Universitat Oberta de Catalunya

uoc.edu

Citation for published version

Ed-douibi, H., Cánovas Izquierdo, J.L. & Cabot, J. (2017). Example-driven web API specification discovery. Lecture Notes in Computer Science, 10376(), 267-284.

DOI
https://doi.org/10.1007/978-3-319-61482-3_16

Document Version

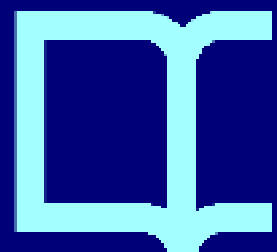
This is the Submitted Manuscript version.
The version in the Universitat Oberta de Catalunya institutional repository, O2 may differ from the final published version.

Copyright and Reuse

This manuscript version is made available under the terms of the Creative Commons Attribution Non Commercial No Derivatives licence (CC-BY-NC-ND) http://creativecommons.org/licenses/by-nc-nd/3.0/es/, which permits others to download it and share it with others as long as they credit you, but they can't change it in any way or use them commercially.

Enquiries

If you believe this document infringes copyright, please contact the Research Team at: repositori@uoc.edu



Example-driven Web API Specification Discovery

Hamza Ed-douibi[1](0000-0003-4342-4818), Javier Luis Cánovas Izquierdo[1](0000-0002-2326-1700), Jordi Cabot[1,2](0000-0003-2418-2489)

[1] UOC. Barcelona, Spain
{hed-douibi,jcanovasi}@uoc.edu
[2] ICREA. Barcelona, Spain
jordi.cabot@icrea.cat

Abstract. REpresentational State Transfer (REST) has become the dominant approach to design Web APIs nowadays, resulting in thousands of public REST Web APIs offering access to a variety of data sources (e.g., open-data initiatives) or advanced functionalities (e.g., geolocation services). Unfortunately, most of these APIs do not come with any specification that developers (and machines) can rely on to automatically understand and integrate them. Instead, most of the time we have to rely on reading its ad-hoc documentation web pages, despite the existence of languages like Swagger or, more recently, OpenAPI that developers could use to formally describe their APIs. In this paper we present an example-driven discovery process that generates model-based OpenAPI specifications for REST Web APIs by using API call examples. A tool implementing our approach and a community-driven repository for the discovered APIs are also presented.

Keywords: REST Web APIs, Discovery process, OpenAPI, Repository

1 Introduction

Web APIs are becoming the backbone of Web, cloud, mobile applications and even many open data initiatives. For example, as of February 2017, ProgrammableWeb lists more than 16,997 public APIs. REST is the predominant architectural style for building such Web APIs, which proposes to manipulate Web resources using a uniform set of *stateless operations* and relying only on simple URIs and HTTP verbs.

Despite their popularity, REST Web APIs do not typically come with any precise specification of the functionality or data they offer. Instead, REST "specifications" are typically simple informal textual descriptions [11] (i.e., documentation pages), which hampers their integration in third-party tools and services. Indeed, developers need to read documentation pages, manually write code to assemble the resource URIs and encode/decode the exchanged resource representations. This manual process is time-consuming and error-prone and affects not only the adoption of APIs but also its discovery so many web applications are missing good opportunities to extend their functionality with already available APIs.

Actually, languages to formalize APIs exist, but they are barely used in practice. Web Application Description Language (WADL) [6], a specification language for REST Web APIs was the first one to be proposed. However, it was deemed too tedious to use

and alternatives like Swagger[3], API Blueprint[4] or RAML[5] quickly surfaced. Aiming at standardizing the way to specify REST Web APIs, several vendors (e.g., Google, IBM, SmartBear, or 3Scale) have recently announced the OpenAPI Initiative[6], a vendor neutral, portable and open specification for providing metadata (in JSON and YAML) for REST Web APIs.

This paper aims to improve this situation by helping both API builders and API users to interact with (and discover) each other by proposing an approach to automatically infer OpenAPI-compliant specifications for REST Web APIs, and, optionally, store them in a community-oriented directory. From the user's point of view, this facilitates the discovery and integration of existing APIs, favouring software reuse. For instance, API specifications can be used to generate SDKs for different frameworks (e.g., using APIMATIC[7]). From the API builder's point of view, this helps increase the exposure of the APIs without the need to learn and fully write the API specifications or alter the API code, thus allowing fast-prototyping of API specifications and leveraging on several existing toolsets featuring API documentation generation (e.g., using Swagger UI[8]) or API monitoring and testing (e.g., using Runscope[9]).

Our approach is an example-driven approach, meaning that the OpenAPI specification is derived from a set of examples showing its usage. The use of examples is a well-known technique in several areas such as Software Engineering [8, 10] and Automatic Programming [5]. In our context, the examples are REST Web API calls expressed in terms of API requests and responses.

We follow a metamodeling approach [1] and create an intermediate model-based representation of the OpenAPI specifications before generating the final OpenAPI JSON Schema definition[10] for two main reasons: i) to leverage the plethora of modeling tools to generate, transform, analyze and validate our discovered specifications (as existing JSON schema tools are limited and may produce contradictory results [12]); and ii) to enable the integration of APIs into model-driven development processes (for code-generation, reverse engineering,..). For instance, we envision designers being able to include API calls in the definition of web-based applications using the Interaction Flow Modeling Language (IFML) [2].

The remainder of this paper is structured as follows. Section 2 show the running example used along the paper. Section 3 presents the overall approach and then Sections 4, 5 and 6 describe the OpenAPI metamodel, the discovery process and the generation process, respectively. Section 7 describes the validation process and limitations of the approach. Section 8 presents the related work. Section 9 describes the tool support, and finally, Section 10 concludes the paper.

