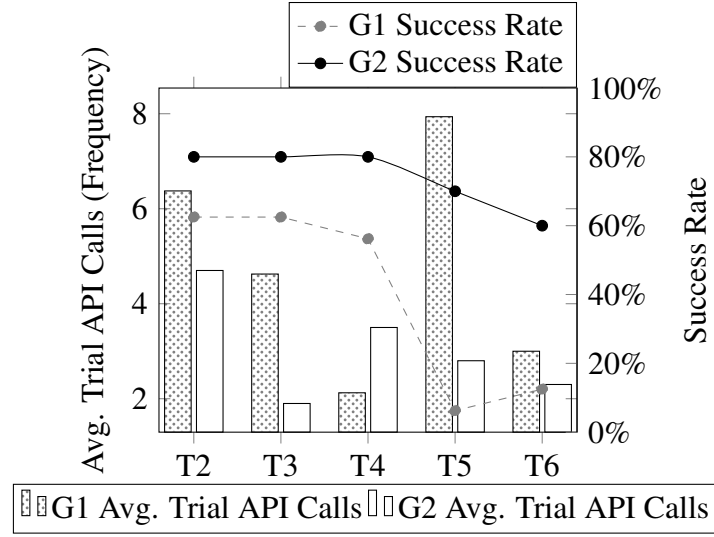
(a) Trial vs. Successful API Calls
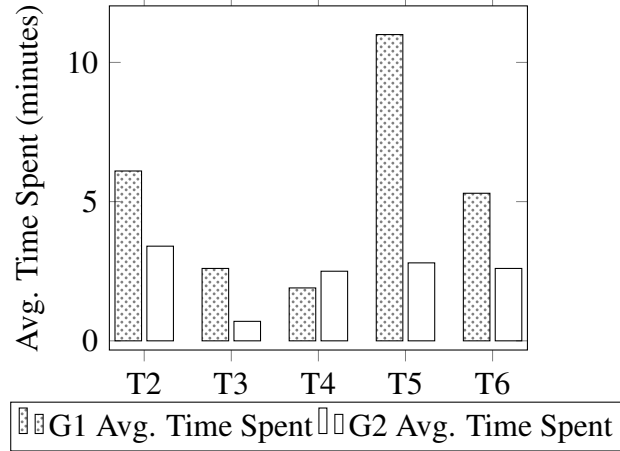
(b) Average Time Spent

Figure 6.4: Quantitative Comparison between G1 and G2

where the participants are required to inspect an HTTP response header, G2 executed more API calls on average due to P2.7 who made 11 API calls even though reached the successful call on the 3rd attempt. For the more complex tasks T5 and T6, we see that participants in G1 had very low success rate and very high number of average API trials compared to G2. We observed a p-value of 0.0003 for the success rates using Fisher test, and p-values of 0.017 and 0.051 for the trial API calls and time spent using Wilcoxon rank sum test. The data here confirms that without usage examples developers spend more time and execute more trial attempts yet have lower success rates.
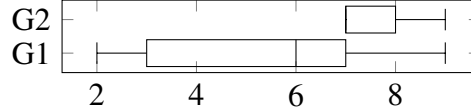
Figure 6.5: Participation Satisfaction Ratings on a Scale of 0-10

In Fig. 6.5, we see that the G2 participants provided higher satisfaction rating compared to G1 participants (with a p-value of 0.003 using Wilcoxon rank sum test). P1.1 provided a satisfaction rating of 9, and analyzing the participant feedback against the success, we see a false positive case, where the participant perceived that the tasks were successfully completed even though only 2 of the 5 tasks were correctly answered. Other than this case, the satisfaction ratings provided by G1 participants consistently fall behind the G2 participants' ratings.

These results support using a more qualitative approach due to our limited number of participants. The results provide an initial quantification of the impact of examples for typical REST-API usage tasks. This quantification is preliminary evidence that can help justify future investment in API documentation.

6.5.2   Qualitative Analysis Results

In their free-form feedback, 11 of the 16 G1 participants mentioned the lack of examples with HTTP headers, request and response as the primary problem with the API documentation. On the other hand, 7 out of the 10 G2 participants mentioned the examples being helpful from the enhanced documentation. We identify usage examples as the most frequent and central topic of interest from our participants as per their feedback. This is also supported by our quantitative analysis results. In the remainder of this section, we provide our observations related to each task and their implications.

