



PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA

RAD-NLQ: A REST API RESOURCE DISCOVERY FRAMEWORK SUPPORTING NATURAL LANGUAGE QUERIES

NIKOLAS GONZALO BRAVO RAKELA

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Advisor:

ROSA ALARCÓN

Santiago de Chile, January 2017

© MMXVII, NIKOLAS BRAVO



PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA

RAD-NLQ: A REST API RESOURCE DISCOVERY FRAMEWORK SUPPORTING NATURAL LANGUAGE QUERIES

NIKOLAS GONZALO BRAVO RAKELA

Members of the Committee:

ROSA ALARCÓN

JAIME NAVÓN

CÉSAR AGUILAR

SERGIO GUTIÉRREZ

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Santiago de Chile, January 2017

© MMXVII, NIKOLAS BRAVO

To humanity

ACKNOWLEDGEMENTS

I would like to thank my parents for supporting me during my studies. I would also like to thank Javiera Pérez for her continued and always present support and advise. Additionally, special thanks are in order to those who directly contributed to the underlying project involving this thesis, especially my friend Rodrigo Saffie.

I also like to thank my friends. I would like to thank Tomás Vukasović, Felipe Rivera, and Felipe Kopplin for their immense and continued support and companionship during this period. I would also like to thank all my past classmates and future colleagues for their companionship and counsel throughout my thesis: Augusto Sandoval and Guillermo Valenzuela with whom we have shared much work together; Hans Findel and Alfredo Cobo with whom we shared an office and always had the time to share their helpful advise; and Nicolás Risso, Santiago Larraín, and José Ignacio Navarro for their continued companionship in these past months leading to my work's end. Finally I would like to acknowledge my e-pen-pal Sebastián Hoch for our always enjoyable mind-sparring.

Finally, but not least, I would like to thank my Graduate Committee, professors Rosa Alarcón, Jaime Navón, César Aguilar, and Sergio Gutiérrez for their time and insight reviewing this work.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	iv
LIST OF FIGURES	vii
LIST OF TABLES	ix
ABSTRACT	x
RESUMEN	xi
1. INTRODUCTION	1
2. RELATED WORK	3
2.1. REST Web API description	3
2.2. Natural Language for Web service and REST API discovery	5
3. RAD: REST API DESCRIPTION	9
3.1. RAD Concept Vocabulary	10
3.2. RAD as a JSON document	12
3.3. RAD as a graph	17
4. RAD-NLQ: NATURAL LANGUAGE REST RESOURCE DISCOVERY	19
4.1. RAD-based Service Discovery	19
4.2. Supporting Natural Language Queries: RAD-NLQ	20
5. Implementation	26
5.1. RAD-QL: an API to query the RAD graph	26
5.2. System overview	26
6. Evaluation	30
6.1. Vocabulary and API Dataset	30
6.2. Input Query Dataset	31
6.3. Evaluation Criteria	33

6.4. Results	35
7. CONCLUSIONS	38
REFERENCES	40
APPENDIX	43
A. Web Application Screenshots	44

LIST OF FIGURES

3.1	RAD metamodel	9
3.2	Associated Schema.org based vocabulary: document metadata, vocabulary, and prefixes	11
3.3	Associated Schema.org based vocabulary: relationships	12
3.4	JSON implementation schema of RAD	13
3.5	Spotify Web API described as RAD-JSON: API, description metadata, and resources	14
3.6	Spotify Web API described as RAD-JSON: methods	15
3.7	Spotify Web API with RAD in JSON: parameters	16
3.8	Spotify Web API with RAD in JSON: responses	17
3.9	RAD graph model	18
5.1	RAD-NLQ System UML Component Diagram	27
5.2	Screenshot of the RAD-NLQ interface in the Web application: query input . .	28
5.3	Screenshot of the RAD-NLQ interface in the Web application: query graph response	29
6.1	Result's score histogram for RAD-NLQ and the Google Web Search Engine .	37
A.1	RAD-NLQ: User's natural language phrase input	44
A.2	RAD-NLQ: Concept extraction from user's input	45
A.3	RAD-NLQ: RAD-QL query to discover resource-method pairs with the extracted concepts	45

A.4	RAD-NLQ: Ranked <code>method</code> node IDs linked to the extracted concepts . . .	46
A.5	RAD-NLQ: Obtaining RAD-QL query suggestions for the selected <code>method</code> node ID	46
A.6	RAD-NLQ: RAD-QL queries suggested for the given <code>method</code> node ID . . .	47
A.7	RAD-NLQ: Interactive graph detailing the workflow for the given <code>method</code> node	48

LIST OF TABLES

4.1	Typical grammatical structure of queries used by Web searchers with a distribution based on a sample of 222 hand-labeled queries (Excerpt)	23
6.1	Nodes and edges in the graph database	31
6.2	Vocabulary subgraph composition	31
6.3	Activity layer nodes	32
6.4	Use of distinct semantic layer concepts	32
6.5	Composition of our input dataset of 128 queries. RC: Resource Concept, A: Action, PC: Parameter Concept, E: Entity	33
6.6	Evaluation results. RC: Resource Concept, A: Action, PC: Parameter Concept, E: Entity	36

ABSTRACT

A great amount of functionality available on the Web is nowadays provided through Web APIs. Some of them follow the REST design guidelines, characterized by a consistent use of the HTTP methods and the identification of resources with URIs that do not include the media type, among others. REST design empowers APIs and allows them to achieve massive scalability and evolvability. This capability coupled with the introduction of a standardized semantic API description would facilitate machine-clients to discover and use REST Web API dynamically, creating customized ecosystems tailored to the user's needs. In this thesis we present RAD-NLQ, based on the the RAD REST API description, which allows us to implement API discovery through natural language queries. We implemented and tested our approach comparing it with Google Web search engine with promising results.

Keywords: Web API, REST, Service discovery, Natural language.

RESUMEN

Gran parte de las funcionalidades disponibles en la Web es accedida por medio de APIs Web. Algunas de ellas siguen las directrices de diseño de REST que promueven un consistente uso de los métodos HTTP y la identificación de recursos con URIs que no incluyen el *media type*, entre otros. REST empodera a las APIs, y permite que logren gran escalabilidad y evolucionen fácilmente. Ésto, junto con la introducción de una descripción semántica de APIs Web estándar, facilitaría el descubrimiento y consumo de APIs Web REST de forma dinámica, permitiendo la creación dinámica de ecosistemas personalizados a las necesidades del usuario. En esta tesis presentamos RAD-NLQ, un *framework* basado en el modelo de descripción RAD que permite el descubrimiento de pares recurso-método que satisfagan consultas de usuarios expresadas en lenguaje natural. Nuestra implementación de RAD-NLQ fue probada contra el motor de búsqueda de Google arrojando resultados prometedores.

Palabras Claves: API Web, REST, Descubrimiento de Servicios, Lenguaje Natural.

1. INTRODUCTION

The Web was initially conceived as a pull-based content delivery platform. In the past decade we have witnessed a dramatic evolution of its capabilities from its ability to support dynamic content, to customization, rich interfaces, and its support of B2B applications. In the latter case, the evolution of complex traditional services with limited scalability and proprietary platforms have given place to Web APIs that fully comply Web standards and are massively scalable allowing the creation of new business ecosystems and transforming the Web into a marketplace of applications.

The main characteristic that differentiates Web APIs is its underlying architectural style. In our research we focus on the REST (Representational State Transfer) (Fielding, 2000) architectural style. It provides massive scalability, independent evolvability and extensibility, among other benefits to the Web. A REST Web API is a collection of identified resources, which are manipulated through its representations (a snapshot of the resource's state in a moment in time) via a set of self-contained methods, such as those in the Hypertext Transfer Protocol (HTTP) (Fielding et al., 1999). Hypermedia controls (e.g. links) provided in the representations allow clients to discover related resources. REST Web APIs have experienced notorious growth in the developer community: in *ProgrammableWeb*¹, one of the most important Web APIs repositories, most developers declare their APIs as REST-based; *Google Trends*² also shows an overwhelming and increasing interest in REST Web APIs.

Nowadays, an additional revolution is pervading various technological fields by enriching technology with cognitive capabilities. This evolution have been studied in the Web community through efforts such as Semantic Web, though it was not considered in the design of REST Web APIs. One of the advantages of providing semantic support for REST Web APIs is the facilitation of APIs discovery and hence, its composition

¹Programmable Web <http://www.programmableweb.com/>

²Google Trends <https://www.google.com/trends/explore?q=rest%20api,soap%20api>

giving rise to the dynamic creation of new ecosystem customize to the user needs. A fundamental limitation for such case is the lack of a standardized machine-readable API description. REST Web APIs expose their resources and underlying semantics in natural language documents destined to human consumption. Machine clients cannot understand such semantics, and therefore cannot determine the intended business goal achieved by executing a method on a resource. As a direct consequence, the reverse problem exists: resource discovery. A *machine client* cannot determine which REST APIs, resources, or methods should be executed in order to achieve a business goal. Therefore, the discovery of REST web APIs is most commonly done through manual search on large API repositories (e.g. *ProgrammableWeb*) through keywords, tags, or category-based searches; or simply through a Web search engine (e.g. Google³, Bing⁴, or Yahoo!⁵) followed by manual exploration of the returned links.

In a previous publication we have presented the REST API Description (RAD)(Alarcón, Saffie, Bravo, & Cabello, 2015) and showing its capabilities to organize REST API's methods by exploiting a semantic layer associated with the API description. In (Saffie, 2016) we exploited these characteristics to implement an automatic service composition approach. In this thesis we present RAD-NLQ, a framework based on RAD for the discovery of resource-method pair that achieve a specific business goal through natural language phrase queries. In order to validate our approach we implemented a prototype including 6 popular REST Web APIs, which outperformed the Google Web search engine in our tests.

This thesis is organized as follows: Chapter 2 presents related work on REST Web API descriptions and REST Web API discovery. Chapter 3 presents RAD, its underlying metamodel, its implementation as a JSON document, and its representation as a graph. Chapter 4 presents RAD-NLQ, our RAD-based approach for REST web API discovery through natural language, while Chapter 5 presents our prototype's implementation, and Chapter 6 our evaluation. Finally, Chapter 7 presents our conclusions.

³Google <http://google.com/>

⁴Bing <http://bing.com/>

⁵Yahoo! <http://yahoo.com/>

2. RELATED WORK

2.1. REST Web API description

A REST Web Service is a collection of resources, each resource having a unique identifier (e.g. a URI), which can be manipulated by a well defined set of methods (e.g. HTTP methods) (Fielding, 2000) and representations. A *representation* contains information of the resource's state in a particular format (e.g. HTML, JSON, XML, etc.) at a particular time which can be retrieved (e.g. through an HTTP GET operation) or used to modify the resource state (e.g. an HTTP POST operation). Additionally, REST requires that its architectural components (e.g. clients, servers, caches, etc.) interact between each other through self-descriptive messages (e.g. the correct use of HTTP methods). Finally, a REST system must be hypermedia-centered, meaning that a resource's representation must contain the necessary controls and links that allow the client to identify the available actions at any point in the client-server interaction. These four characteristics constitute REST's *Uniform Interface* constraint, which characterizes the Web and as such the design of REST Web APIs. REST also facilitates service evolvability by leveraging Web standards (e.g. data formats, network protocols, etc.), and service scalability by exploiting REST architectural constraints (e.g. layers, caches, etc.).

