A Catalogue of Inter-Parameter Dependencies in RESTful Web APIs

Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés

Department of Computer Languages and Systems
Universidad de Sevilla, Spain
{amarlop,sergiosegura,aruiz}@us.es

Abstract Web services often impose dependency constraints that restrict the way in which two or more input parameters can be combined to form valid calls to the service. Unfortunately, current specification languages for web services like the OpenAPI Specification provide no support for the formal description of such dependencies, which makes it hardly possible to automatically discover and interact with services without human intervention. Researchers and practitioners are openly requesting support for modelling and validating dependencies among input parameters in web APIs, but this is not possible unless we share a deep understanding of how dependencies emerge in practice—the aim of this work. In this paper, we present a thorough study on the presence of dependency constraints among input parameters in web APIs in industry. The study is based on a review of more than 2.5K operations from 40 real-world RESTful APIs from multiple application domains. Overall, our findings show that input dependencies are the norm, rather than the exception, with 85% of the reviewed APIs having some kind of dependency among their input parameters. As the main outcome of our study, we present a catalogue of seven types of dependencies consistently found in RESTful web APIs.

Keywords: Web services · constraints · parameter dependencies.

1 Introduction

Web Application Programming Interfaces (APIs) allow systems to interact with each other over the network, typically using web services [10, 17]. Web APIs are rapidly proliferating as the cornerstone for software integration enabling new consumption models such as mobile, social, Internet of Things (IoT), or cloud applications. Popular API directories such as ProgrammableWeb [14] and RapidAPI [16] currently index over 21K and 8K web APIs, respectively, from multiple domains such as shopping, finances, social networks, or telephony.

Modern web APIs typically adhere to the REpresentational State Transfer (REST) architectural style, being referred to as RESTful web APIs [7]. *RESTful web APIs* are decomposed into multiple web services, where each service implements one or more create, read, update, or delete (CRUD) operations over

a resource (e.g. a tweet in the Twitter API), typically through HTTP interactions. RESTful APIs are commonly described using languages such as the OpenAPI Specification (OAS) [13], originally created as a part of the Swagger tool suite [19], or the RESTful API Modeling Language (RAML) [15]. These languages are designed to provide a structured description of a RESTful web API that allows both humans and computers to discover and understand the capabilities of a service without requiring access to the source code or additional documentation. Once an API is described in an OAS document, for example, the specification can be used to generate documentation, code (clients and servers), or even basic automated test cases [19]. In what follows, we will use the terms RESTful web API, web API, or simply API interchangeably.

Web services often impose dependency constraints that restrict the way in which two or more input parameters can be combined to form valid calls to the service, we call these *inter-parameter dependencies* (or simply *dependencies* henceforth). For instance, it is common that the inclusion of a parameter requires or excludes—and therefore depends on—the use of some other parameter or group of parameters. As an example, the documentation of the YouTube Data API states that when using the parameter `videoDefinition` (e.g. to search videos in high definition) the `type` parameter must be set to `'video'`, otherwise a HTTP 400 code (bad request) is returned. Similarly, the documentation of the cryptocurrency API Coinbase explains that, when placing a buy order, one, and only one of the parameters `total` or `amount` must be provided.

Current specification languages for RESTful web APIs such as OAS and RAML provide little or no support at all for describing dependencies among input parameters. Instead, they just encourage to describe such dependencies as a part of the description of the parameters in natural language, which may result in ambiguous or incomplete descriptions. For example, the Swagger documentation states[1] "*OpenAPI 3.0 does not support parameter dependencies and mutually exclusive parameters. (...) What you can do is document the restrictions in the parameter description and define the logic in the 400 Bad Request response*". The lack of support for dependencies means a strong limitation for current specification languages, since without a formal description of such constraints is hardly possible to interact with the services without human intervention. For example, it would be extremely difficult, possibly infeasible, to automatically generate test cases for the APIs of YouTube or Coinbase without an explicit and machine-readable definition of the dependencies mentioned above. The interest of industry in having support for these types of dependencies is reflected in an open feature request in OAS entitled "Support interdependencies between query parameters", created on June 2015 with the message shown below. At the time of writing this paper, the request has received over 180 votes, and it has received 43 comments from 25 participants[2].