Observation 1.   Use of Data Type.   Without usage examples, we observed 10 of the 16 G1 participants used incorrect data type of `String` instead of `Integer`. To successfully complete T1, participants had to use an integer ID representing an `Author` object given the author name. In the original API documentation, the following is mentioned about the author parameter: "author:

113

Limit result set to posts assigned to specific authors." The documentation does not mention that the data type required is the numeric author ID, not the author name. P1.12 mentioned the following: *"...to find posts for the author, I had to inspect the response to see that indeed the author_id was in there"*. The enhanced documentation showed one example of using numeric ID. 3 of the 10 participants from G2 ran into the same error. When faced with this error, participants used trial and error to eventually complete the task.

> **Recommendation 1.** REST API documentation needs to include examples of data types such as Integer, String, and Array for each API field to satisfy API client developer information need.

Observation 2. Use of Data Format. We observed participants in G1 faced obstacles using correct data formats for `Date` and `Array` type data in API requests without usage examples. Tasks T3 required the use of a correct format to represent an array of numeric post IDs to be excluded using the API. The original documentation mentions the following about this API query parameter: "exclude: Ensure result set excludes specific IDs." Participants attempted to solve this task using multiple different formats for specifying the array. For example: they tried with `exclude=[1,4]`, `exclude=1&4`, `exclude=4&&exclude=1`, `id!=1&id!=4` and other alternatives before eventually finding the right format as follows `exclude=1,4`. Participant P1.6 mentioned the following: *"It was difficult to figure out whether some of the inputs needed to be arrays or just a comma-separated list."*. Seven of the 16 participants in G1 faced this error. The enhanced documentation showed one example of using multiple IDs and all G2 participants could use the right format.

Task T5 also required the participants to use an the ISO8601 formatted date for publishing a post with a specific date. Even though the original documentation mentions ISO8601 format, it does not provide an example. Instead of using `2016-01-01T00:00:00-00:00`, participants from G1 used date formats as follows: `January 1, 2016`, `2016-01-01T12:00:00`, `20160101`. P1.8 mentioned the following: *"Got hung up trying to figure out the formatting of the date (which*

doesn't appear to be ISO8601, despite what it says)". After analyzing the response, we found that P1.8 used the ISO8601 formatted date in the API calls, but did not provide the required `second` portion of the time. Eight of the 16 participants in G1 had an error in the date formats. With an example, 1 of the 10 participants had the same error from G2.

Recommendation 2. REST API developers need to include examples showing the valid data format for the API elements.

Observation 3. Use of Request Headers. We were surprised to see that 14 of the 15 G1 participants that tried T5 and T6 faced problem using correct request headers without usage examples. Tasks T5 and T6 required the use of a HTTP request header named `Content-Type`. In the original API documentation, the following was mentioned: "The API uses JSON exclusively as the request and response format, including error responses.". Experienced software engineers still had problem using this HTTP header as we found from the participants in G1. P1.12 mentioned the following: *"I didn't know how to specify application/json so that I could send POST data via the body instead in the URL parameters... Examples would really help - with a sample request and a sample response."* Even though G1 participants have years of experience with REST APIs, only one of them could use the `Content-Type` header correctly without example. In contrast, 2 out of the 10 participants from G2 failed to use this header correctly as indicated in the example in the enhanced documentation. Moreover, 1 of the 2 G2 participants that did not use the header the first time, later used it correctly on a subsequent attempt.

Recommendation 3. If API requests need to use HTTP Request headers, in addition to the request method and body, REST API develpers need to include examples of the HTTP headers.

Observation 4. Use of Interdependent API Calls. We observed the G1 participants had problem completing API tasks that require multiple API calls without usage examples. Tasks T2

and T6 required the use of multiple API calls to complete successfully. Both tasks require an initial API call to find data using the API that is needed make a second API call. For example, P1.5 mentioned the following about T6, *"The post I created is not visible and not editable, though I got 201. Not sure why"*. After analyzing the response, we observed that P1.5 was able to create a blog post, but failed to publish it because it did not have the `publish` status. As a result, the post was not returned via the API for T6. P2.2 mentioned the following about T2: *"In case of search posts by author, it's not clear how to find the author_id. Although, I realized that users and authors are same in this case."*

Recommendation 4. If there are prerequisites for making an API call, REST API developers need to provide examples showing how to get the prerequisites in the API documentation.

