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
University of Calgary
Calgary, Canada
canslow@ucalgary.ca

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

***Abstract*—API documentation presents both a problem and an opportunity for API usability. Representational State Transfer, or more commonly known as REST based APIs are used by software developers to interconnect applications over HTTP. Developers are required to publish and maintain the documentation of their REST APIs so that other developers can learn and use the APIs as intended. This poses the problem of identifying an efficient and effective process of generating and maintaining the documentation of REST APIs. In this paper, we have discussed our lessons learned from a case study comprising of the production use of an automated example oriented REST API documentation approach using a tool called SpyREST at Cisco over a period of eighteen months. We have observed that continuously updated documentation can be achieved by using automated test code against a REST API. Practitioners can leverage the insights shared in this paper as a guideline to improve the state of their REST API documentation process. Researchers and tool developers can incorporate the ideas from this case study to extend the example oriented documentation approach to APIs beyond the realm of REST APIs.**

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

I. 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 describes API as follows: An API is the interface to implement functionality that developers can access to perform various tasks [1] [2].

REST APIs are a subclass of APIs that use standard web technologies for interconnectivity over HTTP. Fielding defined Representational State Transfer or REST as an architectural style for developing distributed hypermedia systems [3]. REST APIs provide an abstraction of the underlying system by using system specific resources such as documents, images, etc. and unique identifiers to access the resources. Using REST APIs, systems perform actions on the resources by transferring a representation of the resources between various systems. For example, the GitHub REST API[2] has a resource called $Repository$ to denote a code repository hosted on GitHub.

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

Researchers identified the documentation of APIs as both the primary source of information as well as the key obstacle for API usability [2]. 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], [4]. 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.

The case study in this paper presents an evaluation of SpyREST in the industry. SpyREST is being used in production at Cisco for the documentation of a commercial REST API of a cloud based Cyber security product that the first author of this paper is affiliated with. 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. Despite the popularity and adoption of REST APIs, there is a lack of published work about the API documentation techniques used by today's internet companies. 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 technique for maintaining the evolution of API documentation as the API evolves without duplicating effort.
- **Implications for future work**. We show a practical evidence that API documentation can be generated by intercepting and transforming example API calls. Researchers can leverage this 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 a literature review 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

API usability is a qualitative concept derived from the characteristics that make an API easy to use. 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 [1]. Robillard 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. While API documentation is used as the principal source of information about how to use an API, Robillard et al. found that the most severe API learning obstacles are related to the API documentation. For API usability, Robillard 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. Zibran et al. analyzed bug repositories for 562 API usability related bugs from five different projects and found that 27.3% of the reported bugs are API documentation bugs [5]. Scheller et al. provided a framework for objectively measuring API usability based on the number and types of different objects and methods that the API provides [6].

Kuhn performed a user study with 19 professional software developers to understand requirements for tool development to support API learnability [7]. Kuhn 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. observed a large number of API documentation changes related to polishing the custom content, (i.e. fix typos, and API usage examples) [8]. They recommended API documentation tools to support editors for custom content to provide usage tips and simple ways to include API usage examples without syntax errors. Ko et al. found that thorough introductions to the concepts, standards and ideas in API documentation are a prerequisite for developers to be able to effectively use an API [9].

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 [10]. They introduced Roast test as tool support to combine prosaic descriptions of Java APIs along with executable code examples in a unified API documentation. Montandon et al. developed APIMiner as a search tool for Java and Android APIs and recommended providing production-like API usage examples in the API documentation [11]. They observed that 35% of API related web searches performed on APIMiner included the term âĂIJexampleâĂİ inferring that developers search for source code examples while using API documentation. Zhu et al. developed an Eclipse plugin called UsETeC to extract API usage examples by automatically synthesizing JUnit test code of the APIs [12]. Stylos presented Jadeite as an IDE plug-in that combines a few techniques to provide developers with faster access to relevant API documentation [13]. Jadeite uses placeholders for API elements such as classes and method names that developers commonly expect to exist, but the actual classes or methods are named differently. When developers search for placeholder API elements, Jadeite shows links to the API documentation of the actual API elements and finds relevant usage examples from Google code search.