---

[3] http://swagger.io/
[4] https://apiblueprint.org/
[5] http://raml.org/
[6] https://openapis.org
[7] https://apimatic.io/
[8] http://swagger.io/swagger-ui/
[9] https://www.runscope.com/
[10] https://github.com/OAI/OpenAPI-Specification/blob/master/schemas/v2.0/schema.json

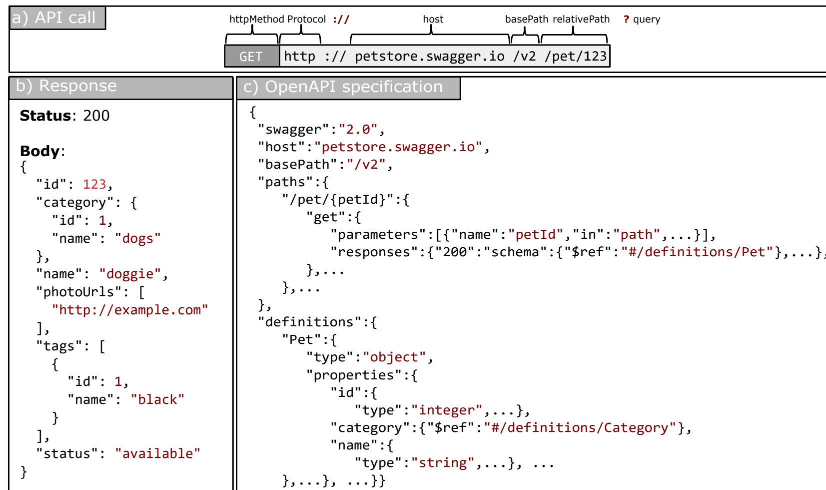


Fig. 1. API call example of the Petsore API: (a) the request, (b) the response, and (c) an excerpt of the corresponding OpenAPI specification.

2 Running Example

This section introduces the running example used along the paper together with the main elements of a REST Web API. The example is based on the Petstore API, a REST Web API for a pet store managament system, released by the OpenAPI community as a reference. This API allows users to manage pets (e.g., add/find/delete pets), orders (e.g., place/delete orders), and users (e.g., create/delete users). Figure 1 shows an excerpt of this API specification, an API access request and a possible response document for that call request.

Figure 1a shows the request to retrieve the pet with the id `123` while Figure 1b shows the returned response with that pet information. A request includes a method (e.g., `GET`), a URL (e.g., `http://petstore.swagger.io/v2/pet/123`) and optionally a message body (empty for this example). The URL in turn includes: (i) the transfer protocol, (ii) the host, (iii) the base path, (iv) the relative path and (v) the query (indicated by the first question mark "`?`", empty for this example). The relative path and the query are optional. A response includes a status code (e.g., `200`) and optionally a JSON response message. Figure 1c shows an excerpt of the OpenAPI-compliant specification for this example call in JSON format. This document includes fields to specify properties such as the host, the base path, the available paths (i.e., the field `paths`), the supported operations for each path (e.g., the field `get`), and the data types produced and consumed by the API (i.e., the field `definitions`). The specification indicates that the `GET` operation of the path `/pet/{petId}` allows retrieving a pet by his ID.

3 Approach

We define a two-step process to discover OpenAPI-compliant specifications from a set of REST Web API call examples. Figure 2 shows an overview of our approach.

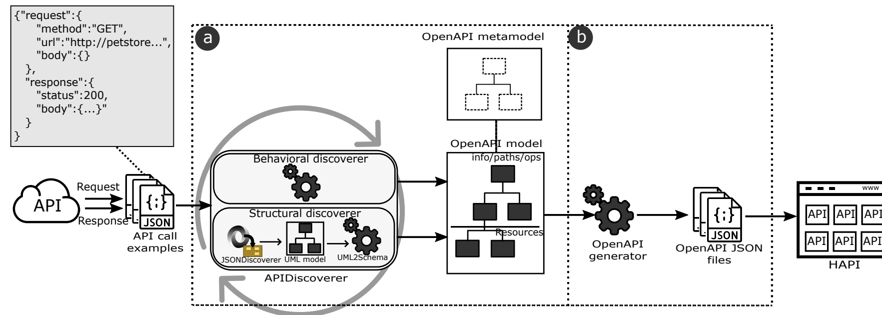


Fig. 2. Overview of the approach.

The process takes as input a set of API call examples. For the sake of simplicity, we assume examples are provided beforehand and later in Section 9 we describe how we devised a solution to provide them both manually and relying on other sources. These examples are used to build an OpenAPI model (see Figure 2a) in the first step of the process. Each example is analyzed with two discoverers, namely: (1) behavioral and (2) structural targeting the corresponding elements of the API definition. The output of these discoverers is merged and added incrementally to an OpenAPI model, conforming to the OpenAPI metamodel presented in the next section. The second step transforms these OpenAPI models to valid OpenAPI JSON documents (see Figure 2b).

To represent the API call examples themselves, we rely on a JSON-based representation of the request/response details. Both, the request sent to the server and the received response message, are represented as JSON objects (i.e., `request` and `response` fields in left upper box of Figure 2). The request object includes fields to set the method, the URL and the JSON message body; while the response object includes fields for the status code and the JSON response message. This JSON format helps to simplify the complexity of directly using raw HTTP requests and responses (which would require to perform HTTP traffic analysis) and facilitate the provision of examples by end-users. As discussed later, we provide also tool support to provide API call examples and even to (semi)automatically derive them from other sources, like existing documentation.

As a final step, the resulting OpenAPI-compliant specifications may optionally be added to HAPI, our community-driven hub for REST Web APIs, where developers can search and query them. In the following sections we describe our OpenAPI metamodel, the discoverers, and the OpenAPI generator. The example providers, APIs importers, and HAPI will be explained in Section 9.