To summarize, we recommend API developers to consider usage examples as an essential requirement for REST API documentation. As such, sufficient resources and a high priority should be devoted to generate and maintain the usage examples. Based on our observations, we recommend REST API documentation tools to provide first-class support for including usage examples with realistic data.

6.6 Threats to Validity

The selection of the API, the tasks, and the participants may introduce selection bias. It is possible that there are obstacles we do not know about that could be more severe, but were not observed because of our task selection. The analysis of the raw data may introduce a bias. To reduce this, we have involved multiple researchers and a repeatable coding scheme. A more complete API documentation without adding usage examples may provide insights from our findings.

116

6.7   Related Work

In this section, we discuss the existing research on the relationship between API usability and usage examples. To this regard, we summarize the related work on APIs from the sub-areas of API learning obstacles, using crowd-sourced API examples, measuring API usability, and controlled studies on the impact of usage examples on API usability.

API Learning Obstacles

Robillard et al. used a mixed approach of surveys and in-person interviews with professional software engineers to understand the API learning obstacles [1] [2]. They found that many obstacles for API learnabilty were related to API documentation. They identified five impactful factors for API documentation as follows: documentation of intent, code examples, matching APIs with scenarios, penetrability of the API, and format and presentation. Uddin and Robillard surveyed 323 professional software engineers to understand the different ways in which a piece of documentation can be unfit for purpose [3]. They identified ambiguity, incompleteness, and incorrectness as the three severest problems that lead API documentations to fail to answer the information needs of developers. Duala-Ekoko and Robillard performed a study with 20 programmers to understand the types of questions that developers have while using an unfamiliar API [4]. They categorized the API related information needs into twenty generic questions that can be used to analyze the quality of API documentation. Myers et al. found that participants had limited success finding the relevant API elements to perform an API task using an enterprise API [5]. Among other factors, they recommended providing code examples in the API documentation to help API client developers. Our work fits in the general space of studies of API documentation obstacles. We focus on problems related to the absence of examples within the area of REST APIs.

Crowd-Sourced API Usage Examples

Wang et al. performed an exploratory study to understand the current state of API related knowledge available on the Internet [6]. They searched the web for usage examples related to the API

of five popular Java libraries and found that on average API examples could be found for 77% of the 4,637 APIs from the libraries. Moreover, they found that the crowd-sourcing sites accounted for 93.7% of the usage examples compared to 6.13% that were officially published. An evaluation of Jadite showed that developers were three times faster to complete API tasks with access to auto-referred crowd-sourced usage examples [7]. Nasehi et al. analyzed StackOverflow threads to understand the characteristics of good and bad code examples [8]. They found that commonly down-voted API related answers lacked code examples, explanations, and shortcoming of solutions. Treude et al. proposed a machine learning based approach to find relevant API documentation from StackOverflow using both the text and metadata found from the questions and answers [9]. Kim et al. presented a technique to automatically augment code examples from the web using code search tools to Java APIs [10]. Jiau et al. observed a severe inequality within the context of crowd-sourced API usage examples where most of the content were related to the popular APIs [11]. Existing research on crowd-sourced API documentation mostly focused on finding usage examples of local APIs. In this paper, we focused on REST APIs, where the API client development programming language is agnostic to API implementation language. This distinction with local APIs makes it hard to use the techniques proposed in the aforementioned papers for finding crowd-sourced REST API usage examples.

Measuring API Usability

Rama et al. presented a set of formulas for computing a measure of API usability based on the API's structural components such as classes, methods, parameters, return values, thread-safety [12]. Scheller et al. presented a framework for automatically measuring the complexity of an API [13]. They identified a list of measurable API properties and provided formulas to compute complexities related to the interfaces, implementation and setup of an API. Grill et al. suggested using a combination of expert opinion and developer workshops to identify and collect feedback about API and its documentation related problems [14]. Farooq et al. recommended using peer reviews of API code in addition to API usability studies to uncover API related bugs and incorporate feed-

back [15]. In our work, instead of measuring API usability, we have focused specifically on the relationship of REST API usability with respect to usage examples.