> *"It would be great to be able to specify interdependencies between query parameters. In my app, some query parameters become "required" only*

[1] `https://swagger.io/docs/specification/describing-parameters/`
[2] `https://github.com/OAI/OpenAPI-Specification/issues/256`

when some other query parameter is present. And when conditionally required parameters are missing when the conditions are met, the API fails. Of course I can have the API reply back that some required parameter is missing, but it would be great to have that built into Swagger."

This feature request has fostered an interesting discussion where the participants have proposed different ways of extending OAS to support dependencies among input parameters. However, each approach aims to address a particular type of dependency and thus show a very limited scope. Addressing the problem of modelling and validating input constraints in web APIs should necessarily start by understanding how dependencies emerge in practice. Some previous papers have addressed this challenge, as a part of other contributions, but they have studied just a few popular APIs and so they have only scratched the surface [12, 21] (c.f. Section 5). A systematic and large-scale analysis of the state of practice is needed in order to answer key questions such as how often dependencies appear in practice or what types of input constraints are found in real-world web APIs. This is the goal of our work.

In this paper, we present a thorough study on the presence of inter-parameter dependencies in industrial web APIs. Our study is based on an exhaustive review of 40 RESTful APIs from multiple application domains carefully selected from the API repository of ProgrammableWeb [14]. All APIs were carefully reviewed and classified following a systematic and structured method. Among other results, we found that 85% of the APIs (34 out of 40) had some kind of dependency among their input parameters. More specifically, we identified 633 dependencies in 9.7% of the operations analysed (248 out of 2,557). The identified constraints are classified into a catalogue of seven types of inter-parameter dependencies in RESTful web APIs. This catalogue will hopefully serve as a starting point for future approaches on modelling and analysis of input dependencies in web APIs.

This paper is structured as follows: Section 2 describes the review method followed. Section 3 presents the results of our study. Section 4 describes some potential threats to validity and how they were mitigated. Related work is discussed in Section 5. Finally, Section 6 draws the conclusions and presents future lines of research.

2 Review method

In what follows, we present the research questions that motivate this study as well as the process followed for the collection and analysis of the data.

2.1 Research questions

The aim of this paper is to answer the following research questions (RQs):

RQ1: How common are inter-parameter dependencies in web APIs? We aim to provide an in-depth view of how frequently dependencies appear in practice, trying to find out whether their presence is correlated to certain characteristics or application domains. Once we confirm the presence of dependencies, we will try to understand how they look like answering the following question.

RQ2: What types of inter-parameter dependencies are found in web APIs? We wish to provide a catalogue of the types of dependencies among input parameters most commonly found in real-world APIs, which can serve as a starting point for future proposals for their modelling and analysis.

2.2 Subject APIs and search strategy

The search for real-world APIs was carried out in ProgrammableWeb [14], a popular and frequently updated online repository with about 21K APIs and 8K mashups at the time of writing this paper. We followed a systematic approach for the selection of a subset of highly-used yet diverse APIs, as follows. First, we selected the top 10 most popular APIs in the repository overall. Then, we selected the 3 top-ranked APIs from the top 10 most popular categories in ProgrammableWeb, i.e. those with a larger number of indexed APIs. APIs on each category are ordered according to the number of registered applications consuming them (mashups). We focused on RESTful APIs only, as the de-facto standard for web APIs. In particular, we selected APIs reaching level 1 or higher in the Richardson Maturity Model [17], which ensured a minimal adherence to the REST architectural style, e.g. using the notion of resources. APIs not following the key REST principles and those with poor or no available documentation were discarded, selecting the next one in the list. Some of the selected APIs were found in different categories and were included just once, ignoring duplicates. Table 1 depicts the list of subject APIs analysed in our study, 40 in total. For each API, the table shows its name, category, number of mashups, number of operations and percentage of operations containing parameter dependencies. For the sake of readability, similar categories have been merged into a single one.