4 The OpenAPI Metamodel

This section presents the OpenAPI metamodel to specify REST Web APIs. In a nutshell, a metamodel describes the set of valid models for a language, specifying how the different elements of the modeling language can be used and combined [1].

This model-based approach to define and store internally OpenAPIs facilitates the integration of our approach with model-based development methods and facilitates the manipulation of such OpenAPI specifications before the final generation of the

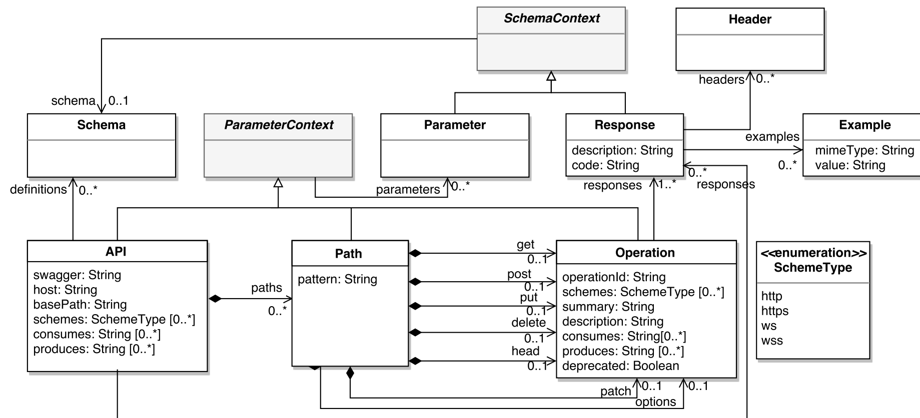


Fig. 3. Behavioral elements of the OpenAPI metamodel.

corresponding JSON documents. Such features are not provided by the JSON Schema definition of OpenAPI[11], which is limited to be used to validate documents against the original specification; or existing implementations (e.g., the Java model for OpenAPI[12]), which generally consist of a set of *POJOs* to serve as parsing facilities.

The metamodel is derived from the concepts and properties described in the OpenAPI specification document. Next we explain the main parts of this metamodel, namely: (1) behavioral elements, (2) structural elements, and (3) serialization/deserialization elements. The metamodel also includes support for metadata (e.g., description or version) and security aspects. The complete metamodel, comprised of 29 different metaclasses, is available in our repository[13].

4.1 Behavioral Elements

Figure 3 shows the behavioral elements of the OpenAPI metamodel. A REST Web API is represented by the `API` element, which is the root element of our metamodel. This element includes attributes to specify the version of the API (`swagger` attribute), the host serving the API, the base path of the API, the supported transfer protocols of the API (`schemes` attribute) and the list of MIME types the API can consume/produce. It also includes references to the available paths, the data types used by the operations (`definitions` reference) and the possible responses of the API calls.

The `Path` element contains a relative path to an individual endpoint and the operations for the HTTP methods (e.g., `get` and `put` references). The description of an operation (`Operation` element) includes an identifier `operationId`, the MIME types the operation can consume/produce, and the supported transfer protocols for the oper-

[11] `https://github.com/OAI/OpenAPI-Specification/blob/master/schemas/v2.0/schema.json`
[12] `https://github.com/swagger-api/swagger-core`
[13] `https://github.com/SOM-Research/APIDiscoverer/tree/master/metamodel`

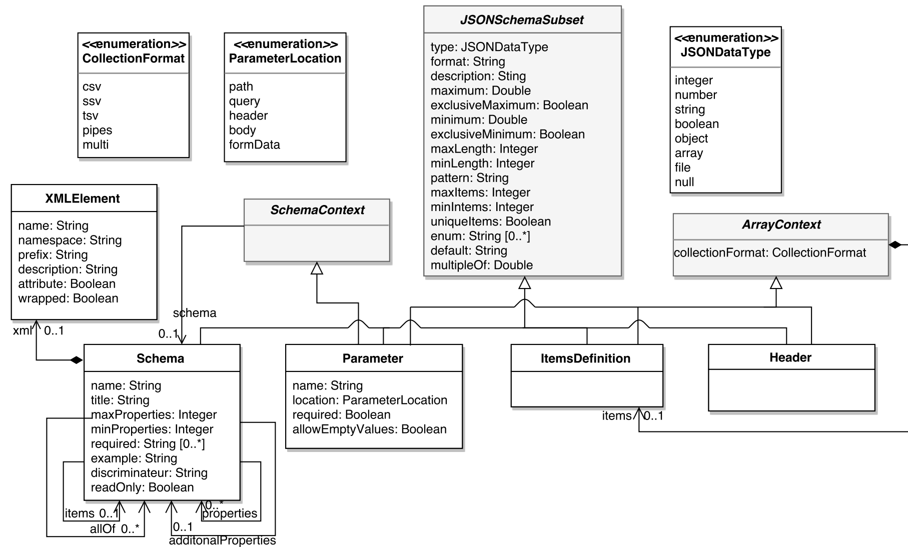


Fig. 4. Structural elements of the OpenAPI metamodel.

ation (`schemes` attribute). An operation includes also the possible responses returned from executing the operation (`responses` reference).

`API`, `Path` and `Operation` elements inherit from `ParameterContext`, which allow them to define parameters at API level (applicable for all API operations), path level (applicable for all the operations under this path) or operation level (applicable only for this operation).

The `Response` element defines the possible responses of an operation and includes the HTTP response code, a description, the list of headers sent with the response, and optionally an example of the response message. `Response` and `Parameters` elements inherit from `SchemaContext` thus allowing them to add the definition of the response structure and the data type (`schema` reference) used for the parameter, respectively. `Parameter` and `Schema` elements will be explained in Section 4.2.

4.2 Structural Elements

Figure 4 describes the structural elements used in a REST Web API, namely: the `Schema` element, which describes the data types; the `Parameter` element, which defines an operation parameter; the `ItemsDefiniton` element, which describes the type of items in an array; and the `Header` element, which describes a header sent as part of a response. These elements use an adapted subset of the JSON Schema Specification defined in the super class `JSONSchemaSubset`[14].