Controlled Studies

Nasehi et al. performed a controlled study to understand if API unit tests can be used to provide as usage examples to facilitate API client developers [16]. The researchers found that the examples from the unit tests helped understanding the API concepts better but it was challenging for the participants to locate relevant examples. Endrikat et al. performed a controlled study with four groups of participants to understand the impact of API documentation on APIs that are implemented using programming languages with static and dynamic type systems [17]. Participants were given a set of failing unit tests and were asked to make them pass by writing code using the studied API. They found that the participant group using a statically typed API with explicit documentation could get the tests to pass in less time than the other groups. Our work in this paper is based on a controlled study and shares part of the setup that were used by the aforementioned controlled studies with the following differences: our goal is to understand the obstacles faced by API client developers without usage examples; we focus on REST APIs instead of local APIs; and, the participants are all professional software engineers.

6.8 Conclusion

In this paper, we presented a set of problems that experienced REST API client developers face while performing API tasks using API documentation that lacks usage examples. We identified that, without examples, REST API client developers have trouble using the correct data types, correct data formats, and required HTTP headers and request body. We've also presented empirical evidence that by adding usage examples makes it possible to reduce mistakes, improve success rate and developer satisfaction of using the API. REST API developers can follow our recommendation as a set of guidelines while documenting REST APIs to improve API usability. REST API documentation tool developers can leverage our findings to improve reusable tool support for software

119

developers.

Acknowledgment

6.9   References

[1] M. Robillard, "What makes APIs hard to learn? the answers of developers," *Software, IEEE*, vol. PP, no. 99, pp. 1–1, 2011.

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

[3] G. Uddin and M. P. Robillard, "How API documentation fails," *IEEE Software*, vol. 32, no. 4, pp. 68–75, July 2015.

[4] E. Duala-Ekoko and M. P. Robillard, "Asking and answering questions about unfamiliar APIs: An exploratory study," in *Proc. of the International Conference on Software Engineering*.   IEEE Press, 2012, pp. 266–276.

[5] B. Myers, S. 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," *J. Organ. End User Comput.*, vol. 22, no. 1, pp. 23–51, 2010.

[6] L. Wang, Y. Zou, L. Fang, B. Xie, and F. Yang, "An exploratory study of API usage examples on the web," in *2012 19th Asia-Pacific Software Engineering Conference*, vol. 1, Dec 2012, pp. 396–405.

[7] J. Stylos, A. Faulring, Z. Yang, and B. A. Myers, "Improving api documentation using API usage information," in *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Sept 2009, pp. 119–126.

[8] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns, "What makes a good code example?: A study of programming Q&A in StackOverflow," in *Proc of. IEEE International Conference on Software Maintenance*, 2012, pp. 25–34.

[9] C. Treude and M. Robillard, "Augmenting api documentation with insights from stack overflow," in *Proc. of ACM International Conference on Software Engineering*, ser. ICSE '16, New York, NY, USA, 2016, pp. 392–403.

[10] J. Kim, S. Lee, S.-w. Hwang, and S. Kim, "Adding examples into java documents," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09.   Washington, DC, USA: IEEE Computer Society, 2009, pp. 540–544.

[11] H. Jiau and F.-P. Yang, "Facing up to the inequality of crowdsourced API documentation," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 1, pp. 1–9, 2012.

[12] G. M. Rama and A. Kak, "Some structural measures of API usability," *Software: Practice and Experience*, vol. 45, no. 1, pp. 75–110, 2015.

[13] T. Scheller and E. Kühn, "Automated measurement of API usability: The API concepts framework," *Information and Software Technology*, vol. 61, pp. 145–162, 2015.

[14] T. Grill, O. Polacek, and M. Tscheligi, *Methods towards API Usability: A Structural Analysis of Usability Problem Categories*.   Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 164–180.

[15] U. Farooq, L. Welicki, and D. Zirkler, "API usability peer reviews: A method for evaluating the usability of application programming interfaces," in *Proc. of the SIGCHI Conference on Human Factors in Computing Systems*.   ACM, 2010, pp. 2327–2336.

[16] S. M. Nasehi and F. Maurer, "Unit tests as API usage examples," in *Proc. of International Conference on Software Maintenance*. IEEE, Sept 2010, pp. 1–10.

[17] S. Endrikat, S. Hanenberg, R. Robbes, and A. Stefik, "How do API documentation and static typing affect api usability?" in *Proc. of the International Conference on Software Engineering*. ACM, 2014, pp. 632–642.