Multiple proposals exist in order to describe REST Web APIs. The Web Application Description Language (WADL) (Hadley, 2009) is the REST equivalent of the Web Services Description Language (WSDL) (Chinnici, Moreau, Ryman, & Weerawarana, 2007). WADL describes a REST API in terms of resources, media types, schemas of the expected request and response, and representations containing parameters with links to other resources. However, it does not offer support for link discovery, ignoring the dynamic nature of REST itself. As a result, it gravitates to being operation-centric, and introduces additional complexity without yielding any clear benefits for either human or machine-clients. These descriptions are maintained independently from the service itself, also arising maintainability issues (John & Rajasree, 2013).

Other approaches include the Hypertext Application Language (HAL)¹ (Kelly, 2016), a lightweight description language, implemented as a JSON document, focusing on hypermedia in order to make the API explorable, but limited only to the HTTP GET method. Google also presents an interesting proposal, the API Discovery Service². It offers an API which serves machine-readable discovery documents for its own set of supported APIs, including information regarding resources, their JSON Schema³, methods available for each resource, and their parameters.

Semantic descriptions have been also proposed for REST services. RESTdesc⁴ (Verborgh et al., 2011) (Verborgh et al., 2013) represents REST API functionality in RDF, including a request's qualified pre and postconditions. Though it is flexible, compact, and able to handle complexity, it requires previous knowledge of the resources' URIs in order to execute more advanced queries. SA-REST⁵ (Lathem, Gomadam, & Sheth, 2007), and hRESTS (Kopecky, Gomadam, & Vitvar, 2008), are simpler approaches; both propose the creation of a new resource describing API resources' URIs, methods, input, and output parameters written either as RDFa property-value pairs (Adida, Birbeck, McCarron, & Pemberton, 2008) (SA-REST), or Microformat annotations (Khare & Çelik, 2006) (hRESTS). Both approaches, SA-REST and hRESTS, support links but do not support dynamic resource discovery by following such links. ReLL (Alarcón & Wilde, 2010) is fully compliant on REST's principles and has shown its hypermedia capability when fully crawling a REST service's resources. However, ReLL only supports the HTTP GET method assuming only one semantic action: reading the resource's state. Finally, Hydra (Lanthaler & Gütl, 2013) is based on JSON-LD (Sporny, Longley, Kellogg, Lanthaler, & Lindström, 2014), which adds lightweight semantics to the service's description. Hydra models resources, operations, and hyperlinks as link templates, but its underlying RDF model adds significant complexity to the proposal.

¹HAL http://stateless.co/hal_specification.html

²Google API Discovery Service <https://developers.google.com/discovery/>

³JSON Schema <http://json-schema.org/>

⁴RESTdesc <http://restdesc.org/>

⁵SA-REST <https://www.w3.org/Submission/SA-REST/>

Industry approaches have been rapidly and steadily growing, especially among Web developers, as the need to standardize descriptions among REST Web API providers increases. Swagger⁶, among the most popular proposals in this category, has been adopted into the Open API Initiative⁷ specification. A Swagger description is a JSON or YAML document which describes an API's resources, methods, parameters, and responses and their schemas. The RESTful API Modeling Language (RAML)⁸ is a similar proposal, where descriptions are implemented as YAML documents, providing additional support to rich data type definitions as well as URI parameters. RAML is more expressive than Swagger, but as a consequence is more complex and less intuitive. Lastly, API Blueprint⁹ uses its own Markdown-based format, and supports resources, methods, parameters, data types, and responses superficially as HTTP codes with an associated example. Though it is easy to understand, it is not intuitive to write, and can not be considered flexible or expressive. The company backing API Blueprint, Apiary¹⁰, has recently begun supporting Swagger in their own services, suggesting the future suspension of support of the former. All the industry-driven approaches mentioned before have a common limitation: the lack of any type of associated semantics, limiting any kind of automated resource discovery.

2.2. Natural Language for Web service and REST API discovery

Various approaches have been proposed for the discovery of functionality available in the Web using natural language techniques, mainly for traditional SOAP/WSDL-based Web Services, in stark contrast with the lack of approaches for REST Web APIs. A SOAP/WSDL Web service exposes functionality on the Web that is described in a WSDL document (Box et al., 2000). It contains a set of endpoints (URLs), user defined methods (e.g. BuyApples), input and output parameters (e.g. Price, Quantity) as well as certain rules (e.g. security related). REST Web APIs differ in that methods are limited to the

⁶Swagger <http://swagger.io/>

⁷Open API Initiative <https://www.openapis.org/>

⁸RAML <http://raml.org/>

⁹API Blueprint <https://apiblueprint.org/>

¹⁰Apiary <https://apiary.io/>

network protocol (e.g. HTTP operations) so that not out-of-band information due to ambiguous definition -which is often the case- is required.

In order to address this ambiguity most proposals are based on Semantic Web techniques, where functional aspects of service elements (e.g. operations, input and output parameters) are associated to concepts that are part of a semantic network. These approaches tend to support schema-agnostic natural language keyword-based queries as their input. For instance, (Lakshmi & Dhas, 2013) proposes an improved Semantic Web Service Discovery method by combining functional and textual similarity matching, by means of matching a keyword-based user query to semantic OWL-S (Martin et al., 2004) annotations. (Sangersa, Frasinara, Hogenbooma, & Chepeginb, 2013) also matches keyword-based user queries to semantic annotations in its WSMO Web Service description. Similarly, (Gunasri & Kanagaraj, 2014) extracts annotated keywords in the Web Service's OWL-S description, and groups them into clusters which are later matched, using similarity algorithms, against a natural language keyword-based user input.

For the case of REST API designers, the main focus has been on defining resources rather than user-defined operations. This capability combined with stateless interaction and caches (due to the proper use of HTTP operations) provide APIs with massive scalability. For REST discovery, efforts mainly come from the industry and focus on the discovery of specific resources called *root* resource (API home page, API directory, API description, API documentation, etc.) which represent a catalog of resources for a specific API. Most proposals come in the form of large Web API repositories or directories: Mulesoft's *ProgrammableWeb*¹¹, Mashape's *PublicAPIs*¹², *APIs.io*¹³, and *APIHound*¹⁴.

¹¹Programmable Web <http://www.programmableweb.com/>

¹²Public APIs <https://www.publicapis.com/>

¹³APIs.io <http://apis.io/>

¹⁴APIHound <http://apihound.com/>

ProgrammableWeb has the largest hand-curated API directory which can be queried by category, the API's name, or through text to be matched against each API's text description. This has the disadvantages of APIs having to be curated by a human, the keyword-style search not supporting natural language phrases, and the fact that search results are returned at the root resource and not the specific resources that are semantically related to the query. Since the root resource is a catalog of the API's resources, users must manually navigate the ad-hoc documentation in order to find what they need.

Similarly, *PublicAPIs* is also hand-curated, and drops the category hierarchy in favor of more flexible tags, but still suffers from most of *ProgrammableWeb*'s issues. *APIs.io* indexes APIs through an *Apis.json*¹⁵ document, a machine readable document that API providers can use to describe their API resources through a name, a human-readable description, a set of tags, and external links which could include a *Swagger* description document of the API. *APIs.io* allows users to search their directory through the API's name and tags, and the search response is also limited to the root resource.

APIHound constantly crawls the Web for new APIs, which they index and assign relevant categories and keywords to. The system is designed to be queried through the API's name or keywords, and even though querying using natural language phrases is not explicitly unsupported, they are still executed as keyword-based searches, and as such, results yielded from these queries are often inconsistent and unreliable. *APIHound*'s contribution resides in indexing APIs by directly crawling the Web without direct intervention from the API's developers, but as stated in their blog¹⁶ on January 5 2015, sometimes non-API related material is mistakenly also indexed. Results are also returned at the root resource, this being the main weakness of all large Web API directories, as the user must manually search the response (most commonly a documentation Web page) in order to find the resource, HTTP method, and parameters they need to achieve their intended business goal.

¹⁵Apis.json <http://apisjson.org/>

¹⁶APIHound's Blog <http://apihound.com/blog.jsp>

Another way to discover APIs' resources is through generic Web search engines such as the Google, Bing, or Yahoo! search engines, which are typically designed to support short (Spink, Jansen, Wolfram, & Saracevic, 2002) verbal and non-verbal phrases (Barr, Jones, & Regelson, 2008), allowing much needed flexibility on the end users' queries. However, unlike previous approaches, the limitation of generic search engines resides in its search space, causing results to often be cluttered with irrelevant and non-API related material. Even though if the search space is restricted to the Web API domain, the quality of the result is directly related to the quality and granularity of the API's description (i.e. the root resource content and its links). For example, for an API where each resource's description has its own URL (and is indexed by it), the search engine may return the exact URL with the documentation the user is looking for; whereas if only the API's root resource is indexed, the accuracy of the result is lost, and since such URL is returned to the user, they must keep searching for what they need in the API's documentation resource. Thus, Web search engines response quality is highly dependent on the quality of the documentation's webpage, and as the latter is inconsistent across the Web, so are the search engine's results.

3. RAD

As previously mentioned, REST API descriptions are key to facilitate API discovery. In a previous work (Alarcón et al., 2015), we introduced an approach called RAD to describe REST Web APIs and allow the automatic creation of workflows (Saffie, 2016). In this chapter, we briefly summarize RAD since is the basis of our approach. RAD is presented as a metamodel, a graph model and an implementation. The RAD metamodel (Figure 3.1) separates REST web API elements into a *semantic* and an *activity* layer.

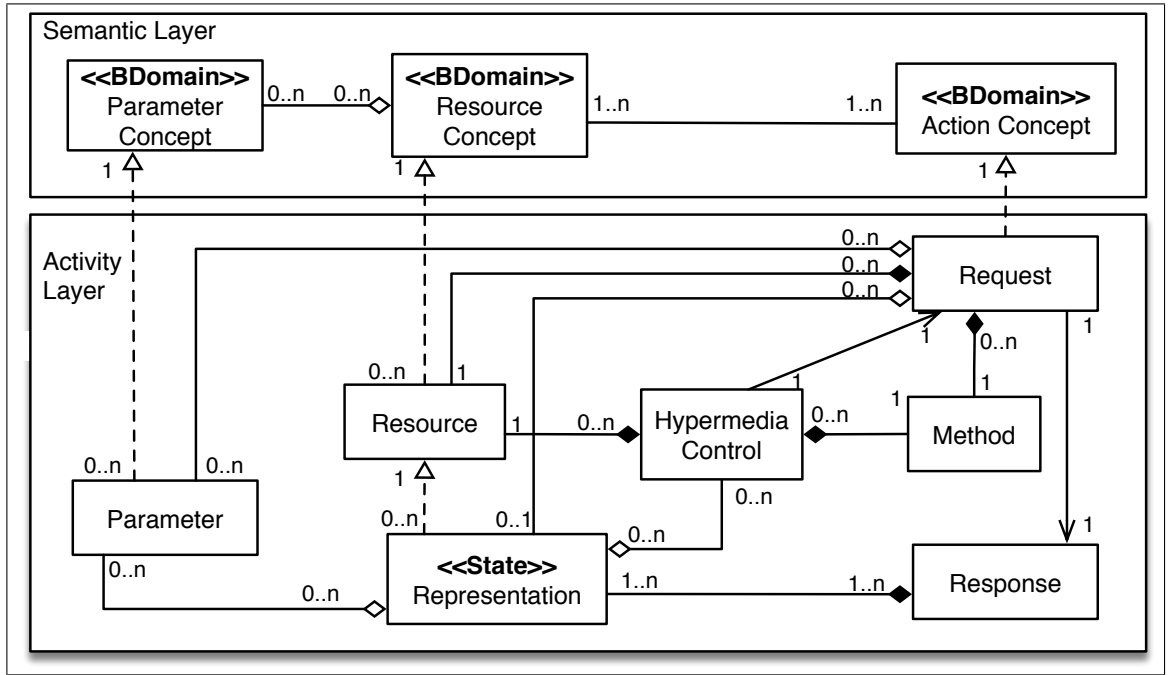


Figure 3.1. RAD metamodel

In the semantic layer, *Resource*, *Parameter* and *Action* are abstract concepts in the *business domain*. They convey the semantics of activity layer elements, but are not bound to its implementation. The activity layer represents the REST Web API itself. *Resource* elements are identified by their URI, and are associated with at least one *Method* (e.g. GET, POST, DELETE, etc. under the HTTP protocol). *Requests* performed over *Resources* may require input *Parameters* (e.g. in the header, body,

or the URI). Upon the execution of the Request, a Response could be received, containing the Resource's state in the form of a Representation, which itself could include output Parameters and a set of Hypermedia Controls (e.g. a hyperlink). Hypermedia Controls refer to a Resource-Method pair, which could potentially be executed through another Request.