A parameter includes a name, and two flags to specify whether either the parameter is required or empty.

[14] More information about the schema information can be found at `http://json-schema.org/latest/json-schema-validation.html`

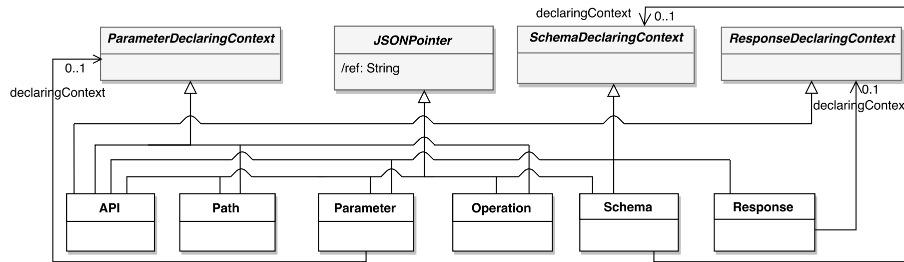


Fig. 5. Serialization/deserialization elements

The location of the parameter is defined by the `location` attribute. The possible locations are: (i) `path`, when it is part of the URL (e.g., `petId` in `/pet/petId`); (ii) `query`, when it is appended to the URL (e.g., `status` in `/pet/findByStatus?status="sold"`); (iii) `header`, for custom headers; (iv) `body`, when it is in the request payload; and (v) `formData`, for specific payloads[15].

`Parameter` and `Header` elements inherit from `ArrayContext` to allow them to specify the collection format and the items definition for attributes of type `array`. Additionally the `Parameter` element inherits from the `SchemaContext` to define the data structure when the attribute `location` is of type body (`Schema` reference).

The `Schema` element defines the data types that can be consumed and produced by operations. It includes a name, a title, and an example. Inheritance and polymorphism is specified by using the `allOf` reference and the `discriminator` attribute, respectively. Furthermore, when the schema is of type `array`, the `items` reference makes possible to specify the values of the array.

4.3 Serialization/Deserialization Support

Figure 5 shows the elements of the metamodel to support serialization and deserialization of OpenAPI models in JSON (or YAML) format. As said before, a parameter can be defined at the API level, path level, or operation level. To specify this, `API`, `Path`, and `Operation` elements inherit from the `ParameterDeclaringContext` element which is referenced in each parameter (`declaringContext` reference). A similar strategy is followed by the `Schema` element (the schema can be declared in the API level, parameter level, response level, or inside a schema) and the `Response` element (a response can declared at the API level or operation level).

All behavioral elements inherit from the `JSONPointer` element which defines a JSON reference for each element. This element includes a derived attribute called `ref` which is dynamically calculated depending on its declaring context. This attribute specifies the path of the element within a JSON document following RCF 6901[16] which can be used to reference a JSON object within the JSON document.

[15] `application/x-www-form-urlencoded` or `multipart/form-data`
[16] `https://tools.ietf.org/html/rfc6901`

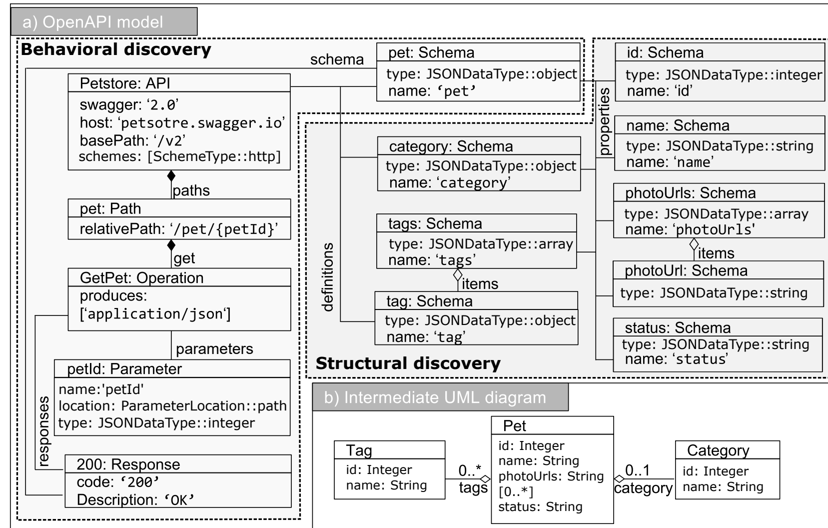


Fig. 6. The discovered OpenAPI model from the Petstore API example.

5 The Discovery Process

The discovery process takes as input a set of API call examples and incrementally generates an OpenAPI model conforming to our OpenAPI metamodel using two types of discoverers: (1) behavioral and (2) structural. The former generates the behavioral elements of the model (e.g., paths, operations) while the latter focuses on the data types elements. In the following we explain the steps followed by these two discoverers.

5.1 Behavioral Discoverer

This discoverer analyzes the different elements of the API example calls (i.e., HTTP method, URL, request body, response status, response body) to discover the behavioral elements of the metamodel.

Table 1 shows the applied steps. *Target elements* column displays the created/updated elements in the OpenAPI model while *Source* column shows the elements of an API call example triggering those changes (see Figures 1a and 1b). The *Action* column describes the applied action at each step and the *Notes* column displays notes for special cases. These steps are applied in order and repeated for each API call example. A new element is created only if such element does not exist already in the OpenAPI model. Otherwise, the element is retrieved and enriched with the new discovered information. Note that the discovery of the schema structure will be assessed by the structural discoverer (see step 6).

