

Automated Example Oriented REST API Documentation at Cisco

S M Sohan
Security Group
Cisco Systems Ltd.
Calgary, Canada
sosohan@cisco.com

Craig Anslow
Department of Computer Science
Middlesex University
London, United Kingdom
c.anslow@mdx.ac.uk

Frank Maurer
Department of Computer Science
University of Calgary
Calgary, Canada
frank.maurer@ucalgary.ca

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.

Keywords—API; REST; Documentation; Tool; Case study; Test; Automation; HTTP; Web API;

I. 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 its 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 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.

II. RELATED WORK

A. 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 requirements 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.

B. 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.

C. 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), RDDL, 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 Swagger¹. Listing 1 shows a fragment of

¹<http://raml.org/> <https://apibuildprint.org/> <http://swagger.io/>

REST API specification following swagger ².

Listing 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.

TABLE I: 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 I 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.

²<https://github.com/OAI/OpenAPI-Specification>

III. REST API DOCUMENTATION TOOL SELECTION

A. 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 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.

B. 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 II summarizes the findings:

TABLE II: 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 |

As shown in Table II, 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.

IV. 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 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: BasicFILTERED*.
- **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. 1 shows SpyREST screenshots of the production REST API documentation at Cisco. Fig. 1a shows a fragment of the manually written overview information juxtaposed to the auto-generated index of the API resources. Fig. 1b shows the output of the different analyzers within SpyREST proxy server that automatically detects the version (v1), API resource (Computer), action (PATCH /v1/computers/...), and the structure of the response fields from the single example API call. Fig. 1c shows the transformed example API request headers, body, and an executable cURL command that can be used to exercise the API call. Fig. 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.

V. CASE STUDY: SPYREST AT CISCO

A. 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.

B. Context

1) *The Stakeholders:* The REST API has several stakeholders as follows: customers, API client developers, customer

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/{:connector_guid}](#)

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

Examples

Moves computer to a group with given connector_guid and group_guid

Request

Requires Authorization

PATCH /v1/computers/ad29d359-dac9-4940-9c7e-c50e

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-a06053fbc"}' \
-u YOUR_API_CLIENT_ID \
'https://api.amp.cisco.com/v1/computers/ad29d359-dac9-4940-9c7e-c50e'
```

Body

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

(c) SpyREST Recorded API Request

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-8
```

```
{
  "version": "v1.0.0",
  "metadata": {
    "links": {
      "self": "https://api.amp.cisco.com/v1/"
    }
  },
  "data": {
    "connector_guid": "ad29d359-dac9-4940-9c7e-c50e",
    "hostname": "Demo_CozyDuke",
    "active": true,
    "links": {
      "computer": "https://api.amp.cisco.com/v1/computers/ad29d359-dac9-4940-9c7e-c50e",
      "trajectory": "https://api.amp.cisco.com/v1/computers/ad29d359-dac9-4940-9c7e-c50e/trajectory",
      "group": "https://api.amp.cisco.com/v1/groups/b077d6bc-bbdf-42f7-8838-a06053fbc"
    },
    "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",
  }
}
```

(d) SpyREST Recorded API Response

Fig. 1: Screenshots of the REST API documentation from Cisco using SpyREST

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

2) *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.

3) *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 Rails³ 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.

4) *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 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.

³<http://rubyonrails.org>

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.

C. The API Development Process

1) *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 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 2 provides an example of an automated test fragment from the case study that renders part of the documentation as shown in Fig. 1c 1d.

Listing 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 eq(202)
9       #more assertions
10    end
11  end
12 end

```

This example code is written using RSpec⁴, 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 the documentation**

⁴<http://rspec.info>

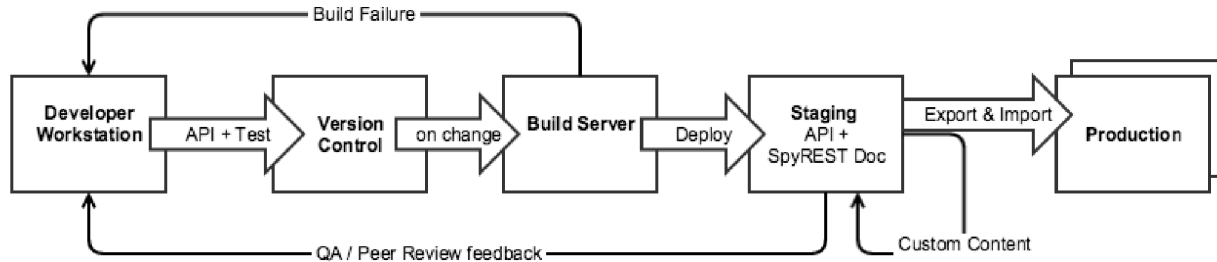


Fig. 2: API Development workflow using SpyREST at Cisco

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.

2) *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.

3) *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.

4) *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.

VI. DISCUSSION

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

B. 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 prose, 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.

C. 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.

D. 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.

E. 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.

F. 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” 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.

G. 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.

H. 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.

VII. 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

We express our gratitude to Cisco Systems Ltd. and the University of Calgary.

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 overflow," 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 RDDL 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.
- [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 Intelligent 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.