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 [14]. Parnin et al. found that examples of 87.9% of all jQuery API methods are found by searching software development blogs and forums [15]. 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 [16]. 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 [17]. Dagenais et al. presented a technique and a tool called RecoDoc to link code-like elements from API mailing lists and developer forums with their corresponding code elements in the API documentation [18]. Treude et al. presented a machine learned based technique called SISE to augment useful information from StackOverflow to API documentation by using text similarity of API elements and StackOverflow content [19].

As mentioned above, we observed that the research on API documentation related tools have focused on local APIs such as Java library APIs. We found a lack of published work on the tool support for including usage examples with REST API documentation.s

C. REST API Documentation

Maleshkova analyzed the state of REST APIs and found that most REST APIs are manually documented which results in API underspecification, and a lack of support for common tasks and reusable tools [20]. Myers et al. performed a user study on the usability of a complex API for enterprise SOA [4]. 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 case study, we found the documentation of REST APIs are performed manually or using bespoke tools [21]. We observed the documentation of the studied REST APIs to commonly include summary information and API examples, with optional description of the structure of API requests and responses.

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. Hadley proposed WADL (Web Application Description Language) [22]. Mangler et al. proposed RIDDL as an extension of WADL to support composition and evolution of REST APIs [23]. Kopecky et al. proposed hRESTS, as an HTML based mico-format to describe REST APIs [24]. Verborgh et al. proposed RESTdesc as a language to define REST APIs using resources and links that connects the resources [25]. Danielsen presented a vocabulary to describe REST APIs using WIfL(Web Interface Language) [26]. Lei et al. presented OmniVoke as a tool to abstract out multiple REST APIs under a unified interface following a specification [27]. Polak proposed a specification format for REST API using the Model-Driven Architectural principle where the evolution of a REST API can be programmatically managed from it's model representation [28].

In addition to the existing literature, several REST API description languages have been proposed by industry practitioners such as RAML[3], Blueprint[4], and Swagger[5]. There are tools that can automatically convert and publish the description of REST APIs from these languages into HTML based REST API documentation.

The primary advantage of these specification languages is code generation and automatic transformation into REST API documentation. On the other hand, we found a lack of automated tool support for producing the API specifications for REST APIs.

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 candidate specifications to standardize the vocabulary and format of describing REST APIs. Manual work is needed by REST API developers to generate and maintain their API specifications as it evolves. These specification formats support describing the structure of different API elements (the syntax), but no support is provided for API usage examples and executable API examples (the semantics), an important attribute for API learnability as identified by the researchers. Also, we found a lack of published papers on the effectiveness of the aforementioned specification languages in an industry setting. In our work with SpyREST, instead of relying on specification, we have focused on automatically intercepting and transforming example REST API calls into usable documentation. In this paper we have shared the lessons learned of using this new technique at Cisco.

[3]http://raml.org/
[4]https://apiblueprint.org/
[5]http://swagger.io/

III. Overview of SpyREST

We provide a brief overview of SpyREST, the REST API documentation tool and the underlying technique used in this case study. In our previous papers, we have discussed the design and implementation of this tool in greater detail [29], [30].

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 furthered 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 a 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 inspects the URL and a request header named *Accept* : to automatically infer 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. It uses regular expressions to match the URL and *Accept* header against the commonly used API versioning formats.

- **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 parses the API URL and detects the rightmost non-numeric part of the URL path as the API resource. For the given example, SpyREST detects $post$ as the API resource.
- **API action analyzer**. Each API resource can be accessed or modified via the API actions. The API action analyzer infers the API action from example API calls by combining the HTTP verb (e.g. $GET, POST$) with an auto-templated representation of the path. For the given example, the action analyzer identifies $POST/v2/posts$ as the API action. If the path contains numeric or uuid parameters, the a templated path is used. For example, the API action analyzer detects a GET request to the path $/v2/posts/1/comments/10$ as the API action $GET/v2/posts/:post_id/comments/:comment_id$. The templated API actions offer a more meaningful representation of the intent, and allows multiple API examples to be grouped under a single 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. For each API action, a query parameter table is shown where each row displays the name, auto-detected data type, and example values for each query parameter.
- **API request header analyzer**: The request header analyzer strips the $Authorization$ header to remove confidential credentials from the API documentation. Additionally, the request header analyzer automatically detects the type of authorization used, e.g. Basic, Token, etc. 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 maintains a list of commonly found response headers (e.g. host, pragma, server, etc.) that add little value to API documentation and automatically removes them from the documentation.
- **Custom content analyzer**: The version, resource, and action analyzers detect commonly seen patterns and extracts the relevant information. The result is also used to automatically infer a human readable description for each API action. For the given example, the auto detected description is as follows: $CreateaPost$ by combining the HTTP verb ($POST$) with the API resource ($post$). When unfamiliar formats are used by an API or customization is required, the custom content analyzer allows API developers to override each of the auto-detected attributes such as the API version, resource, action and description by using a set of SpyREST specific request headers (e.g. $x-spy-rest-version, x-spy-rest-resource$, etc.).