Figure 6a shows the generated OpenAPI model for the API call example shown in Figure 1. The discovery process is applied as follows. Step 1 creates an `API` element and set its attributes (i.e., `schemes` to *SchemeType::http*, `host` to *petstore.swagger.io*, and `basePath` to */v2*). Step 2 creates a `Path` element, sets its only attribute `realtivePath` to */pet/{petId}* (the string *'123'* was detected as identifier), and adds it to the `paths`

Table 1. Steps of the behavioral discoverer applied for each REST Web API call example.

Step	Source	Target elements	Action	Notes
1	*<host>*, *<basePath>*, *<protocol>*	`a:API`	-`a.schemes`= *protocol* -`a.host`= *host* -`a.basePath`= *basePath*	If the path contains many sections (e.g., `/one/two/...`) the base path is set to the first section (e.g., `/one`) otherwise it is set to "`/`".
2	*<relativePath>*	`pt:Path`	-Add `pt` to `a.paths` -`pt.relativePath`= *relativePath*.	If *relative path* contains an identifier, it is replaced with a variable in curly braces to use path parameters. A pattern-based approach is used to discover identifiers.[a].
3	*<httpMethod>*, *<RequestBody>*, *<ResponseBody>*	`o:Operation`	-`pt.{httpMethod}`= `o` -If *requestBody* is of type JSON then add `"application/json"` to `o.consumes` otherwise keep `o.consumes` empty. -If *responseBody* is of type JSON then add `"application/json"` to `o.produces` `o.produces` otherwise keep `o.consumes` empty.	*{httpMethod}* is the reference of `pt` which corresponds to *<httpMethod>* (e.g., *get* or *post*).
4	*<query>*, *<relativePath>*, *<requestBody>*	`pr:Parameter`	-Add `pr` to `o.parameters` -Set `pr.type` to the inferred type[b] -Set `pr.location` to: (i) `path` if parameter is in *relativePath* (ii) `query` if parameter is in *query* (iii) `body` if parameter is in *requestBody*	Apply this rule for all the detected parameters. The discovery of the schema of the body parameter is launched in step 6
5	*<ResponseCode>*	`r:Response`	-Add `r` to `o.responses` -`r.code`= *responseCode* -`r.description`= correspondent description of the response.	The discovery of the schema of the response body is launched in step 6.
6	*<RequestBody>*, *<ResponseBody>*	`s:Schema`	-Add `s` to `a.definitions`. -Set the `s.name`= the last meaningful section of the path. -If the schema is in *<RequestBody>*, set `pr.schema` to `s` where `pr` is the body parameter created in step 4. -If the schema is in *<ResponseBody>*, set `r.schema` to `s` where `r` is the response created in step 5. -Launch the structural discoverer.	We apply this rule only if *requestBody* or *responseBody* contains a JSON object.

[a] We apply an algorithm which detects if a string is a UID (e.g., hexadecimal strings, integer).

[b] When a conflict is detected (e.g., a parameter was inferred as integer and then as string), the most generic form is used (e.g., string).

references of the `API` element. Step 3 creates an `Operation` element, sets its `produces` attribute to *application/json*, and adds it to the `get` reference of the previously created `Path` element. Step 4 creates a `Parameter` element, sets its attributes (i.e., `name` to *petId*, `location` to *path*, and `type` to *JSONDataType::integer*), and adds it to the `parameters` reference of the previously created `Operation` element. Step 5 creates a `Response` element, sets its attributes (i.e., `code` to *200* and `description` to *OK*), and adds it to the `response` reference of the previously created `Operation` element. Finally step 6 creates a `Schema` element, sets only its name to `Pet`, and adds it to the `definitions` reference of the `API` element. The rest of the `Schema` element will be completed by the structural discoverer.

Table 2. Transformation rules from UML to Schema

Source	Target: create	Target: update	Attributes initialization
`Class`	`c`: `Schema`	- Add `c` to the `api.definitions`.	- `c.type` = `Object` - `c.name` = The corresponding class name
`Attribute` (1)	`a`: `Schema`	-Add `a` to `c.properties` where `c` is the correspondent schema of the class containing the attribute.	- `a.type` = the `JSONDataType` correspondent to the type of the attribute - `a.name` = the attribute name
`Attribute` (*)	`a`: `Schema`, `i`: `Schema`	-Add `a` to `c.properties` where `c` is the correspondent schema of the class containing the attribute.	- `a.type` = `array` - `a.items`= `i` - `i.type`= the `JSONDataType` correspondent to the type of the attribute - `i.name` = the attribute name
`Association` (1)	-	-Add `tc` to `c.properties` where `c` is the correspondent schema of the source class of the association and `tc` the correspondent schema of the target class of the association.	-
`Association` (*)	`a`: `Schema`	-Add `a` to `sc.properties` where `sc` is the correspondent schema of the source class of the association	- `a.type` = `array` - `a.items`= `tc` where `tc` is the correspondent schema of the target class of the association.

5.2 Structural Discoverer

This discoverer instantiates the part of the OpenAPI model related to data types and schema information. This process is started after the behavioral discovery when the API call includes a JSON object either in the request body or the response body that will be used to enrich the definition of the discovered `Schema` elements.

We devised a two-step process where we first obtain an intermediate UML-based representation from the JSON objects and then we perform a model-to-model transformation to instantiate the actual schema elements of the OpenAPI metamodel. This intermediate step allows us to benefit from JSONDiscoverer [4], which is the tool used to build a UML class diagram, and to use this UML-based representation to bridge easily to other model-based tools if needed. Then, classes, attributes, and associations of the UML class model are transformed to `Schema` elements. Table 2 shows the transformation rules applied to transform UML models to `Schema` elements. *Source* column shows the source elements in a UML model while *Target: create* and *Target: update* columns display the created/updated elements in the OpenAPI model. The *Attribute initialization* column describes the transformation rules.