Chapter 7

Conclusion

In this thesis, I presented a novel idea of using interception to automate the process of REST API documentation with usage examples from executable code. I started by investigating the current industry practices and challenges associated with developing and maintaining evolving Web APIs. I found modern Web APIs predominently embraced REST architectural style, but the existing literature had primarily focused on solving problems related to the SOAP APIs. I also found that most REST APIs are documented using manual approaches that are time consuming and error-prone to maintain when an API evolves several times a month. Through my research, I aimed to fill this gap between the REST API documentation challenges and the existing literature by finding a reusable solution to the common problems.

In my second study, I investigated the existing literature on API usablity to derive a set of requirements that are applicable to REST API documentation and proposed an idea of automatically generating and maintaining REST API documentation from executable code. I found that existing techniques for REST API documentation systems relied on custom API description languagees that are manually maintained as a parallel artifact to the API code, causing potential drift between the actual API and itâĂŹs documentation as it evolves over time. By using interception, I have demonstrated that it is possible to automatically record and synthesize the recorded API traffic to generate pragmatic REST API documentation from its test code. Additionally, the technique allows API developers to collaboratively document the API with realistic and executable examples for multiple versions as the API evolves.

In my third study, I evaluated the proposed technique in an industrial setup and found it to be a pragmatic solution for production use. The concept of leveraging API tests to maintain an always-updated API documentation has helped the organization establish a quick feedback loop

during the API development process. Practitioners can incorporate the findings from this study to improve their own REST API documentation practice.

To understand the impact of usage examples that can be derived from API tests, I performed the last study with professional software engineers and found that the examples help reduce the common obstacles faced by REST API client developers. I derived a list of recommendations for documenting usage examples to satisfy the common information needs when REST API client developers use an API. Practitiones can follow the recommendations as guideline to allocate appropriate priority while documenting their REST APIs in a resource constrained environment.

7.1   Future Work

Based on my findings from the this research, I identified the following list of potential future research problems.

- **Extension to Support Complex Usage Scenarios.** Future research needs to be carried out to find techniques to automate the process of documenting usage scenarios that require multiple API calls to perform complex API tasks. Future research is also needed to investigate techniques for automatically cross-referencing API elements to allow API client developers with an improved navigation experience while reading the REST API documentation.

- **Automated Changelog Generation.** For frequently evolving APIs, future research needs to extend the proposed technique in this thesis to automatically compute human readable changelogs and detect version compatibility to follow a semantic versioning scheme. This would help API client developers with an accurate information about the potential impact of an API update on their own application.

- **Automated API Client Code Generation.** Several tools exist that can convert custom REST API description files to API client code in multiple programming

languages. Future research can be carried out to map the SpyREST data model with the custom API description languages to leverage the interception technique to reduce the amount of manual effort required to maintain the custom formatted API descriptions as the API evolves.

- **Extensions of the Interception Based Documentation to non-REST APIs.** The proposed technique can be extended to document usage examples for other stateless APIs outside the realm of REST APIs. The structure of REST APIs are commonly documented using programming language features and formatted comments. Future research can investigate the feasibility of leveraging the interception technique to source realistic usage examples for local stateless APIs from existing API test code.

Appendix A

Source Code Fragments of the Spy from Chapter 4

```javascript
1  (function() {
2    var Apithough;
3
4    ApiThough = require('./lib/api_through');
5
6    process.env.NODE_TLS_REJECT_UNAUTHORIZED = "0";
7
8    process.on('uncaughtException', function(err) {
9      console.error('uncaughtException');
10     console.error(err);
11     return console.error(err.stack);
12   });
13
14   new ApiThough().start();
15
16 }).call(this);
```