As shown before, SpyREST uses the aforementioned analyzers to transform the raw HTTP request and response information from example REST API calls into a structured representation of the API as follows: A REST API has one or more versions, each version has one or more API resources, each resource has one of more actions, and each action is defined by its URL, query parameters, request and response headers and bodies, and one or more usage examples.

SpyREST has a web based UI to render a hierarchical representation of the aforementioned structured data as the API documentation. The web UI also features a wiki-like editor on each page so that human written detailed descriptions can be added with rich content to explain business concepts. In addition to displaying the structured API documentation, the web UI also features an executable code using cURL [6], a commonly used tool for accessing resources over HTTP, for each captured API example so that the API client developers can try the API examples without having to write custom code.

To generate usable API documentation, REST API developers need to run appropriate example API calls via SpyREST proxy server. Once a set of example API calls are defined, the documentation can be auto-updated by replaying the example API calls. The replay only updates the auto-generated part of the documentation leaving the human written descriptions unchanged.

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.

IV. SpyREST at Cisco

A. The API

SpyREST has been used to document the REST API for a cloud delivered cyber security software at Cisco. The API allows the customers of the cyber security software 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.

At present, the API is served under two different versions, v0 and v1. The API is also served from three different Cisco managed cloud backend that are geographically separated for North America, Europe and Asia Pacific. Additionally, the API is also shipped with a private cloud appliance, where

[6]https://curl.haxx.se/

customers can host the security software on-premise outside the Cisco managed cloud instances. Even though the different environments are designed to eventually serve the API with identical features, the deployment of the API to each environment is independent of the rest of the environments.

In the latest version, v1, the API allows the API client developers to interact with nine different API resources, and a total of twenty API actions 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, that are also present in the v1.

The API was first launched with version v0 in February 2015, and the version v1 was released in May, 2016. The API has evolved several times while keeping the version numbers unchanged to release backward compatible new features and bug fixes. At present, the API serves approximately 80,000 API calls per day on an average and 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.

B. 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. This adds further value to the purchased cyber security software to the customer. API client developers are often employed by the customers to implement API integrations. The API client developers are interested in the technical details about the API and actively use the API documentation to implement the custom features for the customers. At present, over a hundred unique customers use the API at least once a month.

Inside Cisco, the stakeholders are the customer support team, product management team, API developers and quality assurance engineers. The customer support team connects the developers and quality assurance engineers with the customers to establish a closed communication loop for API related requirements and customer feedback about released APIs. Product management team is responsible for providing input about API requirements and scheduling the release of API updates. API developers are Cisco employees that are in charge of implementing the API and it’s documentation. The quality assurance engineers are responsible for developing and verifying the acceptance criteria for the REST API according to the requirements. At present, a total of 25 people, including the first author, are involved as the Cisco stakeholders for the REST API of the cyber security product in this case study.

C. The Process

1) REST API Documentation Tool Selection: The API developers and QA engineers participated in the selection of an appropriate REST API documentation tool. The following wish-list of API documentation features were collected during meetings involving the stakeholders:

- R1. Support multiple versions
- R2. Support multiple environments
- R3. Easy to maintain
- R4. Include executable API usage examples
- R5. Use familiar tools
- R6. Allow customization