Note that elements are updated/enriched when they already exist in the OpenAPI model. This particularly happens when different examples represent the same `schema` elements, as JSON schema allows having optional parts in the examples.

Figure 6b shows the UML class model discovered by JSONDiscoverer for the API response shown in Figure 1b. This class model is transformed to actual schema elements applying the discovery process as follows. `Tag`, `Pet`, and `Category` classes are transformed to `schema` elements of type `Object`. Single-valued attributes (e.g., `name`, `id`) are transformed to `Schema` elements where `type` is set to the corresponding primitive type. The `photoUrls` multivalued attribute and `tags` multivalued association are transformed to `Schema` elements of type `array` having as `items` a `Schema` element of type `String` and `Tag` element, respectively. Finally, attributes and associations are added to the `properties` reference of the corresponding `Schema` element.

```
1 { "swagger":"2.0",
2    "info":{  },
3    "host":"petstore.swagger.io","basePath":"/v2",
4    "tags":[ "pet" ],"Schemes":[ "http" ],
5    "paths":{
6          "/pet/{petId}":{
7            "get":{
8             "produces":["application/json"],
9             "parameters":[{"name":"petId","in":"path","type":"integer"}],
10            "responses":{
11               "200":{
12                  "description":"OK",
13                  "schema":{"$ref":"#/definitions/Pet"
14                  }}}}
15    }},
16    "definitions":{
17       "Pet":{
18          "type":"object",
19          "properties":{
20             "id":{"type":"integer"},
21             "category":{"$ref":"#/definitions/Category"},
22             "name":{"type":"string"},
23             "photoUrls":{"type":"array","items":{"type":"string"}},
24             "tags":{"type":"array","items":{"$ref":"#/definitions/Tag"}},
25             "Status":{"type":"string"}},
26           }}}
```

Fig. 7. The generated OpenAPI specification of the Petstore API example.

6 The Generation Process

The generator creates a OpenAPI-compliant JSON file from an OpenAPI model by means of a model-to-text transformation. The root object of the JSON file is the `API` model element, then each model element is transformed to a pair of name/value items where the type for the value is (1) a string for primitive attributes, (2) a JSON array for multivalued element or (3) a JSON object for references. Serialization/deserialization model elements are used to resolve references. As said in Section 4, elements such as `Schema`, `Parameter`, and `Response` can be declared in different locations and reused by other elements. While the `declaringContext` reference is used to define where to declare the object, the `ref` attribute (inherited form `JSONPointer` class) is used to reference this object from another element. By default the discovery process sets the declaring context to the containing class of the element (e.g., parameters in operations).

Figure 7 shows the generated JSON file for the OpenAPI model shown in Figure 6a. Note that the declaring context of the `Pet` schema element is set to *API*, which resulted in listing the `Pet` element in the `definitions` object. Consequently, the attribute `ref` is set to `#/definitions/Pet` and will be used to reference `Pet` from any another element (as in the `response` object).

7 Validation and Limitations

To ensure the quality of the OpenAPIs we generate, we have first enriched the OpenAPI metamodel with a set of well-formedness constraints written using the Object Constraint Language (OCL) [3] (e.g., to guarantee the uniqueness of the parameters in a call). These constraints are checked during the discovery process to validate the generated OpenAPI specification against the constraints published in the last official OpenAPI specification document. Note that this is in itself a useful contribution with regard to other syntax checkers for API documents that offer a limited support in terms of constraint checking.

Additionally, we have validated our approach by manually comparing the results of our generated OpenAPI with the original specification for a number of APIs providing already such information. This has been an iterative process but we would like to highlight the latest test, comprising the following five APIs: (1) Refuge Restrooms[17], a web application that seeks to provide safe restroom access for transgender; (2) OMDb[18], an API to obtain information about movies; (3) Graphhopper[19], a route optimization API to solve vehicle routing problems; (4) Passwordutility[20], an API to validate and generate passwords using open source tools; and finally (5) the Petstore API. Several factors influenced the choice of these APIs to serve for our testing purposes. Beside having an OpenAPI specification, these APIs did not involve fees or invoke services (e.g., SMS APIs), they managed JSON format (to test our structural discoverer) and were concise (to keep limited the number of examples required).

For the chosen APIs, our approach was able to generate on average 80% of the required specification elements and did not generate any incorrect result. Mainly, the missing information was due to the structure of the call examples which cannot cover advanced details such as: (i) the enumerations used for some parameters, (ii) the optionality or not of the parameters, (iii) form parameters, and (iv) the headers used in some operations. Furthermore, the quality of the results depend on the number and the variety of the API call examples used to discover the specification. Our experience so far shows that the number of examples should be higher than the number of operations of an API covering all the parameters. However, more experiments are required to identify the ideal balance between the quality of the result and the number of needed experiments.

Note that even if the result is not complete, it can still be useful. Even for APIs that do provide an OpenAPI as starting point. For instance, for Refuge Restrooms, we were able to discover both the operations and data model of the API even if the latter was not part of the original specification. The complete set of examples and generated APIs are available in our repository.

8 Related Work

Several tools supporting the OpenAPI initiative have recently appeared[21], e.g., able to generate documentation and code (e.g., client SDKs, server skeletons) from OpenAPI-compliant specifications making OpenAPIs a more valuable artefact. Third party companies like Lucybot[22] or ReDoc[23], provide similar capabilities while others as Restlet Studio[24] and stoplight[25] add also the feature of helping developers manually design such APIs with visual tools. Our approach can join this tool ecosystem by inferring the OpenAPIs to be used as input for all these tools out of the box.