Figure 1 shows the classification of subject APIs by category and size. As illustrated, the subject APIs are evenly distributed among 10 different categories such as communication, social and mapping. Regarding the size, the majority of reviewed APIs (75%) provide between 1 and 50 operations, with the largest APIs having up to 305 (DocuSign eSignature) and 492 (Github) operations. Overall, the selected APIs represent a large, diverse, and realistic dataset.

2.3 Data collection and analysis

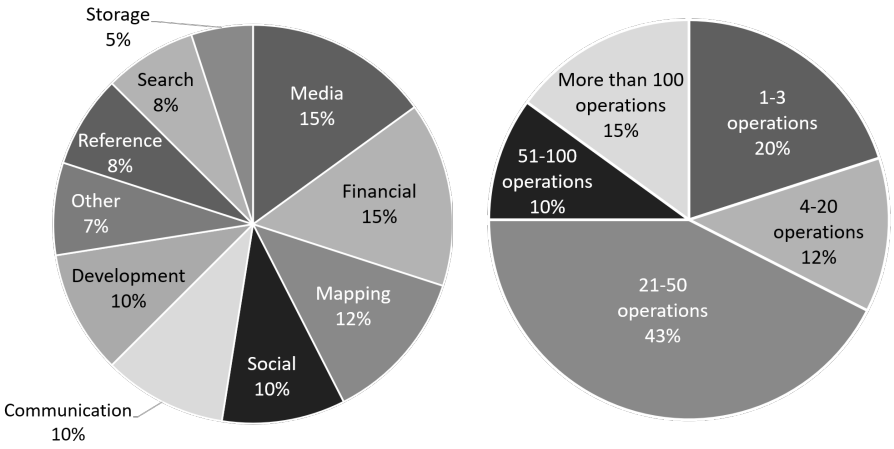
We carefully analysed the information available in the official website of the 40 subject APIs to answer our research questions. For each API, we collected the name, link to the documentation, API version, category, number of mashups and followers registered in ProgrammableWeb, and total number of operations. Additionally, for each operation with dependencies, we collected the number and type of input parameters, type of CRUD operation and inter-parameter dependencies.

Dependencies were identified in two steps. First, we recorded all the dependencies among input parameters found in the documentation of the subject APIs. It is worth mentioning that every dependency can be represented in multiple ways, e.g. in conjunctive normal form. At this point, we strove to represent

Name	Category	Mashups	Operations	%Op. with dep.
Google Maps Places	Mapping	2,579	7	57.1
Twitter Search Tweets	Social	829	3	100
Youtube	Media	707	50	34.0
Flickr	Media	635	222	13.1
Twilio SMS	Communication	361	31	6.5
Last.fm	Media	246	58	31.0
Microsoft Bing Maps	Mapping	175	51	21.6
Google App Engine Admin	Development	124	38	0.0
Foursquare	Social	113	40	40.0
DocuSign eSignature	Other	98	305	4.6
Amazon S3	Storage	95	94	16.0
GeoNames	Reference	90	41	24.4
Bing Web Search	Search	67	1	100
Yelp Fusion	Reference	61	12	41.7
Indeed	Search	48	2	0.0
Paypal Invoicing	Financial	39	21	23.8
Google Custom Search	Search	39	2	0.0
Google Geocoding	Mapping	36	1	100
SoundCloud	Media	34	49	2.0
Oodle	Other	34	1	100
NationBuilder	Social	33	107	5.6
Tumblr	Social	26	25	20.0
OpenStreetMap	Mapping	23	39	5.1
iTunes	Media	22	1	100
Google Fusion Tables	Development	20	33	9.1
Tropo	Communication	19	25	8.0
Heroku	Development	18	262	0.0
MapLarge	Mapping	14	31	0.0
Google Drive	Storage	13	39	10.3
CrunchBase	Reference	11	23	8.7
Github	Development	11	492	2.8
Nexmo SMS	Communication	10	3	33.3
Stripe	Financial	8	220	7.7
Kiva	Financial	8	32	0.0
AT&T In-App Messaging	Communication	7	11	9.1
PicPlz	Media	5	18	61.1
Coinbase	Financial	3	43	7.0
Pryv	Other	1	25	16.0
QuickBooks Payments	Financial	1	20	20.0
Forte (Payments Gateway)	Financial	1	79	19.0

Table 1: List of subject APIs

them as they were described in the documentation of the API. This allowed us, for example, to record the *arity* of each dependency, i.e. number of parameters involved in each constraint. In a second step, we studied the shape of all the dependencies and managed to group them into seven general dependency types (c.f. Section 3.2). Additionally, we used an online text analysis tool [20] to



(a) Categories. (b) Number of operations.