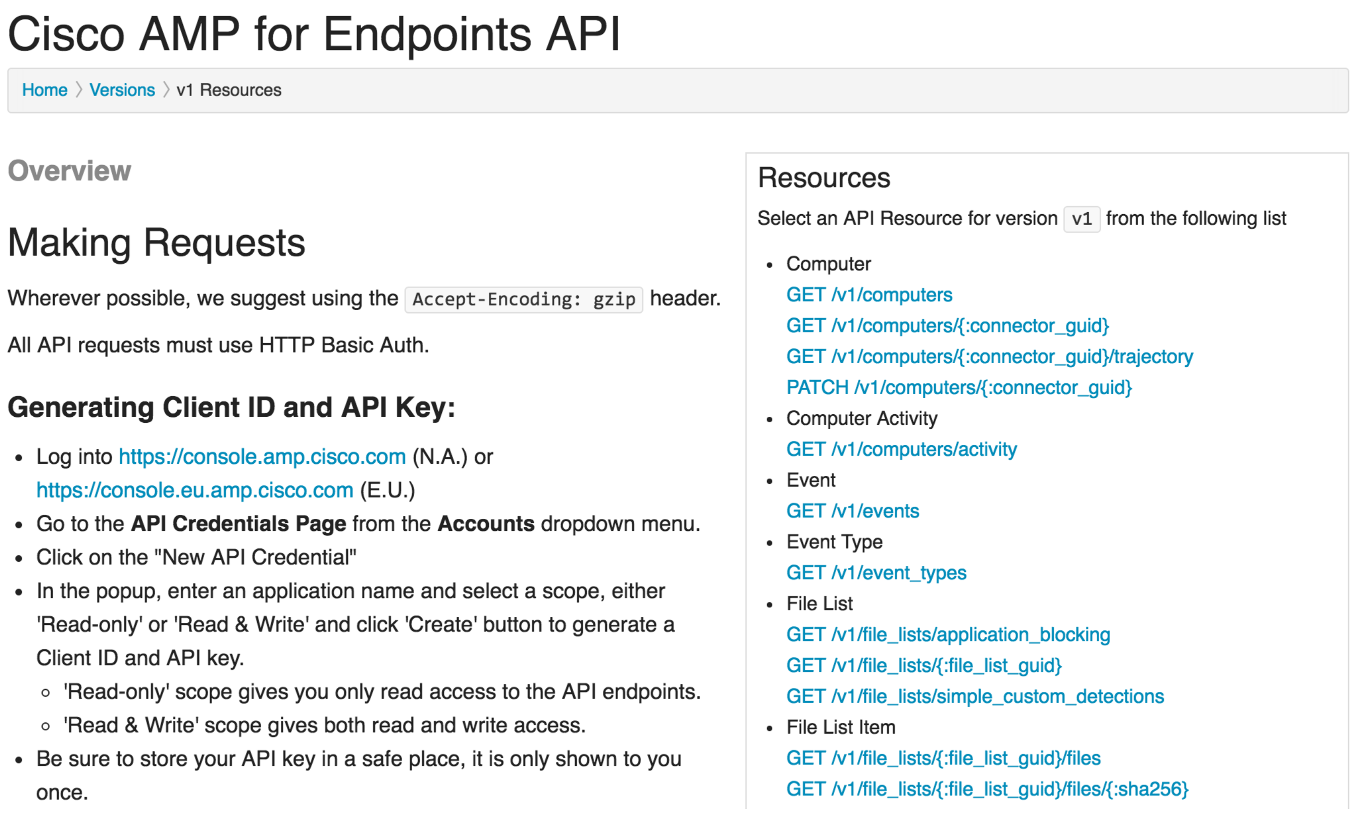
A list of REST API documentation tools were selected for “spikes”, a practice used by the software engineers to better understand a tool and how it fits a the problem in hand. The API developers and QA engineers shared informal opinions based on their spikes about about three different tools, Swagger, API Blueprint and SpyREST. Swagger and Blueprint were researched because of their perceived maturity and industry adoption information as found through online searches and StackOverflow.com activities. The first author in this paper introduced SpyREST to the teams as a candidate solution to consider.

The verdict to use SpyREST was based on the observation that using Swagger or API blueprint required us to maintain the REST API documentation as a separate but parallel artifact from the API related code. The available tool support for Swagger or API Blueprint specifications had little support for Ruby on Rails, the primary programming language used by the team to develop it’s products. As a result, it’d require manual effort to keep the two in sync as the API evolves. 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 that are already written and maintained with the code. A round of informal feedback on SpyREST was collected after the first production release of the API. The teams decided to continue using SpyREST for API documentation.