---

[17] `http://www.refugerestrooms.org/api/docs/`
[18] `http://www.omdbapi.com/`
[19] `https://graphhopper.com/`
[20] `http://passwordutility.net`
[21] `http://swagger.io/tools/`
[22] `https://lucybot.com/`
[23] `http://rebilly.github.io/ReDoc/`
[24] `https://studio.restlet.com`
[25] `http://stoplight.io/platform/design/`

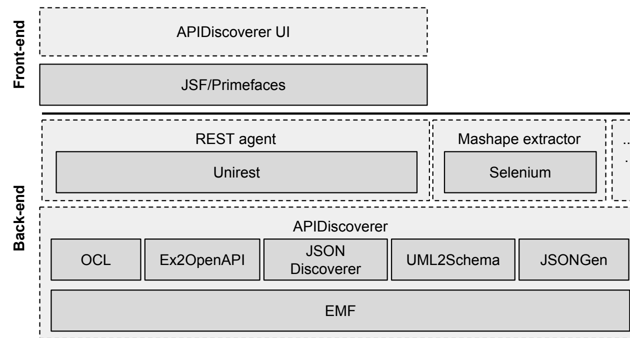


Fig. 8. Tool architecture.

Regarding the discovery process itself, there is a limited number of related efforts and barely any targeting specifically REST or Web APIs in general. Some research efforts (i.e., [9, 17]) focus on the analysis of service interaction logs to discoverer message correlation in business processes. Other works (i.e., [14, 16, 13]) are more proactive and try to suggest possible compositions based on a WSDL (or similar) description of the service. Nevertheless, they all focus on the interaction patterns and do not generate any description of Web APIs specification (or the initial WSDL document for previous approaches) themselves. SpyREST [18] is a closer work to ours. It proposes a Proxy server to analyze HTTP traffic involved in API calls to generate API documentation. Still, the generated documentation is intended to be read by humans and therefore does not adhere to any formal API specification language.

Other research efforts limit themselves to discover the data model underlying an API, specially by analysing the JSON documents it returns. For instance, the works in [7] and [15] analyze JSON documents in order to generate their (implicit) schemas. However, they are specially bounded to NoSQL databases and are not applicable for Web APIs. On the other hand, JSONDiscoverer [4] generates UML class diagrams from the JSON data returned after calling a Web API. We use this tool in our structural discoverer phase.

9 Tool Support

Figure 8 shows the underlying architecture of our discovery tool. Our tool includes a front-end, which allows users to collect and run API call examples (see *APIDiscoverer UI*) to trigger the launch of the core API discoverer process; and a back-end, which all the components to parse the calls and responses, generate the intermediate models, etc. Our tool has been implemented in Java and is available as an Open Source application[26].

More specifically, *APIDiscoverer* is a Java Web application that can be deployed in any Servlet container (e.g., Apache Tomcat). The application relies on JavaServer Faces (JSF), a server-side technology for developing Web applications, and Primefaces[27], a UI framework for JSF applications. Figure 9 shows a screenshot of the APIDiscoverer

[26] `https://github.com/SOM-Research/APIDiscoverer`

[27] `http://www.primefaces.org`

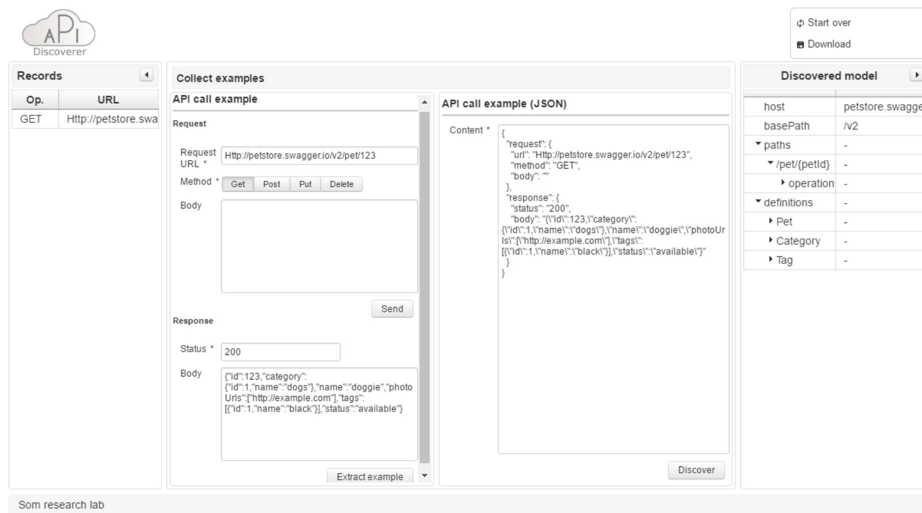


Fig. 9. Screenshot of the discoverer UI.

interface. The center panel of `APIDiscoverer` contains a form to provide API call examples either by sending requests or using our JSON-based representation format. The former requires providing the request and obtaining a response from the API. As result, a JSON-based API call example is shown on the right. The latter only requires providing the JSON-based API call example. API call examples are then used by APIDiscoverer to obtain/enrich the corresponding OpenAPI model. The examples history is shown on the left panel and an intermediate OpenAPI model is shown on the right panel. The OpenAPI model is updated after each example with the new information discovered by the last request. Finally, a button in the top panel allows the user to download the final OpenAPI description file.

The main components of the back-end are (1) a REST agent and (2) the core APIDiscoverer. The REST agent relies on unirest[28], a REST library to send requests to APIs to build and collect API call examples. The APIDiscoverer relies on a plethora of web/modeling technologies, namely, (1) the Eclipse Modeling Framework (EMF)[29] as a modeling framework to implement the OpenAPI metamodel, (2) the Eclipse OCL to validate models and (3) the JSONDiscoverer to discover models from JSON examples. Additionally, we have implemented the required components (1) to discover OpenAPI elements from API call examples (see *Ex2OpenAPI*), (2) to transform UML models to a list of schema elements using model-to-model transformations (see *UML2Schema*), and (3) to generate an OpenAPI description file from an OpenAPI model by using model-to-text transformations (see *JSONGen*).