3.1. RAD Concept Vocabulary

As discussed in the previous section, resources, parameters and methods are all individually referenced to single unambiguous concepts in the business domain. These concepts, as well as their relationships, are declared in a separate vocabulary document. This vocabulary corresponds to the semantic layer in the RAD metamodel, and could consist of a simple vocabulary or even a full-fledged ontology.

We based our vocabulary on Schema.org¹, a specification backed by companies such as Google, Microsoft and Yahoo to enrich search result snippets through annotations in HTML documents. The Schema.org specification comprehends entities with their own URI, such as items, objects and actions. In this work we chose to extend the current specification by adding properties to a concept through the URI pattern `Concept/newProperty` (which has since been deprecated since May 2015), in which the new property is nested in the original concept's URI. It is important to mention that Resource Concepts are represented in upper camelcase, while its Parameter Concepts are notated in lower camel case. We also saw the need for more specific concepts than those already present, so we added new ones to the vocabulary following the previously mentioned URI pattern, and creating relationships to link them to existing concepts.

Our vocabulary file is implemented as a JSON document. Figure 3.2 and Figure 3.3 portray snippets of such document. The required keys in this document are `name`, `version`, `baseUri`, `prefixes`, and `relationships`. Prefixes are prepended with

¹Schema.org <http://schema.org/>

```

{
  "baseUri": "http://schema.org",
  "name": "RAD-Schema.org",
  "codename": "rad-schema-1.1",
  "version": "1.1",
  "created_at": "3/9/2015",
  "updated_at": "18/10/2016",
  "description": "Extension and adaptation of Schema.org's dictionary for RAD.",
  "prefixes": {
    "actions": {
      "@AchieveAction": "/AchieveAction",
      "@ActivateAction": "/ActivateAction",
      "@AddAction": "/AddAction",
      "@AssessAction": "/AssessAction",
      ...
    },
    "resources": {
      "@Place": {
        "parameters": {
          "@placeAddress": "/address",
          "@placeIdentifier": "/identifier",
          "@placeDistance": "/distance",
          ...
        },
        "reference": "/Place"
      },
    },
  },
  "relationships": {
    ...
  }
}

```

Figure 3.2. Associated Schema.org based vocabulary: document metadata, vocabulary, and prefixes

the '@' character, and are abbreviations of the vocabulary's concepts, and as such are associated with a URI formed by concatenating the vocabulary's `baseUri` and the `reference` value for each `Resource Concept` or `Action Concept`, as well as the `Parameter Concept`'s name in the case of one. RAD descriptions should reference the vocabulary's concepts through these prefixes in order to increase the document's maintainability. Finally, relationships between `Resource Concepts`, and between `Parameter Concepts` are specified under `relationships` key (Figure 3.3). Such relationships

```

    "relationships": {
      "resources": {
        "@Place": {
          "IS A": [
            "@Thing"
          ]
        },
        ...
      },
      "parameters": {
        "@musicEventSongkickIdentifier": {
          "IS A": [
            "@musicEventIdentifier"
          ]
        },
        ...
      }
    }
  }

```

Figure 3.3. Associated Schema.org based vocabulary: `relationships`

are normally those of a specific concept to its generalization, or between a collection of concepts and its individual concepts. Vocabulary entities, through their relations to more general and abstract concepts, form a tree, with `http://schema.org/Thing` at its root.

3.2. RAD as a JSON document

This section presents an overview of the RAD metamodel implemented as a JSON document. This document's purpose is twofold: it must not only serve as documentation, but it also must be machine readable. In Figure 3.4, fields which only serve as human-targeted documentation are presented in *italics* (e.g. *name* for human-friendly names, *description* for human-friendly descriptions, *additional_doc* for links to further documentation, and *example* with example values), and their presence is optional. Additionally, semantic references from `Resources`, `Methods` and `Parameters` to their

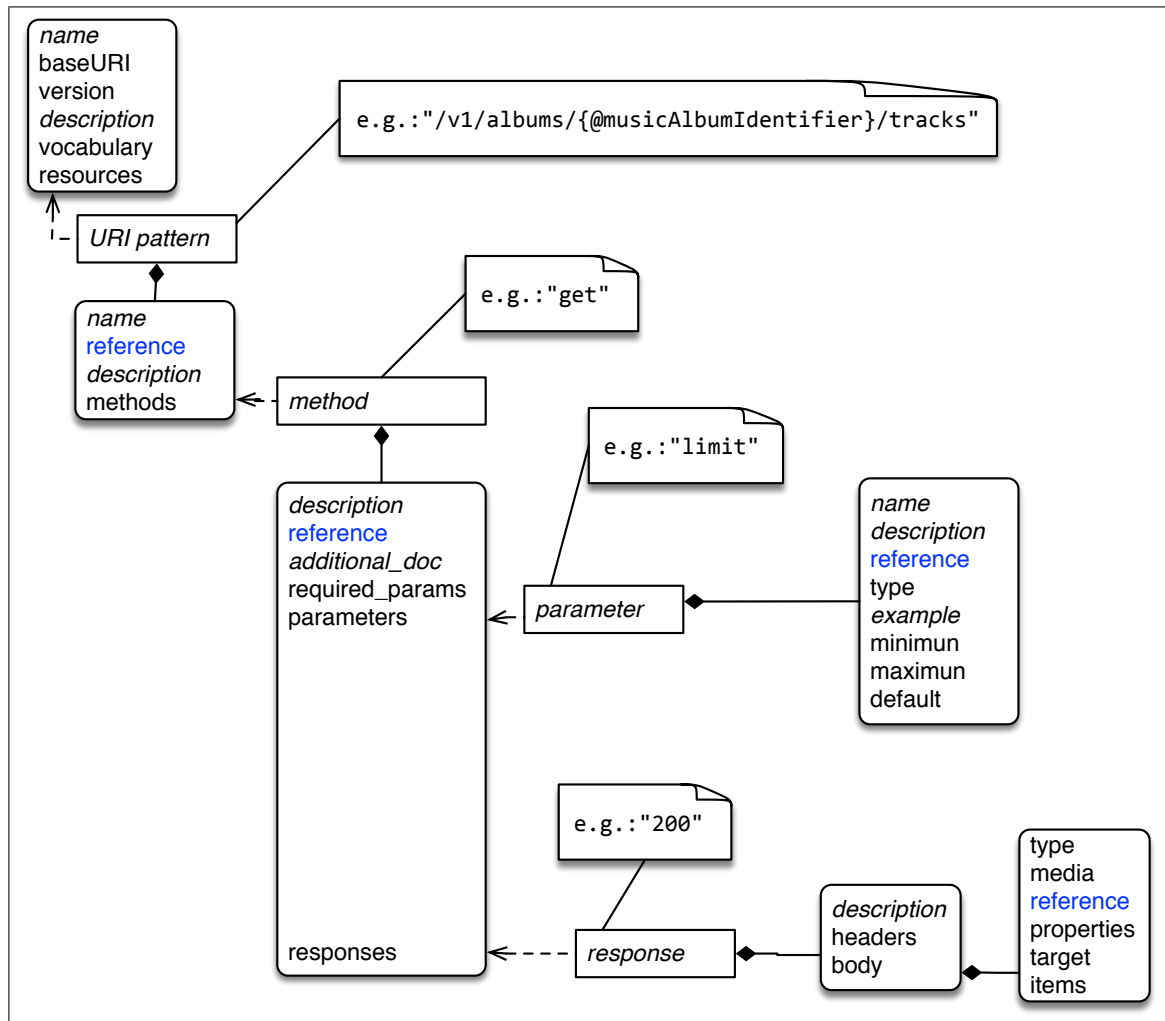


Figure 3.4. JSON implementation schema of RAD

corresponding concepts are done through the `reference` field, seen in blue in the figure. Further explanation of each portion of the JSON document will be accompanied by a snippet of an example JSON description.

Figure 3.5 presents a snippet of the document at its highest level, including general API information, as well as metadata regarding the description itself. Required keys at this level are `baseURI`, `version`, `vocabulary` and `resources`. `baseURI` refers to the invariable and common root portion of the URI preceding the resource's unique path (Webber, Parastatidis, & Robinson, 2010), `version` allows for API versioning,


```

{
  "name": "Spotify",
  "baseURI": "https://api.spotify.com",
  "version": "v1",
  "description": "Our Web API endpoints give external applications access to Spotify catalog and user data.",
  "vocabulary": "rad-schema-1.1",
  "resources": {
    "/v1/users/{@userSpotifyIdentifier}/playlists": {
      "name": "User's Playlists",
      "reference": "@UserMusicPlaylistCollection",
      "description": "Spotify user's playlists.",
      "methods": {
        ...
      }
    },
    "/v1/users/{@userSpotifyIdentifier}/playlists/{@musicPlaylistSpotifyIdentifier}": {
      ...
    },
    ...
  }
}

```

Figure 3.5. Spotify Web API described as RAD-JSON: API, description metadata, and resources

`vocabulary` specifies the unique name of the semantic vocabulary to be used across this whole document, and the `resources` key lists the paths to all the API's resources.

Resources listed in `resources` can have their URI formed by appending the key upon which they are listed to the API's `baseURI`. Semantic references to parameters in the resource's URI template itself can be annotated directly in the path by directly including the relevant `Parameter Concept`'s prefix (see the `/v1/users/{@userSpotifyIdentifier}/playlists` resource in the example). Moving into a resource itself, two keys are mandatory: `reference` links the `Resource` in the activity layer to its corresponding `Resource Concept` in the semantic layer, while `methods` lists all available methods (e.g. GET, POST, DELETE, etc. for the HTTP protocol) available to be executed upon the resource.

Figure 3.6 explores the contents of a resource's `methods` key. A method's identifier is used as the key (see HTTP "get" and "post" in the example). An individual method's required keys are `reference`, `required_params`, `parameters`, and `responses`. Once again the `reference` links the `Method` in the activity layer to its corresponding `Action Concept` in the semantic layer. The `required_params` key determines all

```

"methods": {
  "get": {
    ...
  },
  "post": {
    "description": "Create a playlist for a Spotify user. (The playlist will be empty until you add tracks.)",
    "reference": "@CreateAction",
    "additional_doc": "https://developer.spotify.com/web-api/create-playlist/",
    "required_params": "!Authorization AND !Content-Type AND #name",
    "parameters": {
      ...
    },
    "responses": {
      ...
    }
  }
}
}

```

Figure 3.6. Spotify Web API described as RAD-JSON: `methods`

possible parameter combinations that could be used to execute the method through the evaluation of a logical expression (supporting AND, OR, XOR, and the use of parenthesis). Finally `parameters` and `responses` keys list all the method's parameters and responses respectively.