Listing A.1: index.js

```javascript
1  (function() {
2    var ApiThrough, PassThrough;
3
4    module.exports = function() {
5      return new ApiThrough();
6    };
7
8    PassThrough = require('stream').PassThrough;
9
```

```
10    ApiThrough = (function() {
11      var Proxy, SSLCert, fs;
12
13      Proxy = require('http-mitm-proxy');
14
15      SSLCert = require('./ssl_cert');
16
17      fs = require('fs');
18
19      function ApiThrough() {
20        var host, mongooose;
21        mongooose = require('mongoose');
22        host = process.env["MONGODB_PORT_27017_TCP_ADDR"] || 'localhost';
23        console.log("Connecting to mongo at " + host);
24        mongooose.connect("mongodb://" + host + ":27017/apis");
25      }
26
27      ApiThrough.prototype.start = function() {
28        var proxy;
29        proxy = new Proxy();
30        proxy.use(this);
31        proxy.onCertificateMissing = (function(_this) {
32          return function(ctx, files, callback) {
33            return _this.onCertificateMissing(ctx, files, callback);
34          };
35        })(this);
36        proxy.listen({
37          port: process.env['PROXY_PORT'] || 9081,
38          sslCertCacheDir: './scripts/certs/http-mitm-proxy'
39        });
40        return this.proxy = proxy;
41      };
```

```
42

43     ApiThrough.prototype.onCertificateMissing = function(ctx, files, callback)
            {
44       var sslCert;
45       console.log('Looking for "%s" certificates', ctx.hostname);
46       console.log('"%s" missing', ctx.files.keyFile);
47       console.log('"%s" missing', ctx.files.certFile);
48       sslCert = new SSLCert(ctx.hostname);
49       return sslCert.create((function(_this) {
50         return function() {
51           return callback(null, {
52             keyFileData: fs.readFileSync(ctx.files.keyFile),
53             certFileData: fs.readFileSync(ctx.files.certFile)
54           });
55         };
56       })(this));
57     };

58

59     ApiThrough.prototype.onError = function(ctx, err) {
60       console.error('proxy error:', err);
61       return console.error('proxy error stack:', err.stack);
62     };

63

64     ApiThrough.prototype.onRequest = function(ctx, callback) {
65       var ApiExample, User, apiExample, responseAggregator, responseBody;
66       ApiExample = require('./api_example');
67       User = require('./user');
68       ctx.onError((function(_this) {
69         return function(ctx, err) {
70           return _this.onError(ctx, err);
71         };
72       })(this));
```

```javascript
73        apiExample = new ApiExample();
74        apiExample.populateFromRequest(ctx.clientToProxyRequest);
75        apiExample.setFullUrl(ctx.isSSL, ctx.proxyToServerRequestOptions);
76        responseBody = '';
77        responseAggregator = new PassThrough();
78        responseAggregator.on('finish', function() {
79          apiExample.responseBody = responseBody;
80          apiExample.stripResponseBody();
81          return User.findOne({
82            api_token: apiExample.apiToken
83          }, function(err, user) {
84            var apiExampleRaw;
85            console.log('findOne', err, user);
86            if (user) {
87              apiExample.userId = user.id;
88            }
89            apiExampleRaw = apiExample.toObject();
90            delete apiExampleRaw._id;
91            return ApiExample.findOneAndUpdate({
92              digest: apiExample.digest
93            }, apiExampleRaw, {
94              upsert: true
95            }, function(error) {
96              if (error != null) {
97                return console.log("Failed to save due to error", error);
98              }
99            });
100         });
101       });
102       ctx.addResponseFilter(responseAggregator);
103       ctx.onRequestData(function(ctx, chunk, callback) {
104         apiExample.requestBody += chunk.toString('utf8');
```

129

```
105        return callback(null, chunk);
106      });
107    ctx.onResponse(function(ctx, callback) {
108      apiExample.responseHeaders = ctx.serverToProxyResponse.headers;
109      apiExample.statusCode = ctx.serverToProxyResponse.statusCode;
110      ctx.serverToProxyResponse.on("finish", function() {
111        return console.log("FINISH");
112      });
113      return callback();
114    });
115    ctx.onResponseData(function(ctx, chunk, callback) {
116      responseBody += chunk.toString('utf8');
117      return callback(null, chunk);
118    });
119    return callback();
120  };

121

122  return ApiThrough;

123

124  })();

125

126 }).call(this);
```