Figure 1: Classification of subject APIs by category and size.

identify the linguistic patterns most frequently used for the description of each type of dependency. The documentation collected from each API was reviewed by at least two different authors to reduce misunderstanding or missing information. The complete dataset used in our study, including all the data collected from each API, is publicly available in a machine-processable format [5].

3 Results

In this section, we describe the results and how they answer the research questions. Firstly, we present how frequently inter-parameter dependencies appear in practice and whether their presence is correlated to certain API characteristics. Secondly, we detail the different types of dependencies found in the subject APIs.

3.1 Presence of inter-parameter dependencies

This section aims to provide an answer to RQ1 by studying how common inter-parameter dependencies are in web APIs and where they are typically found. We identified 633 total dependencies among input parameters in 85% of the APIs under study (34 out of 40). Specifically, we found dependencies in 9.7% of the operations analysed (248 out of 2,557). The percentage of operations with dependencies of each API is shown in the last column of Table 1. This percentage ranged from less than 5%, in APIs such as Soundcloud and Github, to 100%, in APIs such as Bing Web Search and Twitter Search Tweets. Figure 2 depicts a bar graph with the percentage of operations with dependencies on each category. As illustrated, we found dependencies in all the categories under study, with

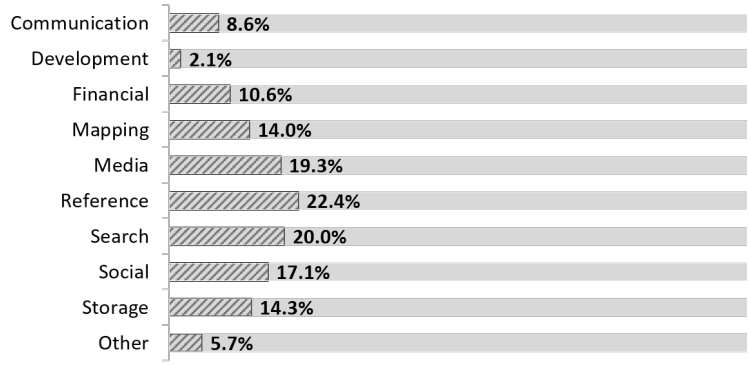


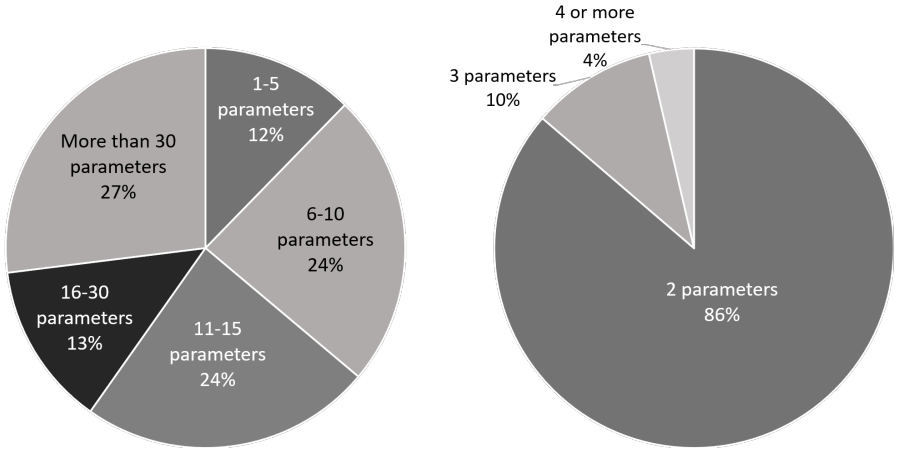
Figure 2: Percentage of operations with dependencies per category.

the percentage of operations with dependencies ranging between 2.1% in the category *development* and 22.4% in the category *reference*. This suggests that the presence of dependencies in real-world APIs is very common, independently of their application domain.