Figure 3.7 explores the contents of a method's `parameters` key, where all available parameters for that method are listed. Each parameter is listed by the name it must have in the request itself, and may be prepended by a symbol depending on their location in it: `'!'` for the header, and `'#'` for the body. If the parameter is present as a query parameter no symbol is prepended, while if the parameter is part of the resource's URI template it is listed in the resource's key and not in this section. The required keys for each parameter are: `reference` which references the activity layer's `Parameter` to its corresponding semantic layer `Parameter` concept, and `type` which represents the parameter's data type (e.g. `"string"`, `"integer"`, `"boolean"`, `"array"`, etc.). A parameter's value could also have restrictions: `enum` indicates the list of values the parameter can take, `default` states the value the parameter will take if it is not included in the request, and `maximum` and `minimum` respectively limit the maximum and minimum values a number-based data type parameter is allowed to have.

```

"parameters": {
  "!Authorization": {
    "name": "Authorization Access Token",
    "description": "A valid access token from the Spotify Accounts service",
    "reference": "@webApplicationSpotifyApiKey",
    "type": "string"
  },
  "!Content-Type": {
    "name": "Content Type",
    "description": "The content type of the request body.",
    "reference": "@webApplicationContentType",
    "example": "application/json",
    "type": "string"
  },
  "#name": {
    "name": "Name",
    "description": "The name for the new playlist",
    "reference": "@musicPlaylistName",
    "type": "string",
    "example": "A New Playlist"
  },
  "#public": {
    "name": "Public",
    "description": "If true the playlist will be public, if false it will be private",
    "reference": "@musicPlaylistIsPublic",
    "type": "boolean",
    "example": "true",
    "default": true
  }
},

```

Figure 3.7. Spotify Web API with RAD in JSON: parameters

Lastly, Figure 3.8 explores the contents of a method's `responses` key, where all possible responses for that method are listed. Each response is listed by a unique ID, which in the case of the HTTP protocol would be its response code. The required keys in this case are `headers` and `body`, both of which are arrays containing expectations of the response header and body respectively. The `body` itself requires three keys: `reference` links a resource's representation in the activity layer to a corresponding `Resource Concept` in the semantic layer, `media` specifies the response's media type (currently limited to `application/json`), and `type` states the data type of the information in the response's body. Accepted values for `type` are those defined by JSON Schema (i.e. *string*,

```

"responses": {
  "200": {
    ...
  },
  "201": {
    "description": "On success, the response body contains the created playlist object in JSON format",
    "headers": ["Location"],
    "body": {
      "type": "object",
      "reference": "@MusicPlaylist",
      "properties": {
        "href": {
          "type": "hyperlink",
          "target": "/v1/users/{@userSpotifyIdentifier}/playlists/{@musicPlaylistSpotifyIdentifier}"
        },
        "id": {
          "type": "string",
          "reference": "@musicPlaylistSpotifyIdentifier"
        },
        "name": {
          "type": "string",
          "reference": "@musicPlaylistName"
        },
        ...
      }
    }
  }
}

```

Figure 3.8. Spotify Web API with RAD in JSON: responses

integer, *number*, *object*, *array*, *boolean*, *null*), as well as *hyperlink*. The *hyperlink* value requires an additional *target* key to indicate the URI of a referenced resource in the response (Hypermedia Control).

3.3. RAD as a graph

RAD elements and their relationships are modeled as a single graph, presented in Figure 3.9, which mimics the metamodel previously shown in Figure 3.1. Nodes and edges stemming from the RAD Concept Vocabulary, form part of the metamodel's Semantic Layer, while all other graph elements stem from each service's descriptions, therefore belonging to the metamodel's Activity Layer.

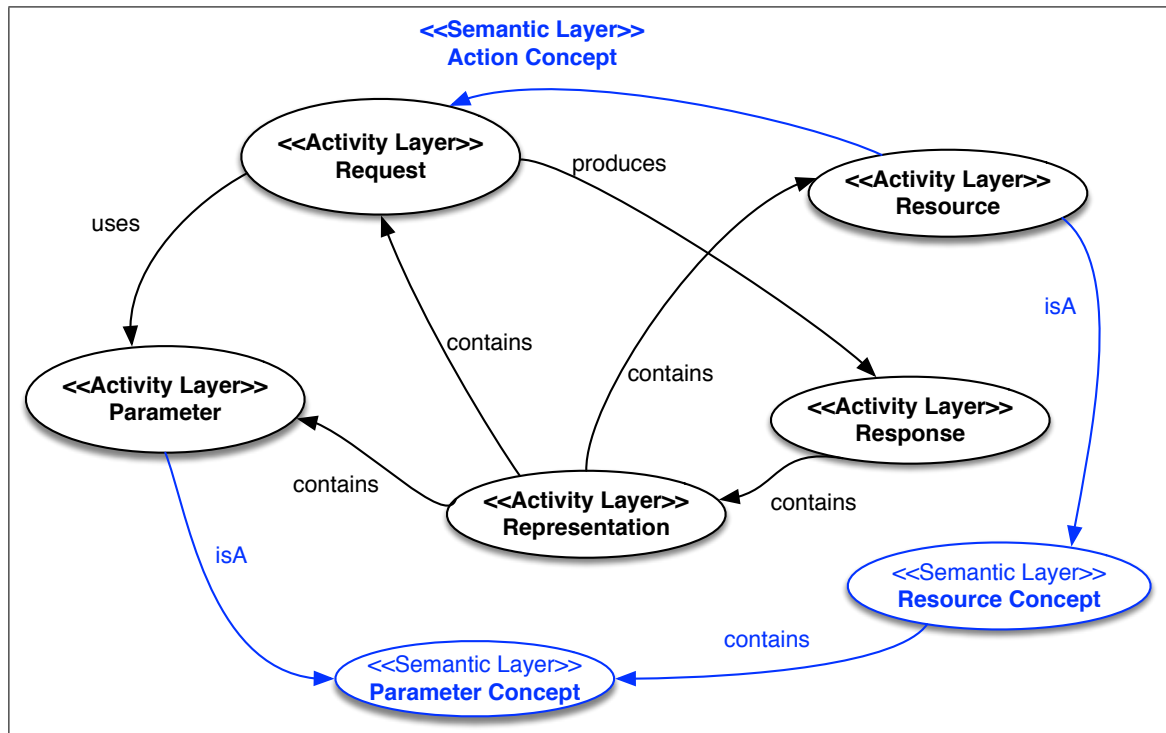


Figure 3.9. RAD graph model

4. RAD-NLQ: NATURAL LANGUAGE REST RESOURCE DISCOVERY

4.1. RAD-based Service Discovery

As previously stated, the semantic layer presented in the RAD metamodel could consist of a vocabulary, or even a full-fledged ontology. It is individually tied with the activity layer through its resource, parameters, and the underlying method in the request. In practice, this means that each `Resource` refers to a semantic `Resource Concept`, each `Parameter` refers to a `Parameter Concept`, and each `Method` to an `Action Concept`.

Given that resources, parameters, and methods are semantically annotated, a machine can now make sense of the underlying business goal in a method executed over a resource. For example, let's consider the case of an API which allows users to rent apartments with a resource `/apartments/4` accepting HTTP POST requests. Not much information can be extracted from this description, as the resource representing an apartment may not necessarily have a self-descriptive name, and a POST method could mean anything, including buying, building, or renting the apartment. In a RAD description, the resource would be referencing the `https://schema.org/Apartment` concept, and the POST method would reference the `https://schema.org/RentAction` concept. Since both concepts are unambiguously described in the Schema documentation, it is clear that by executing a POST HTTP method over the `/apartments/4` resource will allow us to rent such apartment.

Now that we know the business goal associated to a method-resource pair (hypermedia control), we may solve the reverse problem: to discover the resources and methods that allow us to achieve a specific business goal. We understand a business goal as an `Action` concept that has an effect on a `Resource` concept and is grounded through a set of instances of method, resource, and parameter concepts, optionally restricted on how the execution of the resulting request takes place.

With a resource concept (e.g. `http://schema.org/MusicGroup`), an action concept (e.g. `http://schema.org/FindAction/`), and optionally one or more parameter concepts (e. g. `http://schema.org/MusicGroup/name/`) we can query the RAD graph. For instance, we can search for *all the resources referencing a given concept, that are affected by a given action*, and optionally *require the presence of certain parameter concepts*. This query will return all the resources with the corresponding methods that would allow us to achieve our business goal. For example, to find out a music group by its name we shall search for methods described by a `http://schema.org/FindAction/` concept associated to resources described by a `http://schema.org/MusicGroup` concept that accept parameters described by a `http://schema.org/MusicGroup/ name/` concept. Even though such search query can be executed on the RAD graph, its expression is cumbersome: the user must know in advance the syntax of the concept URLs, its meaning and the whole set of available concepts which requires out-of-band information (i.e. description documents, manuals, examples, etc.).

4.2. Supporting Natural Language Queries: RAD-NLQ

In this section we introduce RAD-NLQ, a framework allowing users to query the RAD graph through natural language queries. Such queries allow users to discover the resources and methods which achieve the business goal stated in the query. Two challenges that arise from this approach need to be solved: the extraction of relevant *query concepts*, and the matching of those *query concepts* with the appropriate concepts in the RAD vocabulary.

Based on the analysis of typical grammatical forms of user queries presented in (Barr et al., 2008) (see excerpt in Table 4.1), we designed and implemented an algorithm to extract *concepts* from a natural language search phrase (see Algorithm 1). Once a noun is found, all adjectives directly preceding it and the nouns directly following it are extracted and are considered as a `Resource` concept. Meanwhile, all verbs, and particles directly following them, are extracted and considered as an `Action` concept. Finally, if a preposition

Algorithm 1 Pseudo-code of the concept extraction algorithm

Input: *tagged_words* (an ordered list of the query’s words and their tags)

Output: The query’s extracted *concepts*, categorized by type

```
1: concepts  $\leftarrow$  An object containing three arrays that store resource concepts, action
   concepts, and parameter concepts
2: tagged_words  $\leftarrow$  tagged_words without apostrophes
3: i  $\leftarrow$  0
4: while i < length of tagged_words do
5:   tagged_word  $\leftarrow$  tagged_words[i]
6:   if tagged_word.tag is a noun then
7:     concept  $\leftarrow$  tagged_word.word
8:     k  $\leftarrow$  i - 1
9:     while k  $\geq$  0 do
10:      k_tagged_word  $\leftarrow$  tagged_words[k]
11:      if k_tagged_word.tag is not an adjective then
12:        break
13:      end if
14:      concept  $\leftarrow$  k_tagged_word.word + " " + concept
15:      k  $\leftarrow$  k - 1
16:    end while
17:    k  $\leftarrow$  i + 1
18:    while k < length of tagged_words do
19:      k_tagged_word  $\leftarrow$  tagged_words[k]
20:      if k_tagged_word.tag is not a noun nor an adjective then
21:        break
22:      end if
23:      concept  $\leftarrow$  concept + " " + k_tagged_word.word
24:      i  $\leftarrow$  k
25:      k  $\leftarrow$  k + 1
26:    end while
27:    concepts.resources.add(concept)
28:  else if tagged_word.tag is a verb then
29:    concept  $\leftarrow$  tagged_word.word
30:    k  $\leftarrow$  i + 1
31:    while k < length of tagged_words do
32:      k_tagged_word  $\leftarrow$  tagged_words[k]
33:      if k_tagged_word.tag is not a particle then
34:        break
35:      end if
```

Algorithm 1 Pseudo-code of the concept extraction algorithm (continued)

```
36:         concept  $\leftarrow$  concept + " " + k_tagged_word.word
37:         i  $\leftarrow$  k
38:         k  $\leftarrow$  k + 1
39:     end while
40:     concepts.resources.add(concept)
41:     else if tagged_word.tag is a preposition,  $i \geq 0$ , and tagged_words[i - 1].tag is
    not a verb then
42:         concept  $\leftarrow$  null
43:         k  $\leftarrow$  i + 1
44:         while k < length of tagged_words do
45:             k_tagged_word  $\leftarrow$  tagged_words[k]
46:             if k_tagged_word.tag is a noun or an adjective then
47:                 if concept is null then
48:                     concept  $\leftarrow$  k_tagged_word.word
49:                 else
50:                     concept  $\leftarrow$  concept + " " + k_tagged_word.word
51:                 end if
52:             else
53:                 if concept is not null then
54:                     concepts.parameters.add(concept)
55:                 end if
56:                 concept  $\leftarrow$  null
57:             end if
58:             i  $\leftarrow$  k
59:             k  $\leftarrow$  k + 1
60:             if k = length of tagged_words, and concept is not null then
61:                 concepts.parameters.add(concept)
62:             end if
63:         end while
64:     end if
65:     i  $\leftarrow$  i + 1
66: end while
67: if length of concepts.actions = 0 then
68:     concepts.actions.add("get")
69: end if
70: return concepts
```

is detected, all nouns and adjectives following the same grammatical rules as those used to extract Resource concepts are extracted as Parameter concepts. If at the end of the

execution no action concept is extracted, the default `http://schema.org/GetAction/` (i.e. equivalent to HTTP GET) action concept is assumed.