Beyond these key components, we have also developed *MashapeDiscoverer*, a proof-of-concept to show how the API call examples can be derived from other sources like available examples in the API documentation (in this specific case, from APIs in the

[28] `http://unirest.io`
[29] `http://www.eclipse.org/modeling/emf/`

Mashape marketplace[30], a documentation portal with over 2,000 APIs) by using Selenium[31] to crawl the documentation pages and extract the relevant examples information (i.e, entrypoints, parameters, response examples).

Additionally, we have created HAPI[32], a public REST Web API directory and an open source community-driven project, which stores the discovered Web APIs. Besides allowing users to download the Web API specifications, this directory invites developers to contribute using the well-known pull-request model of GitHub. In order to enrich *HAPI*, we have also created two OpenAPI importers for APIs.guru and APIs.io that use their dedicated Web APIs[33]. This allows easily adding to HAPI APIs already with a predefined specification.

10 Conclusion

We have presented an example-driven approach to generate OpenAPI specifications for REST Web APIs. These specifications are stored in a shared directory where anybody can comment and improve them. We believe our process and repository is a significant step forward towards API reuse, helping developers to find and integrate the APIs they need to provide their software services. The discovery tool is available online as an open source application.

As further work, we are interested in extending the OpenAPI metamodel to add Quality of Service (QoS) and business plan aspects, which play a fundamental role in the API economy, as well as ontology and vocabulary concepts (e.g., FOAF ontology) to describe the APIs not only on a syntactical level but also on a semantic level. We are also interested in discovering security aspects, non-functional properties, and the semantic definitions of the APIs under scrutiny, and supporting non-JSON data (e.g. XML). The discovery process per se could also be improved by extending our approach to support the generation of call examples based on the textual analysis of the API documentation websites, this way speeding up the process of interacting with the API to infer its specification. Finally, we plan to systematically apply our process to a large number of APIs (linked from other directories or repositories) in order to expand HAPI.

Acknowledgment

This work has been supported by the Spanish government (TIN2016-75944-R project).

---

[30] `https://market.mashape.com`
[31] `http://docs.seleniumhq.org/projects/webdriver/`
[32] `https://github.com/SOM-Research/hapi`
[33] `https://apis.guru/api-doc/` and `http://www.apis.io/apiDoc`

References

1. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Morgan & Claypool Publishers (2012)
2. Brambilla, M., Fraternali, P., et al.: The Interaction Flow Modeling Language (IFML). Tech. rep., Object Management Group (OMG) (2014)
3. Cabot, J., Gogolla, M.: Object Constraint Language (OCL): a Definitive Guide. In: Formal methods for model-driven engineering, pp. 58–90 (2012)
4. Cánovas Izquierdo, J.L., Cabot, J.: JSONDiscoverer: Visualizing the Schema Lurking Behind JSON Documents. Knowledge-Based System 103, 52–55 (2016)
5. Frankle, J., Osera, P.M., Walker, D., Zdancewic, S.: Example-Directed Synthesis: A Type-Theoretic Interpretation. In: ACM Symp. on Principles of Programming Languages. pp. 802–815 (2016)
6. Hadley, M.J.: Web Application Description Language (WADL). Tech. rep. (2006)
7. Klettke, M., Störl, U., Scherzinger, S., Regensburg, O.: Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores. In: Conf. on Database Systems for Business, Technology, and Web. pp. 425–444 (2015)
8. López-Fernández, J.J., Cuadrado, J.S., Guerra, E., de Lara, J.: Example-Driven Meta-Model Development. Software & Systems Modeling 14(4), 1323–1347 (2015)
9. Motahari-Nezhad, H.R., Saint-Paul, R., Casati, F., Benatallah, B.: Event Correlation for Process Discovery from Web Service Interaction Logs. Inter. J. on Very Large Data Bases 20(3), 417–444 (2011)
10. Nierstrasz, O., Kobel, M., Girba, T., Lanza, M.: Example-Driven Reconstruction of Software Models. In: Euro. Conf. on Software Maintenance and Reengineering. pp. 275–286 (2007)
11. Pautasso, C., Zimmermann, O., Leymann, F.: RESTful Web Services vs. "Big"' Web Services. In: Inter. Conf. on World Wide Web. pp. 805–814 (2008)
12. Pezoa, F., Reutter, J.L., Suarez, F., Ugarte, M., Vrgoč, D.: Foundations of JSON Schema. In: Inter. Conf. on World Wide Web. pp. 263–273 (2016)
13. Quarteroni, S., Brambilla, M., Ceri, S.: A Bottom-up, Knowledge-Aware Approach to Integrating and Querying Web Data Services. ACM Transactions on the Web 7(4), 19–33 (2013)
14. Rodriguez Mier, P., Pedrinaci, C., Lama, M., Mucientes, M.: An Integrated Semantic Web Service Discovery and Composition Framework. IEEE Transactions on Services Computing 9(4), 537–550 (2015)
15. Ruiz, D.S., Morales, S.F., Molina, J.G.: Inferring Versioned Schemas from NoSQL Databases and its Applications. In: Int. Conf. on Conceptual Modeling. pp. 467–480 (2015)
16. Schmidt, C., Parashar, M.: A Peer-to-Peer Approach to Web Service Discovery. In: Inter. Conf. on World Wide Web. pp. 211–229 (2004)
17. Serrour, B., Gasparotto, D.P., Kheddouci, H., Benatallah, B.: Message Correlation and Business Protocol Discovery in Service Interaction Logs. In: Int. Conf. on Advanced Information Systems Engineering. pp. 405–419 (2008)
18. Sohan, S., Anslow, C., Maurer, F.: SpyREST: Automated RESTful API Documentation Using an HTTP Proxy Server (N). In: Int. Conf. on Automated Software Engineering. pp. 271–276 (2015)