2) 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. 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. In the following, we provide an example of an automated test fragment from the case study:

```
context 'v1' do
  describe 'Computer' do
    it 'Moves computer to a group with
        given connector_guid and
        group_guid' do

      response = API.patch("/v1/
          computers/#{connector_guid}",
          {
```


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:

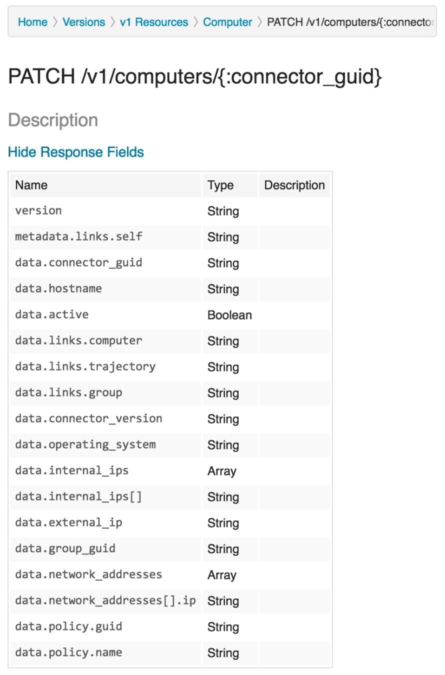
- Log into https://console.amp.cisco.com (N.A.) or https://console.eu.amp.cisco.com (E.U.)
- Go to the **API Credentials Page** from the **Accounts** dropdown menu.
- Click on the "New API Credential"
- In the popup, enter an application name and select a scope, either 'Read-only' or 'Read & Write' and 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

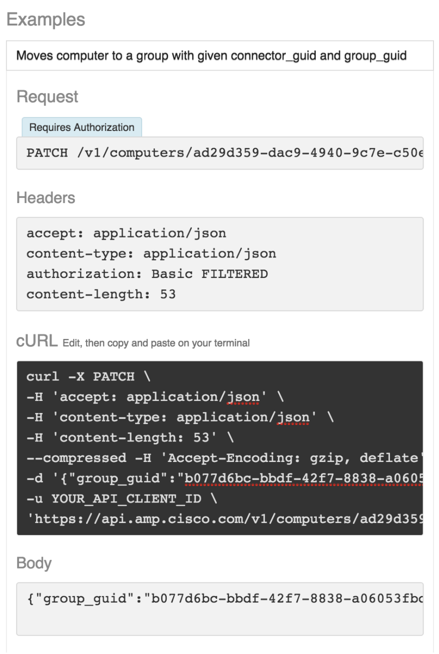
PATCH /v1/computers/{:connector_guid}

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


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
```

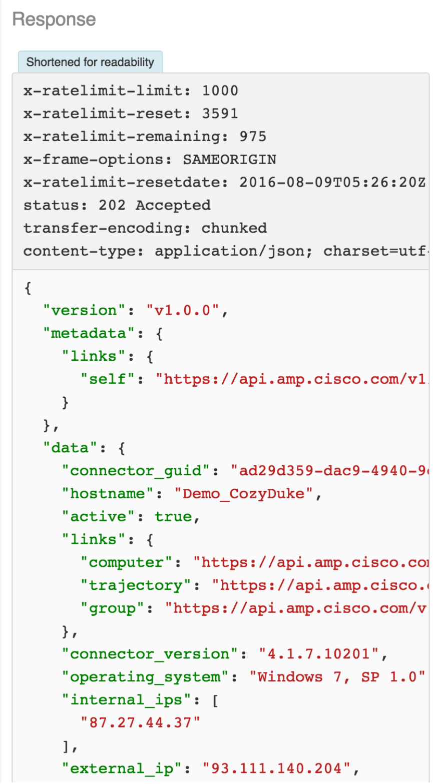
cURL Edit, then copy and paste on your terminal

```
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-a0605
-u YOUR_API_CLIENT_ID \
'https://api.amp.cisco.com/v1/computers/ad29d359
```

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
```

```
{
  "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",
```

(d) SpyREST Recorded API Response

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

```
6             headers: 'x-spy-rest-action'
                  => '/v1/computers/{:
                  connector_guid}',
7             body: { group_guid: group_guid
                  }.to_json
8           })
9
10          expect(response.code).to eql
                (202)
11          #more assertions
12        end
13      end
14    end
```

This example code is written using RSpec[7], 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 5-8, an example HTTP patch API request is made using the method $API.patch$. Then, an example assertion is added on line 10. The test is written similar to other RSpec tests.