Table 4.1. Typical grammatical structure of queries used by Web searchers with a distribution based on a sample of 222 hand-labeled queries (Excerpt)

Grammatical Type	Example	Freq %
noun-phrase	free mp3s	69.8%
URI	<code>http://answers.yahoo.com/</code>	10.8%
word salad	mp3s free	8.1%
other-query	florida elementary reading conference2006-2007	6.8%
unknown	nama-nama calon praja ipdn	2.7%
verb-phrase	download free mp3s	1.4%
question	where can I download free mp3s	0.45%

RAD-NLQ inputs are verb phrases, that is, for the case of *question-type* queries, our algorithm will remove the question portion and will treat the query as a verb phrase. Noun phrases will be treated the same way as verb phrases, as if no action concept is found a default "get" action concept is assumed afterwards. Therefore, our approach supports a 71.65% of the query structures (see section 4.2 for a detailed explanation) used on Web search engines. Other grammatical types are not supported since they are not appropriate for API searches (i.e. "URI" and "unknown"), or the grammatical structure of the query does not provide information regarding the expectations of the user (i.e. "word salad").

Once all relevant concepts have been extracted, a suitable match in the vocabulary must be made for each one of them. Algorithm 2 presents our approach for implementing such match. Each extracted *query concept* is matched against relevant concepts in the vocabulary and a similarity score is calculated; those scores that pass a minimum threshold are selected as suitable candidates to be matched with. Once all suitable candidates are selected (resource concepts, action concepts, and optionally parameter concepts), they are permuted. Each permutation is assigned a score, consisting on the aggregated score of each of its candidate concepts. Once again, those scores that pass a threshold are considered as the closest to the business goal stated in the initial query.

Algorithm 2 Pseudo-code of the concept permutation matching algorithm

Input: *query, vocabulary*

Output: *permutations*

```
1:  $q\_resource\_concept \leftarrow$  The resource concept in the query
2:  $q\_action\_concept \leftarrow$  The action concept in the query
3:  $q\_parameter\_concept \leftarrow$  The resource concept in the query if any, else null
4:  $v\_resource\_concepts \leftarrow$  All resource concepts in the vocabulary
5:  $v\_action\_concepts \leftarrow$  All action concepts in the vocabulary
6:  $v\_parameter\_concepts \leftarrow null$ 
7: if  $q\_parameter\_concept$  is not null then
8:    $v\_parameter\_concepts \leftarrow$  All parameter concepts in the vocabulary
9: end if
10:  $candidate\_resource\_concepts \leftarrow [ ]$ 
11:  $candidate\_action\_concepts \leftarrow [ ]$ 
12:  $candidate\_parameter\_concepts \leftarrow [ ]$ 
13: for each  $rc\_candidate$  in  $v\_resource\_concepts$  do
14:    $rc\_candidate.score \leftarrow$  concept similarity score between  $q\_resource\_concept$  and  $rc\_candidate$ 
15:   if  $rc\_candidate.score \geq$  resource concept score threshold then
16:      $candidate\_resource\_concepts.add(rc\_candidate)$ 
17:   end if
18: end for
19: for each  $ac\_candidate$  in  $v\_action\_concepts$  do
20:    $ac\_candidate.score \leftarrow$  concept similarity score between  $q\_action\_concept$  and  $ac\_candidate$ 
21:   if  $ac\_candidate.score \geq$  action concept score threshold then
22:      $candidate\_action\_concepts.add(ac\_candidate)$ 
23:   end if
24: end for
25: if  $q\_parameter\_concept$  is not null then
26:   for each  $pc\_candidate$  in  $v\_parameter\_concepts$  do
27:      $pc\_candidate.score \leftarrow$  concept similarity score between  $q\_parameter\_concept$  and  $pc\_candidate$ 
28:     if  $pc\_candidate.score \geq$  parameter concept score threshold then
29:        $candidate\_parameter\_concepts.add(pc\_candidate)$ 
30:     end if
31:   end for
32: end if
```

Algorithm 2 Pseudo-code of the concept permutation matching algorithm (continued)

```
33: permutations  $\leftarrow$  List of permutations of candidate_resource_concepts,  
    candidate_action_concepts, and candidate_parameter_concepts (if any)  
34: for each permutation in permutations do  
35:   permutation_score  $\leftarrow$  multiplication of scores for all two or three (with parameter concept) concepts  
36:   if permutation_score < concept permutation score threshold then  
37:     delete permutation from permutations  
38:   end if  
39: end for  
40: return permutations
```

5. IMPLEMENTATION

5.1. RAD-QL: an API to query the RAD graph

In order to offer an simple interface to query the RAD graph we created RAD-QL as a query language-like API which enables the traversal of the RAD graph. RAD-QL can currently answer 10 different types of queries, such as obtaining all necessary elements to execute a successful REST API call, obtaining the most similar concepts in the `Semantic Layer` to a given input, and obtaining all `Operation` nodes directly related to a given `Resource Concept` and `Action` pair.

5.2. System overview

Our implementation architecture is presented in Figure 5.1. We store the RAD graph instance in a `Neo4J 3.0`¹ graph database. The database is accessed solely by `rad-core`, a module written in Python 3² which provides an API to query the RAD graph. Two modules, also written in Python3, populate the graph: `vocabulary-parser` parses our vocabulary JSON document and adds the `Semantic Layer` to the graph; while `json-description-parser` processes each RAD JSON document and adds the service to the graph, creating the respective `Activity Layer` nodes in the graph, and linking them to the `Semantic Layer`.

The `query-engine` module is the only access point for end users to query the graph, and as such it is the main module concerning this thesis. It (1) suggests RAD-QL queries based on input parameters, (2) processes and answers RAD-QL queries, and most importantly (3) can suggest RAD-QL queries based on natural language queries. This module was developed using Python3, using the Natural Language Toolkit's (NLTK)

¹Neo4J <https://neo4j.com/>

²Python 3.x documentation <https://docs.python.org/3/>

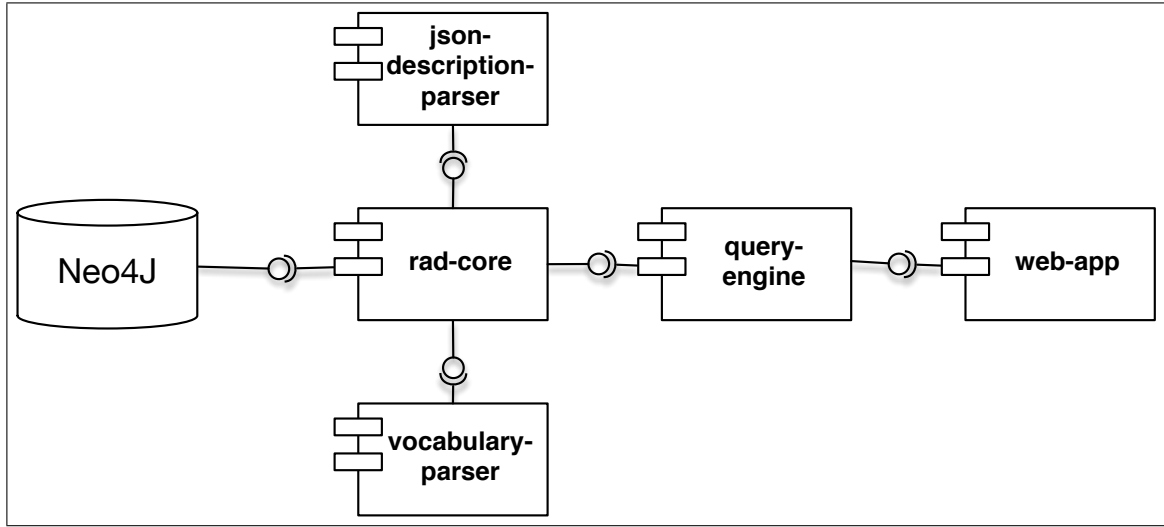


Figure 5.1. RAD-NLQ System UML Component Diagram

³PorterStemmer⁴, as well as the standard Part-Of-Speech tagger `pos_tag` and tokenizer `word_tokenize` functions to process the input phrase. Once the input phrase has been analyzed, and the key concepts and actions have been extracted, each of them is compared to existing concepts in the RAD graph's semantic layer using the UMBC Semantic Similarity service⁵ (Han, Kashyap, Finin, Mayfield, & Weese, 2013). Finally, entities are found and replaced using the Dandelion API⁶.

Finally, the `web-app` module is a Web application available at <http://rad.ing.puc.cl/demo/query>, and screenshots can be seen in Figure 5.2 and Figure 5.3. This application allows users to consume all three functionality provided by the `query-engine`. Depending on the query, the user will receive an interactable response in the form of: (1) a suggested RAD-QL query to execute, (2) a subgraph of the RAD graph which can be

³Natural Language Toolkit (NLTK) <http://www.nltk.org/>

⁴Porter Stemmer <https://tartarus.org/martin/PorterStemmer/>

⁵UMBC Semantic Similarity service <http://swoogle.umbc.edu/SimService/index.html>

⁶Dandelion API <https://dandelion.eu/>

Query Interface Demo

Use natural language or RAD-QL to query the RAD graph. RAD-QL is our query language-like interface to easily query the graph.

☒ Natural language
☐ RAD-QL
☐ Query suggestion

SUBMIT >

Figure 5.2. Screenshot of the RAD-NLQ interface in the Web application: query input

traversed, or (3) a link. For further explanation and screenshots of this module see Appendix A. This module was developed with the Express⁷ framework over Node.js⁸ and AngularJS⁹, and graphs are rendered using vis.js¹⁰.