Listing A.2: lib/api_through.js

```
 1
 2 (function() {
 3   var ApiExample, ApiExamplesSchema, CUSTOM_HEADERS, VERSION_IN_HEADER,
       VERSION_IN_URL, crypto, inflector, mongoose, url, validator, _u;
 4
 5   mongoose = require('mongoose');
 6
 7   url = require('url');
```

```
8

9    _u = require('underscore');

10

11   crypto = require('crypto');

12

13   validator = require('validator');

14

15   inflector = require('i')();

16

17   VERSION_IN_HEADER = /v(\d|\.\d)+/;

18

19   VERSION_IN_URL = /\/(v\d[^\/]*)/;

20

21   CUSTOM_HEADERS = {

22     DESC_HEADER: "x-spy-rest-desc",

23     VERSION_HEADER: "x-spy-rest-version",

24     RESOURCE_HEADER: "x-spy-rest-resource",

25     API_TOKEN_HEADER: "x-spy-rest-api-token",

26     HOST_HEADER: 'x-spy-rest-host',

27     NO_STRIP_HEADER: 'x-spy-rest-no-strip',

28     ACTION_HEADER: 'x-spy-rest-action'

29   };

30

31   ApiExamplesSchema = new mongoose.Schema({

32     description: {

33       type: String

34     },

35     version: {

36       type: String,

37       "default": 'Default'

38     },

39     resource: {
```

```
40      type: String
41    },
42    action: {
43      type: String
44    },
45    url: {
46      type: String
47    },
48    templatedURL: {
49      type: String
50    },
51    query: {
52      type: Object
53    },
54    host: {
55      type: String
56    },
57    http_method: {
58      type: String
59    },
60    requestBody: {
61      type: String,
62      "default": ''
63    },
64    requestHeaders: {
65      type: Object
66    },
67    statusCode: {
68      type: Number
69    },
70    responseHeaders: {
71      type: Object
```

```
72        },
73        responseBody: {
74          type: String,
75          "default": ''
76        },
77        strippedResponseBody: {
78          type: String
79        },
80        recordedAt: {
81          type: Date,
82          "default": function() {
83            return new Date();
84          }
85        },
86        fullURL: {
87          type: String,
88          "default": ''
89        },
90        digest: {
91          type: String,
92          "default": ''
93        },
94        requiresAuth: {
95          type: Boolean,
96          "default": false
97        },
98        apiToken: {
99          type: String,
100          "default": ''
101        },
102        userId: {
103          type: mongoose.Schema.Types.ObjectId,
```

```
104        "default": null
105      },
106      authType: {
107        type: String
108      }
109    }, {
110      collection: 'api_examples'
111    });
112
113    ApiExamplesSchema.index({
114      host: 1,
115      version: 1,
116      resource: 1,
117      action: 1
118    });
119
120    ApiExample = mongoose.model('ApiExample', ApiExamplesSchema);
121
122    ApiExample.prototype.populateFromRequest = function(request) {
123      this.host = request.headers[CUSTOM_HEADERS.HOST_HEADER] || request.headers
              .host;
124      this.url = this.filteredUrl(request.url);
125      this.http_method = request.method;
126      this.requestHeaders = request.headers;
127      this.description = request.headers[CUSTOM_HEADERS.DESC_HEADER];
128      this.version = request.headers[CUSTOM_HEADERS.VERSION_HEADER] || this.
              guessedVersion() || 'Default';
129      this.resource = request.headers[CUSTOM_HEADERS.RESOURCE_HEADER] || this.
              guessedResource();
130      this.query = this.parsedUrl().query;
131      this.apiToken = this.requestHeaders[CUSTOM_HEADERS.API_TOKEN_HEADER];
132      this.filterAuthHeaders();
```

```javascript
133      this.action = this.computedAction();

134      return this.digest = this.computeDigest();

135    };

136

137    ApiExample.prototype.stripResponseBody = function() {

138      var StrippedObject, strippedObject;

139      if (!this.shouldStrip()) {

140        return;

141      }

142      StrippedObject = require('./stripped_object');

143      strippedObject = new StrippedObject();

144      try {

145        return this.strippedResponseBody = JSON.stringify(strippedObject.strip(
             JSON.parse(this.responseBody)));

146      } catch (_error) {

147        return this.strippedResponseBody = this.responseBody;

148      }

149    };

150

151    ApiExample.prototype.saveWithErrorLog = function() {

152      return this.save(function(error) {

153        if (error) {

154          return console.error("Failed to save because of error", error);

155        }

156      });

157    };

158

159    ApiExample.prototype.guessedVersion = function() {

160      return this.guessedVersionFromURL() || this.guessedVersionAcceptHeader();

161    };

162

163    ApiExample.prototype.guessedVersionFromURL = function() {
```

```
164    if (this.url == null) {
165      return null;
166    }
167    if (this.url.match(VERSION_IN_URL)) {