However, 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, x-spy-rest-description, to the proxy server. As discussed earlier, the SpyREST analyzers can inspect the HTTP request and response information with the custom headers to produce a usable REST API documentation from this automated test code.

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: "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 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, it's 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.

[7]http://rspec.info

Fig. 1 shows screenshots of the production REST API documentation generated from the aforementioned functional test suite. 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/compouters/...), 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.

3) 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. The rest of the lifecycle steps follow similar process as the documentation of a new API action.

4) Documentation of an updated API Action: When an API update requires a new example, a new automated test case is written. For example, in the case study, 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.

5) REST API Evolution: In the case study we have observed the evolution of the API and it's documentation is triggered by the following: 1) new requirements, and 2) internal reviews of the API. For example, the primary difference between the 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 through 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 has helped establish a quick feedback loop between the API development and QA teams and the ability to make frequent changes to the API and its documentation throughout the life-cycle of the API.

V. 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. We have found the documentation of the REST API from its functional test code to be a welcome side-effect. When the data from the tests are intercepted to generate API documentation, it also helps improve the quality of the test code. 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 feedback. REST API documentation from the test code also requires a consistently repeatable set of API usage examples so that each run of the test suite can be used in the documentation if the tests pass. This enforces the need for a stable test suite.

The continuously updated documentation improves its correctness and verification of version compatibility as an API evolves. A breaking change in the API is likely to fail 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, we have observed that not all API tests are suitable to be used in the REST API documentation. For example, while it is common to add both happy and unhappy paths in the automated tests, unhappy paths are seldom included in the documentation to reduce information overload. To support such differences, API developers may need to select a subset of the tests that are run through SpyREST proxy server to be included in the 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.

B. Maintaining Custom Content

We’ve observed both the benefits and drawbacks of using a wiki-like editor for custom content with the API documentation. It allows the API developers to add rich content comprising of proses, images, tables, code fragments, web links, etc. to explain concepts that are required to understand the API. The juxtaposition of custom content with auto-generated content complements each other. 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 a custom edited content.

The custom content using a wiki has a drawback as it’s maintained as a separate artifact within SpyREST outside the API code repository. As a result, the version control related features that are available to the code are not applicable to the custom content. If desired, API developers need to implement a versioning system for the custom content that is not currently supported by SpyREST.

For researchers, 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 benefits of a version control system can be utilized.

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, etc. On the other hand, a network detection event contains network specific data such as the remote host, URL, IP, protocol, etc. A single API call may return a list of “Events” of such different types. In our case study at Cisco, the SpyREST documented API describes the “Event” API element as an amalgamation 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’ve observed both the advantages and disadvantages of automatically merging different attributes of the API elements within an array to construct the structure of an API element.

The primary advantage is, it provides a list of all possible attributes that are observed in an API element. So, an API client developer reading the API documentation gets a complete picture of the API element. The primary disadvantage is, 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 within the context of specific use cases.

D. API Documentation Life-cycle

We found it to be 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, a corresponding documentation can be generated from the accompanying test code.

When a REST API documentation is published from a staging environment to production, the data from the API examples may need to be obfuscated and transformed. SpyREST automatically rewrites 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 need to be obfuscated by the API developers. 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 this confidential data, we run the tests against a sandbox API environment that serves dummy data. For practitioners and API documentation tool developers, we suggest incorporating this idea of continuous API documentation into consideration 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. As discussed before, API evolves more frequently before it's published to production. To provide feedback and verify the changes during development, a changelog is often necessary as a communication artifact within the development and QA teams. SpyREST can automatically detect API changes when API 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 from the API and its 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 but often leaves the details of a change to be discovered by the API client developers. For example, in the v1 changelog, we mentioned the addition of a new feature to "move a computer from one group to another", but the actual difference in the computer API response wasn't discussed in the changelog. For researchers, we identify this as an opportunity for future work to extend automation support for API changelog generation for evolving 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. One of the developers from the Cisco team asked the following question about the SpyREST auto-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 identify several possible improvements to the auto-generated navigation experience from this case study.