⁷Express <http://expressjs.com/>

⁸Node.js <https://nodejs.org/en/>

⁹AngularJS <https://nodejs.org/en/>

¹⁰vis.js <http://visjs.org/>

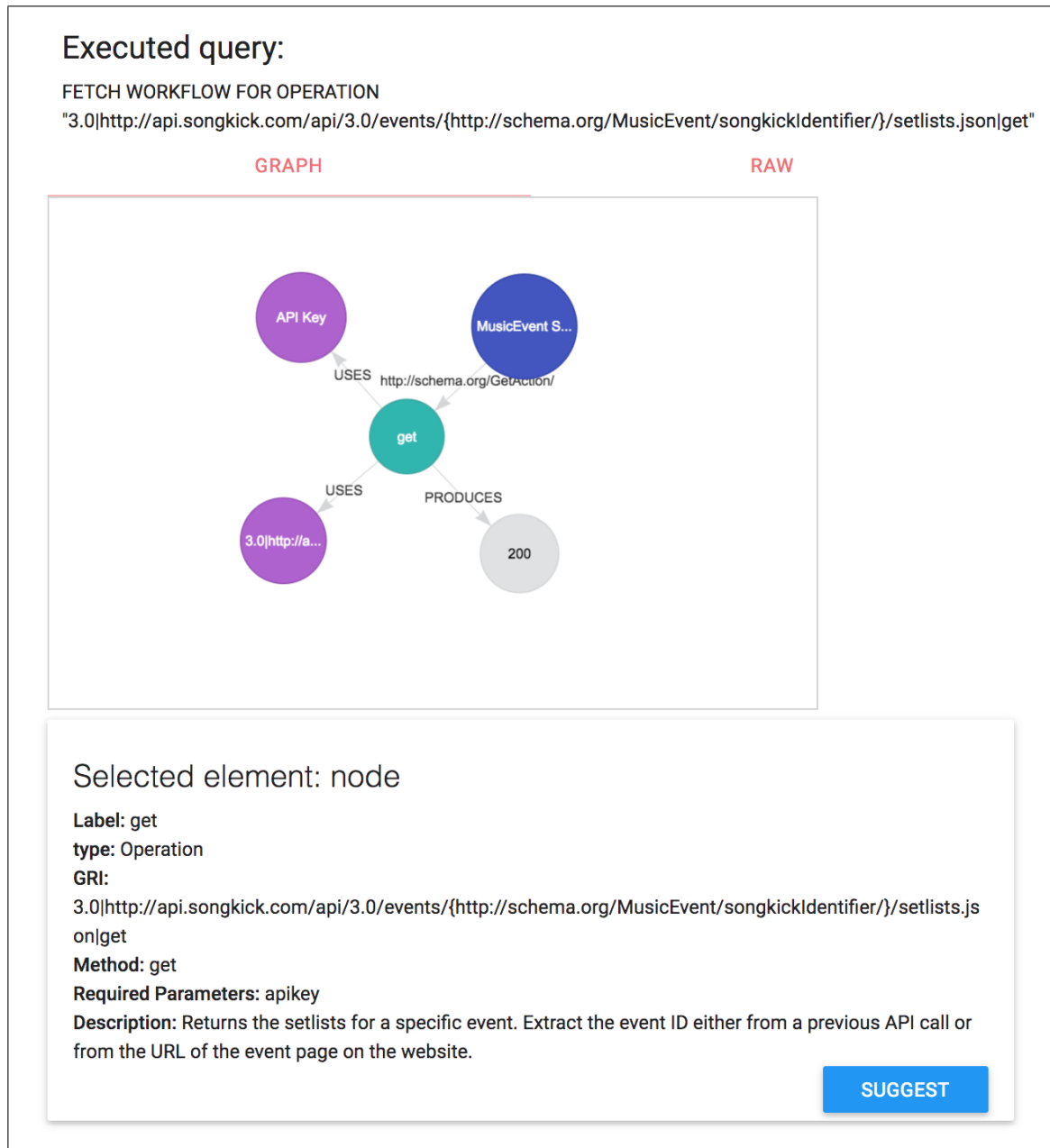


Figure 5.3. Screenshot of the RAD-NLQ interface in the Web application: query graph response

6. EVALUATION

6.1. Vocabulary and API Dataset

In order to evaluate our proposal we chose to use real, industry-standard Web APIs. Selected APIs had to adhere to REST's constraints as much as possible, provide a comprehensive documentation website, and their business domain should be somewhat related. Due to the nature of the Web APIs we found, we were forced to relax our criteria regarding some of REST's constraints. A total of six Web APIs were selected:

- (i) **Foursquare**¹: Provides *read* and *creation* methods to its directory of venues and their *events* resources, as well as a user's *private profile* and *lists*.
- (ii) **Google Travel Partner**²: Only two out of ten resources were included in our example: (1) the *Hotels API 2.0*³ provides programmatic access to a user's *hotel list feed*, and (2) the *Prices API 2.0*⁴ allows users to *query pricing* and *itinerary data* for a given hotel.
- (iii) **Songkick**⁵: Provides access to its *live music database*, including *past and upcoming concerts*, *artist search and suggestion*, and *concert setlists*.
- (iv) **Spotify**⁶: Provides access to its *music streaming service's catalog*, including *artists*, *albums*, *playlists*, and *tracks*.
- (v) **Taxi Fare Finder**⁷: Allows a user to get an *fare estimate* for a given taxi ride, retrieves a list of *registered taxi companies* in a given city, and searches for *supported cities*.

¹Foursquare API <https://developer.foursquare.com/docs/>

²Google Travel Partner's API <https://developers.google.com/hotels/hotel-ads/api-reference/>

³Google Travel Partner's Hotel API <https://developers.google.com/hotels/hotel-ads/api-reference/hotels-api-v2>

⁴Google Travel Partner's Prices API <https://developers.google.com/hotels/hotel-ads/api-reference/prices-api-v2>

⁵Songkick API <http://www.songkick.com/developer>

⁶Spotify API <https://developer.spotify.com/web-api/endpoint-reference/>

⁷Taxi Fare Finder API <https://www.taxifarefinder.com/api.php>

- (vi) **Uber**⁸: Allows users to obtain information and estimates of Uber rides as well as cancel requested Uber rides. It also provides access to a user’s profile and history, as well as the ability to retrieve all available products for a user.

These Web APIs were described by a RAD JSON document as described in section 3.2 and then parsed into the RAD graph presented in section 3.3. Table 6.1 presents a general overview of the resulting graph. Table 6.2 details the vocabulary subgraph composition, and Table 6.3 details the activity layer nodes by API. Finally, Table 6.4 details the use of distinct semantic layer concepts across all APIs.

Table 6.1. Nodes and edges in the graph database

	Nodes	Edges
Vocabulary	405	463
Foursquare	612	1174
Google Travel Partner	29	50
Songkick	318	579
Spotify	611	1189
Taxi Fare Finder	47	82
Uber	175	304
Total	2197	3841

Table 6.2. Vocabulary subgraph composition

Concept Type	Distinct Elements
Resource Concepts	106
Actions	11
Parameter Concepts	299

6.2. Input Query Dataset

We consider a use case to be a combination of one *Resource* Concept, one *Action* Concept, and optionally one *Parameter* Concept, with a clear underlining business goal. In

⁸Uber API <https://developer.uber.com/docs/riders/references/api>

Table 6.3. Activity layer nodes

API	Resources	Methods	Parameters	Responses	Representations
Foursquare	12	12	562	13	13
Google Travel Partner	2	2	21	2	2
Songkick	15	15	260	15	13
Spotify	20	27	513	28	23
Taxi Fare Finder	3	3	35	3	3
Uber	10	11	129	15	10
Total	62	70	1520	76	64

Table 6.4. Use of distinct semantic layer concepts

API	Resource Concepts	Actions	Parameter Concepts
Foursquare	8	4	120
Google Travel Partner	2	1	17
Songkick	10	3	38
Spotify	13	9	63
Taxi Fare Finder	3	2	21
Uber	9	3	46

order to create a testbed we defined the vocabulary of concepts from the APIs documentation. That is, we retrieved all documentation websites for each RAD-indexed resource and each API, and extracted relevant keywords to act as *Resource* Concepts, *Action* Concepts, and *Parameter* Concepts. A total of 42 unique use cases were formed through this process.

In order to eliminate bias in the grammatical construction of the input search query phrases used in our evaluation, we automatically generated such phrases. In subsection 4.2 we analyze how and why our concept extraction algorithm supports a 71.65% of user query’s grammatical compositions across web search engine users through the direct support of only one of them: verb phrases. Therefore, we created the *SimplePhraseTransform*⁹ tool for generating phrases based on simplified grammatical rules constructed out of part-of-speech tags used in the Penn Treebank Project¹⁰. Our input set consists of a total of 128

⁹SimplePhraseTransform’s Source Code <https://github.com/rad-lab/simple-phrasetransform>

¹⁰Penn Treebank Project https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html

queries generated out of the original 42 use cases, and its composition can be viewed in Table 6.5.

Table 6.5. Composition of our input dataset of 128 queries. RC: Resource Concept, A: Action, PC: Parameter Concept, E: Entity

Query Type	Example	Freq %
RC + A	search for venues	73.44%
RC + A + PC	search for venues by their name	19.53%
E + A	get U2's upcoming events	7.03%

6.3. Evaluation Criteria

We evaluated our model through the direct comparison of our system versus the Google search engine using as input the same automatically generated natural language phrase-queries, since all other API repositories search engines listed in section 2.2 do not support this kind of search queries. For a proper comparison, we used a similar rating criteria for the search results of both systems (our approach and the Google search engine). We also restricted the Google search domain to the API's documentation through the use of the 'site:' command in order to eliminate results from sources not in the dataset. Equation 6.1 defines our `ranking_score` based on the position of the item in the response (the higher the rank, the lower the value). Equation 6.2 defines a `correctness_score` based on the content of the response: we consider that a *partial* answer for a query only includes *some* of the expected items in the response, while a *tangential* answer satisfies the query goal through the execution of an indirectly related action and resource (e.g. obtaining a price estimate for the taxi by canceling the request itself).

$$\begin{aligned}
 & \text{ranking_score}(\text{ranked_position}) = \\
 & \begin{cases} 1.1 - 0.1 * \text{ranked_position} & \text{if } 1 \leq \text{ranked_position} \leq 10 \\ 0 & \text{else} \end{cases} \quad (6.1)
 \end{aligned}$$

$$\begin{aligned}
& correctness_score(content) = \\
& \left\{ \begin{array}{ll} 1 & \text{if the response } content \text{ directly answers the query} \\ 0.8 & \text{if the response } content \text{ partially answers the query} \\ 0.5 & \text{if the response } content \text{ tangentially answers the query} \\ 0.2 & \text{if the response } content \text{ has a link to the query's answer} \\ 0 & \text{if the response } content \text{ does not answer the query} \end{array} \right. \quad (6.2)
\end{aligned}$$

We also consider the `concept_extraction_score` (Equation 6.3) which is applied only to the RAD-NLQ system. This score measures the ability of the system to use all possible concepts present in the input to form a RAD-QL query. We also define the `hit_score` (Equation 6.4) which is applied only to the Google search engine results. This score measures the content of each link according to the number of relevant elements in the target Web page (e.g. a single URL documenting all the APIs resources will score lower than a URL that targets a specific resource directly related to the answer).

$$concept_extraction_score(used_concepts, total_concepts) = \frac{used_concepts}{total_concepts} \quad (6.3)$$

$$hit_score(number_of_elements) = \frac{1}{number_of_elements} \quad (6.4)$$

Finally, we defined the performance scores for RAD-NLQ (Equation 6.5) and the Google search engine (Equation 6.6). These scores will be directly compared in the next section (higher is better).

$$\begin{aligned}
rad_nlq_score(responses) = & \sum_{r \in responses} (ranking_score(ranked_position_r) \\
& \times correctness_score(content_r) \\
& \times concept_extraction_score(used_concepts_r, total_concepts_r)) \quad (6.5)
\end{aligned}$$

$$\begin{aligned}
google_score(responses) = & \sum_{r \in responses} (ranking_score(ranked_position_r) \\
& \times correctness_score(content_r) \\
& \times hit_score(number_of_elements_r)) \quad (6.6)
\end{aligned}$$

6.4. Results

An overview of the results of our evaluation are presented in Table 6.6, and a histogram of the result's scores can be observed in Figure 6.1. At a first glance we see that queries including entities instead of concepts are all successfully resolved by RAD-NLQ, while the Google search engine is unable to correctly answer these queries. Additionally, queries which have a parameter as a restriction are much more reliably answered by RAD-NLQ.