Figure 3a shows the distribution of dependencies by the number of parameters of the operation. Overall, we found that operations with dependencies had between 2 and 221 parameters, 20.1 on average (standard deviation = 21.0, median = 13). Moreover, most of these operations were read operations (61%), followed by create (26%), update (13%) and delete operations (less than 1%). Figure 3b depicts the distribution of dependencies by their arity. The largest portion of dependencies were binary (86%), followed by those involving three (10%) or more parameters (4%). In total, arity ranged between 2 and 10, with dependencies involving 2.2 parameters on average (standard deviation = 0.6, median = 2). Furthermore, the dependencies mostly involved query parameters (65%), followed by body (34%), header (3%) and path parameters (1%). Interestingly, we found 22 dependencies that involved more than one type of parameter. For example, the Bing Web Search API documentation states that, in order to obtain results in a given language, either the `Accept-Language` header or the `setLang` query parameter must be specified, but not both.

Finally, we investigated whether some of the data could be used as effective predictors for the amount of dependencies in a web API. To that end, we studied some potential correlations among the collected data using the R statistical environment in two steps. First, we checked the normality of the data using the Shapiro-Wilk test concluding that the data do not follow a normal distribution. Second, we used the Spearman's rank order coefficient to assess the relationship between the variables. In particular, we tried to answer the following questions:

– *Are APIs with many operations likely to have a higher percentage of operations with dependencies?* No, quite the opposite. Spearman coefficient reveals a moderate negative correlation (ρ = -0.45, p-value = 0.003), which indicates that as the number of operations increases, the percentage of op-



(a) Operation parameters. (b) Dependency arity.

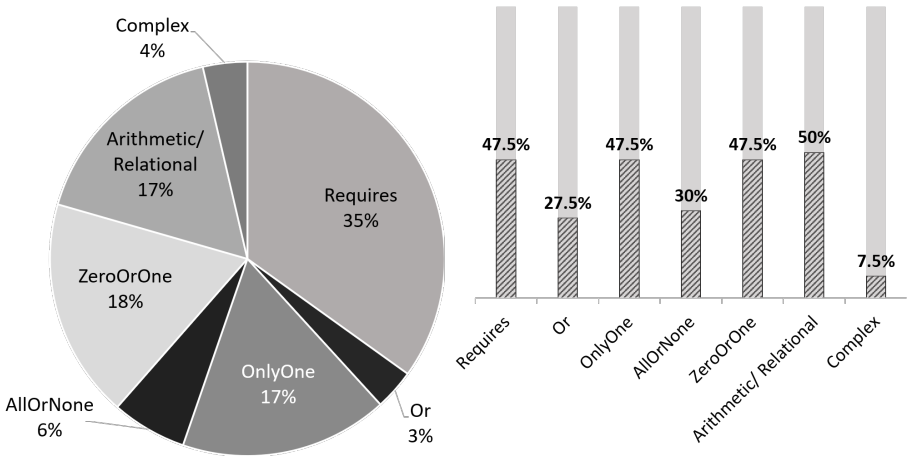
Figure 3: Classification of dependencies by the number of parameters in the operation and their arity.

erations with dependencies decreases, and vice versa. In other words, the percentage of operations with dependencies tends to be higher in APIs with fewer operations. This may be explained by the fact that APIs with few operations often suffer from low cohesion, with a few operations trying to do too many things through the use of a wide set of parameters and dependencies. Conversely, APIs with many operations avoid some dependencies by distributing the functionality across different related operations.

– *Are operations with many parameters likely to have more dependencies?* Yes. Spearman coefficient reveals a moderate positive correlation ($\rho = 0.49$, p-value $= 2.2 \times 10^{-16}$), which means that the number of dependencies in an operation typically increases with the number of input parameters. We found an exception, however, in those operations receiving complex objects as input, where the percentage of object properties with dependencies is usually very low, e.g. a PayPal invoice is composed of 112 JSON properties with just 2 dependencies among them. We repeated the correlation study excluding input objects and obtained a Spearman coefficient of 0.67 (p-value $= 2.2 \times 10^{-16}$), which reflects a stronger positive correlation between the number of parameters of an operation and the number of dependencies.