Cross-referencing the API elements may provide alternate 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. This dependency relationship is not captured within the SpyREST generated API navigation. Future research needs to be carried out to automatically mine and surface such dependency relationships among API elements to help the API client developers.

G. Extending Beyond REST API Documentation

In our case study, we've found encouraging results of the feasibility of using interception as a technique to generate REST API documentation with usage examples. While an HTTP proxy server may not be used to intercept non-HTTP APIs, the core concept of interception can be used across such APIs. For example, to document a local API, a custom module can be written to intercept example API calls from the unit test code in memory. The interception technique may provide a general solution to automatically generate usage examples in API documentation from executable code.

We identify several benefits of this approach over publishing the existing unit test code as a documentation. First, organizations may not allow publishing their unit test code to external API users due to intellectual property related policies. Second, even if unit tests are published as API documentation, the API client developers may not be familiar with the unit test framework used and other 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 API documentation. Using an interception module, the actual implementation of the API examples can be abstracted out only to record and document the actual API call that the API developers want to document. We identify this as a potential research topic for researchers to implement and evaluate such interception modules for non-REST APIs to understand the feasibility of this technique beyond the realm of REST APIs.

H. Limitations

We identify several limitations of our case study as well as the interception based REST API documentation technique used in the case study. 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 opportunity to study the industry adoption of the interception based REST API documentation tool developed in research, it also introduces a confirmation bias. In our future work, we'll perform further evaluation of the API documentation technique involving a larger sample of software engineers from different organizations experienced on REST APIs.

The results discussed in this paper may include a selection bias since the results are based on a single organization and a single set of REST APIs that only used SpyREST as the documentation tool. While the team initially experimented with three different REST API documentation techniques, the experiment was not replicated after the API was released to understand if the findings from the "spikes" matched the actual API. This case study needs to be replicated against REST APIs that are implemented by other organizations using different technologies to reduce this bias.

VI. Conclusion

In this case study, we've presented our lessons learned from using a new REST API documentation technique at Cisco to document an API with multiple versions that are deployed to multiple environments. Our primary findings provide an evidence that continuously updated usage based REST API documentation can be generated using SpyREST by leveraging automated functional tests. We've also discussed techniques to augment and maintain auto-generated REST API documentation with human edited custom content. For API elements with flexible structure, we've found a need to include specific examples in the API documentation to show the context around the different possible structures. We have discussed the importance and the technique of including the documentation of the REST API with every step of the API life-cycle to establish a quick feedback loop. We have presented our case for extending the interception based API documentation technique beyond REST APIs for researchers to conduct future studies.

In our future work, we plan to improve the state of REST API documentation tool support by incorporating the ideas from this case study. Specifically, we will perform an experiment to evaluate the impact of usage examples on REST API client developers. We will continue our research on SpyREST to automatically infer the relations among REST API elements and their evolution to improve the tool support for alternate navigation experience and changelog generation for REST APIs.

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] 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.

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

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

[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] 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.

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

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

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

[12] 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.

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

[14] 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.

[15] 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.

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

[17] 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.

[18] B. Dagenais and M. P. 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.

[19] C. Treude and M. P. Robillard, "Augmenting api documentation with insights from stack overflow," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 392–403. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884800

[20] 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.

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

[22] M. J. Hadley, "Web application description language (wadl)," 2006.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] M. Polák and I. Holubová, "Rest api management and evolution using mda," in *Proceedings of the Eighth International C* Conference on Computer Science & Software Engineering*, ser. C3S2E '15. New York, NY, USA: ACM, 2008, pp. 102–109. [Online]. Available: http://doi.acm.org/10.1145/2790798.2790820

[29] S. M. Sohan, C. Anslow, and F. Maurer, "Spyrest: Automated restful API documentation using an HTTP proxy server (N)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, 2015, pp. 271–276. [Online]. Available: http://dx.doi.org/10.1109/ASE.2015.52

[30] ——, “Spyrest in action: An automated restful API documentation tool,” in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, 2015, pp. 813–818. [Online]. Available: http://dx.doi.org/10.1109/ASE.2015.92