168      return RegExp.$1;
169    }
170  };
171
172  ApiExample.prototype.guessedVersionAcceptHeader = function() {
173    var matches;
174    if (this.requestHeaders['accept'] == null) {
175      return null;
176    }
177    matches = this.requestHeaders['accept'].match(VERSION_IN_HEADER);
178    if (matches != null) {
179      return matches[0];
180    }
181  };
182
183  ApiExample.prototype.parsedUrl = function() {
184    return url.parse(this.url, true);
185  };
186
187  ApiExample.prototype.guessedResource = function() {
188    var parts, possibleResource;
189    parts = this.parsedUrl().pathname.split('/').reverse();
190    possibleResource = _u.find(parts, function(part) {
191      return part.match(/^[^\d].*$/);
192    });
193    if (possibleResource != null) {
194      return possibleResource.split('.')[0];
195    }
```

```
196    };
197
198    ApiExample.prototype.computedAction = function() {
199      if (this.requestHeaders[CUSTOM_HEADERS.ACTION_HEADER] != null) {
200        return "" + this.http_method + " " + this.requestHeaders[CUSTOM_HEADERS.
              ACTION_HEADER];
201      } else {
202        this.templatizeURL();
203        return "" + this.http_method + " " + this.templatedURL;
204      }
205    };
206
207    ApiExample.prototype.setFullUrl = function(isSSL, hostPort) {
208      var host, port, scheme;
209      scheme = isSSL ? "https" : "http";
210      host = hostPort.host;
211      port = '';
212      if (hostPort.port) {
213        if (scheme === "http" && hostPort.port !== 80) {
214          port = ":" + hostPort.port;
215        } else if (scheme === "https" && hostPort.port !== 443) {
216          port = ":" + hostPort.port;
217        }
218      }
219      return this.fullURL = "" + scheme + "://" + host + port + this.url;
220    };
221
222    ApiExample.prototype.filterAuthHeaders = function() {
223      return this.requestHeaders = _u.reduce(this.requestHeaders, (function(
              _this) {
224        return function(filteredHeaders, value, key) {
225          if (key === 'authorization') {
```

```
226          console.log("value");

227          console.log(value);

228          if (value.match(/Basic/)) {

229            console.log(value);

230            _this.authType = 'Basic';

231            filteredHeaders[key] = 'Basic FILTERED';

232          } else {

233            filteredHeaders[key] = 'FILTERED';

234          }

235          _this.requiresAuth = true;

236        } else {

237          filteredHeaders[key] = value;

238        }

239        return filteredHeaders;

240      };

241    })(this), {});

242  };

243

244  ApiExample.prototype.filteredUrl = function(rawUrl) {

245    var api_key_param, parasedRawUrl;

246    if (!rawUrl) {

247      return rawUrl;

248    }

249    parasedRawUrl = url.parse(rawUrl, true);

250    api_key_param = _u.find(_u.keys(parasedRawUrl.query), function(param) {

251      return param.toLowerCase() === 'api_key';

252    });

253    if (!api_key_param) {

254      return rawUrl;

255    }

256    return rawUrl.replace("" + api_key_param + "=" + parasedRawUrl.query[
           api_key_param], "" + api_key_param + "=FILTERED");
```

```
257    };
258
259    ApiExample.prototype.computeDigest = function() {
260      var hash, text;
261      text = "__VERSION__" + this.version + "__RESOURCE__" + this.resource + "
              __URL__" + this.action + "__DESC__" + this.description;
262      hash = crypto.createHash('sha');
263      console.log("digest for text " + text);
264      hash.update(text);
265      return hash.digest('base64');
266    };
267
268    ApiExample.prototype.shouldStrip = function() {
269      var strip_header_value;
270      if (!this.requestHeaders) {
271        return true;
272      }
273      console.log("HERE");
274      strip_header_value = this.requestHeaders[CUSTOM_HEADERS.NO_STRIP_HEADER];
275      return strip_header_value !== 'true';
276    };
277
278    ApiExample.prototype.templatizeURL = function() {
279      var parts, path, templatedParts;
280      path = this.parsedUrl().pathname;
281      parts = path.split("/");
282      templatedParts = _u.map(parts, function(part, index) {
283        var previousPart;
284        previousPart = index > 0 ? parts[index - 1] : '';
285        if (validator.isNumeric(previousPart)) {
286          return part;
287        }
```

```
288        if (validator.isNumeric(part)) {
289            return "{:" + (inflector.singularize(previousPart)) + "-id}";
290        } else if (validator.isUUID(part)) {
291            return "{:" + (inflector.singularize(previousPart)) + "-guid}";
292        } else {
293            return part;
294        }
295    });
296    return this.templatedURL = templatedParts.join('/');
297  };
298
299  module.exports = ApiExample;
300
301 }).call(this);
```

Listing A.3: lib/api_example.js