Further inspection on queries where RAD-NLQ was outperformed by Google reveals some limitations of our system. On 7 queries (25.92%) RAD-NLQ performed correctly, averaging a respectable score of 0.65; Google, on the other hand, did not only offer the required link in one of its top ranks, but also delivered a link to the APIs documentation homepage which included a link to the resource documentation. Thus, it obtained a higher score though it did not provide new information but redundant information. Another 9 failed queries (33.33%) are related to use cases solely involving complex resources without a unique concept representing them, but rather a collection of parameter concepts. Such

Table 6.6. Evaluation results. RC: Resource Concept, A: Action, PC: Parameter Concept, E: Entity

Query Type	Success frequency by system (%)			
	RAD-NLQ	Draws	Google	Total
RC + A	67 (52.34%)	4 (3.13%)	23 (17.97%)	94 (73.44%)
RC + A + PC	20 (15.63%)	1 (0.78%)	4 (3.13%)	25 (19.53%)
E + A	9 (7.03%)	0 (0.00%)	0 (0.00%)	9 (7.03%)
Total	96 (75.00%)	5 (3.91%)	27 (21.09%)	128 (100%)

is the case of Google Travel Partner’s Price¹¹ resource which offers a given hotel’s pricing and itinerary data, as well as Foursquare’s Venue Hours¹² resource which offers exclusively a given venue’s opening and popular hours. In both cases there is no clear underlying concept for them, and as such, our concept matching performs poorly. In all other 11 failed cases (40.74%), the problem can be traced to correctly extracted concepts failing to match concepts in our vocabulary.

¹¹Google Travel Partner’s Prices API <https://developers.google.com/hotels/hotel-ads/api-reference/prices-api-v2>

¹²Foursquare API’s Venue Hours resource <https://developer.foursquare.com/docs/venues/hours>

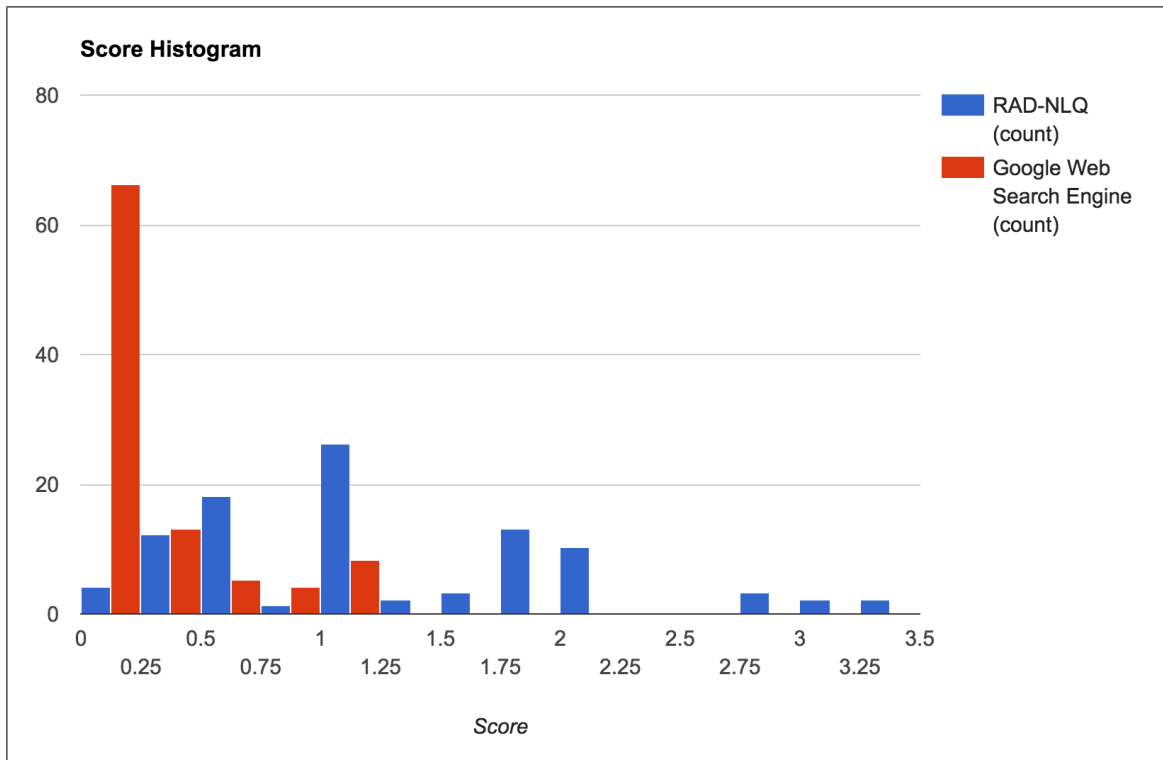


Figure 6.1. Result's score histogram for RAD-NLQ and the Google Web Search Engine

7. CONCLUSIONS

In this thesis we offer further validation of the RAD description metamodel for REST APIs, which is capable of representing well-known real Web APIs. RAD is able to successfully support most common practices in Web API design, including a lightweight model of resources, methods, required parameters and their data type and location in the request (i.e. URL, header, or body), and all responses as well as their parameters.

This thesis also offers further validation of the metagraph derived from the metamodel. The metagraph allows for the discovery of services at a resource-method pair level in order to achieve a given business goal. This metagraph also allows flexibility on its implementation: we have shown in this thesis the use of a graph database and the Schema.org dictionary as our semantic layer, but any other graph-based approach is supported, including a full-fledged RDF-based ontology as the semantic layer.

Most importantly, this thesis is proof that the RAD metamodel and metagraph allow for the discovery of services at a resource-method pair level through Web search engine-like natural language phrase queries. We have presented a framework able to support natural language beyond industry-standard keywords, but rather phrases, which successfully outperforms Google’s Web search engine in most Web API discovery use cases.

An important advantage of our proposal is that the description constitutes a separate layer not only from the semantic layer itself, but also from the implementation of the API: the RAD description has no effect over data exposed by the API, its functionality, and the supported media types (e.g. JSON, XML, YAML, HTML, etc.). An important disadvantage stems from this: the description is tightly coupled to the API’s implementation. By allowing the description to be flexible and not having an effect on the implementation, coupling occurs the other way around, where any change on the API must be reflected on its documentation, decreasing service evolvability. Considering most APIs maintain a documentation website anyways, this coupling could be maintained, and not increased,

through the generation of the documentation website itself from its RAD description document.

Another limitation appears when analyzing real industry-level Web API's design: though common Web API design practices are supported, special cases exist. As discussed in section 6.4, not all resources have a clear underlying concept to be represented by, but rather a collection of loose parameters. Another case we encountered is when the the presence or value of a parameter directly changes the concept representing the URL. Both cases are an inconvenience in our our discovery efforts, and though they could be attributed to problems in the API's design, these cases occur in well-known Web APIs, and should be acknowledged.

Additionally, the quality and precision of our concept matching process results are limited by the the complexity and completeness of the semantic layer in use. In this theses we based our semantic layer on the Schema.org vocabulary, but ontologies and other graph-based solutions could also be used. A semantic layer with more detailed and complete relationships could be exploited in order to obtain more precise results than those shown in this thesis. This would significantly increase the solution's complexity, as the underlying algorithm for the concept matching process would have to be specifically tailored for each variation of the semantic layer.

As of future work, we will focus on improving RAD-NLQ's results through the use of an ontology over the Schema.org vocabulary approach, allowing us to make use of more complex relationships between concepts. We also aim to improve our concept similarity process which is directly linked from the previous goal. Finally, we look forward on supporting both border case limitations previously discussed, through the support of virtual one-time-use concepts composed by loosely coupled parameters.

REFERENCES

- Adida, B., Birbeck, M., McCarron, S., & Pemberton, S. (2008, September). *RDFa in xHTML: Syntax and processing a collection of attributes and processing rules for extending xHTML to support RDF* (Tech. Rep.). World Wide Web Consortium.
- Alarcón, R., Saffie, R., Bravo, N., & Cabello, J. (2015). REST web service description for grpah-based service discovery. In *Engineering the web in thee big data era* (pp. 461–478). Springer.
- Alarcón, R., & Wilde, E. (2010, April). RESTler: crawling RESTful services. In *Proceedings of the 19th international conference on World Wide Web* (pp. 1051–1052). ACM.
- Barr, C., Jones, R., & Regelson, M. (2008, October). The linguistic structure of English Web-search queries. In *Proceedings of the conference on empirical methods in natural language processing* (pp. 1021 – 1030). ACM.
- Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H., ... Winer, D. (2000, May). *Simple Object Access Protocol (SOAP) 1.1* (Tech. Rep.). World Wide Web Consortium.
- Chinnici, R., Moreau, J., Ryman, A., & Weerawarana, S. (2007, June). *Web Services Description Language (WSDL) version 2.0 part 1: core language* (Tech. Rep.). World Wide Web Consortium.
- Fielding, R. (2000). *Architectural styles and the design of network-based software architectures* (Unpublished doctoral dissertation). University of California, Irvine.
- Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., & Berners-Lee, T. (1999, June). *Hypertext Transfer Protocol – HTTP/1.1* (Tech. Rep.). Internet Engineering Task Force.

Gunasri, R., & Kanagaraj, R. (2014). Natural language processing and clustering based service discovery. *International Journal of Scientific & Technology Research*, 3(4), 28–31.

Hadley, M. (2009, August). *Web Application Description Language* (Tech. Rep.). World Wide Web Consortium.

Han, L., Kashyap, A., Finin, T., Mayfield, J., & Weese, J. (2013). UMBC EBIQUITY-CORE: Semantic textual similarity systems. In *Proceedings of the second joint conference on lexical and computational semantics*. Association for Computational Linguistics.

John, D., & Rajasree, M. (2013). RESTDoc: Describe, discover and compose RESTful semantic web services using annotated documentations. *International Journal of Web & Semantic Technology*, 4(1), 37–49.

Kelly, M. (2016, May). *JSON hypertext application language* (Tech. Rep.). Internet Engineering Task Force.

Khare, R., & Çelik, T. (2006). Microformats: a pragmatic path to the semantic Web. In *Proceedings of the 15th international conference on world wide web* (pp. 865–866).

Kopecky, J., Gomadam, K., & Vitvar, T. (2008). An HTML microformat for describing RESTful web services. In *International conference on web intelligence and intelligent agent technology*.

Lakshmi, D., & Dhas, J. (2013). An user-friendly and improved semantic-based web service discovery approach using natural language processing techniques. *International Journal of Innovative Research in Computer and Communication Engineering*, 1(10), 2435–2442.

Lanthaler, M., & Gütl, C. (2013). Hydra: A vocabulary for hypermedia-driven web APIs. In *Proceedings of the 6th workshop on linked data on the web at the 22nd international world wide web conference*.

Lathem, J., Gomadam, K., & Sheth, A. P. (2007). SA-REST and (s)mashups: Adding semantics to RESTful services. In *Proceedings of the international conference on semantic computing* (pp. 469 – 476).

Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., . . . Sycara, K. (2004, November). *OWL-S: Semantic markup for web services* (Tech. Rep.). World Wide Web Consortium.

Saffie, R. (2016). *Towards automatic service composition in REST* (Unpublished master's thesis). Pontificia Universidad Católica de Chile.

Sangersa, J., Frasinara, F., Hogenbooma, F., & Chepeginb, V. (2013). Semantic web service discovery using natural language processing techniques. *Expert Systems with Applications*, 40(11), 4660–4671.

Spink, A., Jansen, B., Wolfram, D., & Saracevic, T. (2002). From e-sex to e-commerce: Web search changes. *IEEE Computer*, 35(3), 107–109.

Sporny, M., Longley, D., Kellogg, G., Lanthaler, M., & Lindström, N. (2014, January). *JSON-LD 1.0-a JSON-based serialization for linked data* (Tech. Rep.). World Wide Web Consortium.

Verborgh, R., Steiner, T., Deursen, D. V., Roo, J. D., de Walle, R. V., & Gabarró, J. (2011). Description and interaction of RESTful services for automatic discovery and execution. In *Proceedings of the FTRA 2011 international workshop on advanced future multimedia services*.

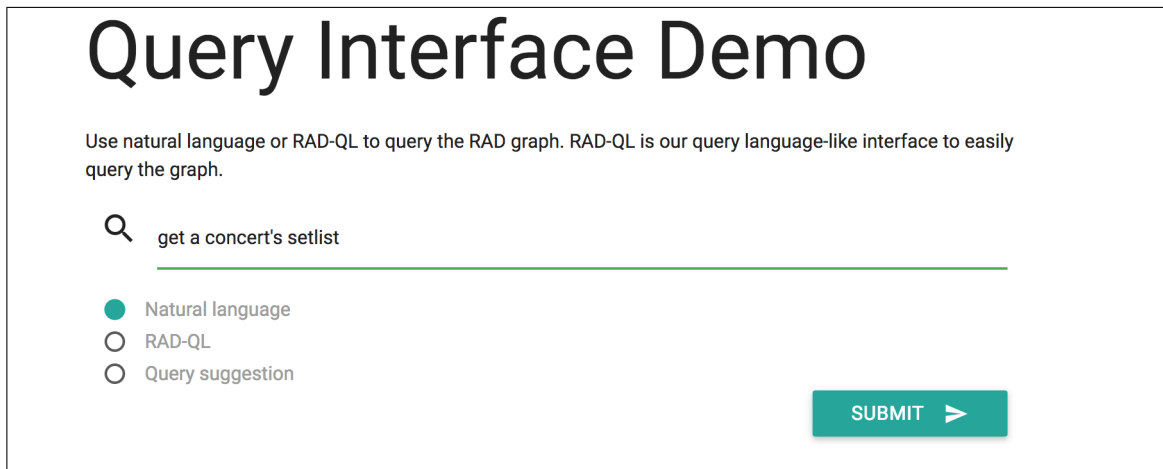
Verborgh, R., Steiner, T., Deursen, D. V., Roo, J. D., de Walle, R. V., & Gabarró, J. (2013). Capturing the functionality of Web services with functional descriptions. *Multimedia tools and applications*, 64(2), 365–387.

Webber, J., Parastatidis, S., & Robinson, I. (2010). *REST in practice: Hypermedia and systems architecture*. O'Reilly Media, Inc.

APPENDIX

A. WEB APPLICATION SCREENSHOTS

This appendix details the user’s flow throughout the Web Application described in Chapter 5 when discovering `resource.method` pairs through natural language phrase queries. The demonstration presented in this appendix is available at rad.ing.puc.cl/demo/query.



The screenshot shows a web interface titled "Query Interface Demo". Below the title is a subtitle: "Use natural language or RAD-QL to query the RAD graph. RAD-QL is our query language-like interface to easily query the graph." There is a search input field with a magnifying glass icon on the left and the text "get a concert's setlist". Below the input field are three radio button options: "Natural language" (which is selected), "RAD-QL", and "Query suggestion". To the right of these options is a green button labeled "SUBMIT" with a right-pointing arrow.

Figure A.1. RAD-NLQ: User’s natural language phrase input

Figure A.1 presents the data input field for the Web Application. A total of 3 queries modes are supported: Natural language, RAD-QL, and Query Suggestion. The first mode is selected and allows the user to input a natural language phrase query in the query box (e.g. ”get a concert’s setlist”). The former 2 modes will be explained further ahead.

The response to the natural language query submitted in Figure A.1 is presented in Figure A.2. The response contains a list of RAD-QL queries containing the extracted concepts from the input query, ranked by the amount of extracted concepts out of the total are used to form each RAD-QL query (the numbers on the right). Each RAD-QL query in the response is interactable, and upon clicking one the query is inputted into the query box and the query mode is switched to RAD-QL. Figure A.3 presents this scenario when selecting the suggested highest-scored RAD-QL query which allows us to discover resources and methods involving the specified resource and action concepts.

Executed query:	
get a concert's setlist	
TABLE	RAW
SUGGEST SERVICES FOR CONCEPT "concert setlist" AND ACTION "get"	1
SUGGEST SIMILAR ACTION CONCEPTS FOR "get"	0.5
SUGGEST SIMILAR RESOURCE CONCEPTS FOR "concert setlist"	0.5

Figure A.2. RAD-NLQ: Concept extraction from user's input

☐ Natural language
 ☒ RAD-QL
 ☐ Query suggestion

SUBMIT >

Figure A.3. RAD-NLQ: RAD-QL query to discover resource-method pairs with the extracted concepts

Figure A.4 presents the response to the query inputted in Figure A.3. A ranked list of interactable method node IDs are presented, as they serve as they are unique to a resource-method pair, as a single method node can only be associated to a single resource node. Each ID is accompanied buy a number ($\geq 0, \leq 1$) on the right representing how well the execution of that resource-method pair answers the RAD-QL query.

Upon selecting the best-ranked method node ID, the Query suggestion query mode is selected and the ID is added to the list. This mode allows users to find RAD-QL queries based on the type of parameters they have, alongside filling the returned query's template

Executed query:

SUGGEST SERVICES FOR CONCEPT "concert setlist" AND ACTION "get"

TABLE

RAW

3.0 http://api.songkick.com/api/3.0/events/{http://schema.org/MusicEvent/songkickIdentifier}/setlists.json get	0.62175
3.0 http://api.songkick.com/api/3.0/artists/{http://schema.org/MusicGroup/songkickIdentifier}/gigography.json get	0.25751

Figure A.4. RAD-NLQ: Ranked `method` node IDs linked to the extracted concepts

Input type

Value

REMOVE

Operation GRI

▼

3.0|http://api.songkick.com/api/3.0/events/{http://s

Input type

Value

ADD

Select input type

▼

Value

☐ Natural language
 ☐ RAD-QL
 ☒ Query suggestion

SUBMIT ➤

Figure A.5. RAD-NLQ: Obtaining RAD-QL query suggestions for the selected `method` node ID

with the stated value for each parameter. This can be seen in Figure A.6, where a list of RAD-QL queries is presented which use the given parameters, alongside a number by which they are ranked representing the proportion of parameters used to form the query.

Finally, upon executing the only RAD-QL query returned in Figure A.6, which aims to retrieve the workflow for the `method` node’s ID, an interactable graph is returned. This graph is a subgraph of the RAD graph, and presents the `method` node, alongside its associated `resource`, `parameters`, and `responses`, as well as all relationships between

Executed query:

operation_gri:"3.0|http://api.songkick.com/api/3.0/events/{http://schema.org/MusicEvent/songkickIdentifier}/setlists.json|ge

TABLERAW

FETCH WORKFLOW FOR OPERATION

"3.0|http://api.songkick.com/api/3.0/events/{http://schema.org/MusicEvent/songkickIdentifier}/setlists.json|get" 1

Figure A.6. RAD-NLQ: RAD-QL queries suggested for the given method node ID

them. All nodes and edges can be selected in order to view more detailed information about them. Additionally, all nodes can be directly added into the Query suggestion query mode as parameters, allowing users to continue exploring the RAD graph.

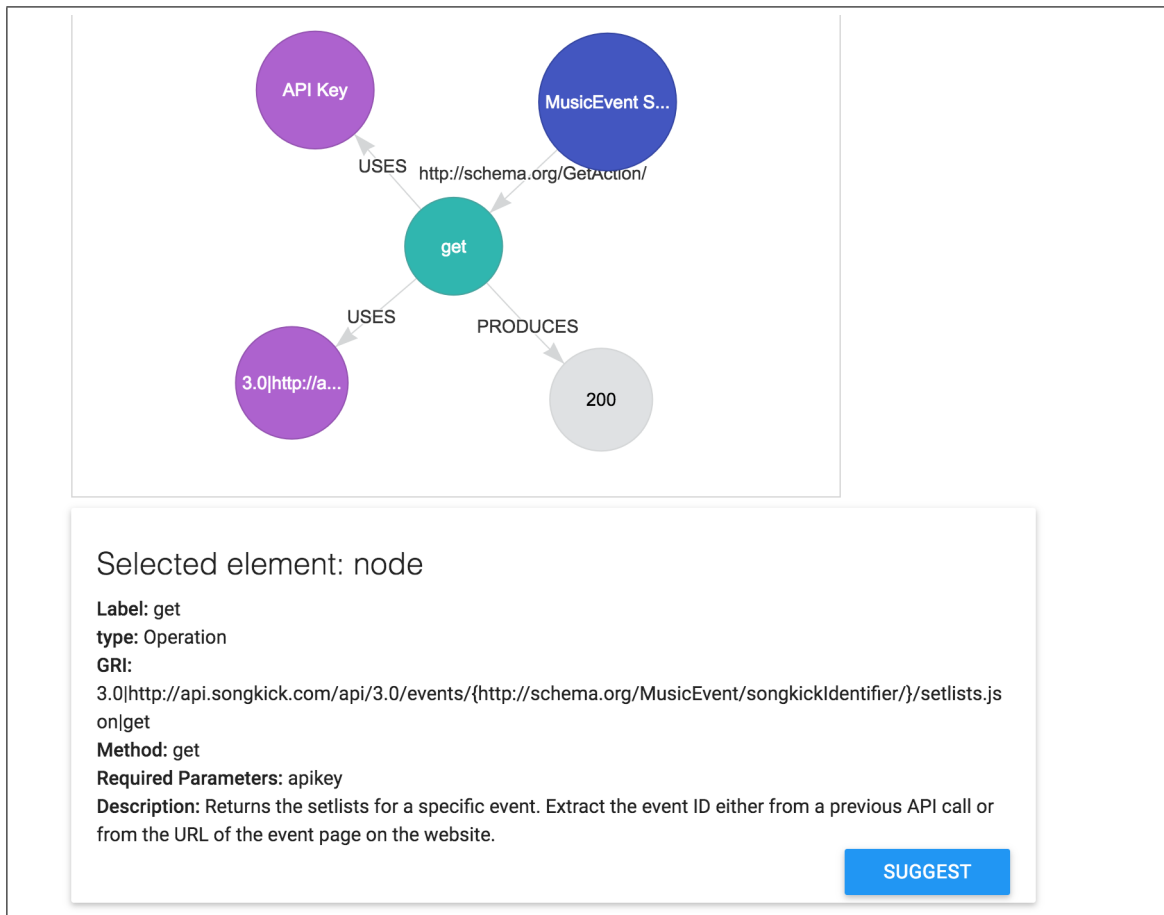


Figure A.7. RAD-NLQ: Interactive graph detailing the workflow for the given method node