3.2 Catalogue of inter-parameter dependencies

In this section, we answer RQ2 by classifying the inter-parameter dependencies identified into seven general types. We took inspiration in the constraints used to model dependencies in feature models, in the context of software product lines,



(a) Dependency types. (b) Occurrences in APIs

Figure 4: Distribution of dependencies by type and percentage of APIs.

where the authors have wide expertise [3], although we propose more intuitive and self-explanatory names in our work.

Before going in depth into each type of dependency, a number of considerations must be taken into account. First, for the sake of simplicity, dependencies are described using single parameters. However, all dependencies can be generalized to consider groups of parameters using conjunctive and disjunctive connectors. Second, dependencies can affect not only the presence or absence of parameters, but also the values that they can take. In what follows, when making reference to a parameter *being present* or *being absent*, it could also mean a parameter *taking* or *not taking a given value*, respectively. Finally, when introducing each dependency type we will make reference to Figure 4, which shows the distribution of dependencies by type (Figure 4a) and the percentage of subject APIs including occurrences of each dependency type (Figure 4b). Next, we describe the seven types of dependencies found in our study, including examples.

Requires. The presence of a parameter p_1 in an API call requires the presence of another parameter p_2, denoted as $p_1 \rightarrow p_2$. As previously mentioned, p_1 and p_2 can be generalized to groups of parameters and parameters' assignments, e.g. $a \wedge b = x \rightarrow c \vee d$. Based on our results, this is the most common type of dependency in web APIs, representing 35% of all the dependencies identified in our study (Figure 4a), and being present in 47.5% of the subject APIs (Figure 4b). The syntactical analysis of API documentations revealed that the most frequently used linguistic patterns to describe this type of dependencies are *"you must also set X"*, *"X must also be specified"*, *"only valid if X is"* and *"[yes,/required] if X is specified"*. This type of dependency is equivalent to the *requires* cross-tree constraint in feature models [3].

As an example, in the Paypal Invoicing API, when creating a draft invoice, if the parameter `custom.label` is present, then `custom.custom_amount` becomes required, i.e. `custom.label` $\rightarrow$ `custom.custom_amount`. Similarly, in the YouTube Data API, when searching for videos with a certain definition (parameter `videoDefinition`), the `type` parameter must be set to `'video'`, i.e. `videoDefinition` $\rightarrow$ `type=video`.

Or. Given a set of parameters $p_1, p_2, \ldots, p_n$, one or more of them must be included in the API call, denoted as $Or(p_1, p_2, \ldots, p_n)$. As illustrated in Figure 4, this type of dependencies represent only 3% of the dependencies identified in the subject APIs. Interestingly, however, we found that more than one fourth of the APIs (27.5%) included some occurrence of this type of dependency, which suggests that its use is fairly common in practice. Typical syntactic structures to describe these dependencies are *"X or Y must be set"* and *"required if X is not provided"*. This type of dependency is equivalent to the *or* relationship in feature models [3].

As an example, when setting the information of a photo in the Flickr API, at least one of the parameters `title` or `description` must be provided, i.e. Or(`title`, `description`). Similarly, in the DocuSign eSignature API, at least one of the parameters `from_date`, `envelope_ids` or `transaction_ids` must be submitted in the API call when retrieving the status of several envelopes, i.e. Or(`from_date`, `envelope_ids`, `transaction_ids`).

OnlyOne. Given a set of parameters $p_1, p_2, \ldots, p_n$, one, and only one of them must be included in the API call, denoted as $OnlyOne(p_1, p_2, \ldots, p_n)$. As observed in Figure 4, this group of dependencies represent 17% of all the dependencies identified, and they appear in almost half of the APIs under study (47.5%). Among others, we found that this type of dependency is very common in APIs from the category *media*, where a resource can be identified in multiple ways, e.g. a song can be identified by its name or by its ID, and only one value must be typically provided. Common syntactic structures for describing this type of dependencies are *"specify one of the following"*, *"only one of X or Y can be specified"*, *"use either X or Y"*, *"required unless X"* and *"required if X is not provided"*. This type of dependency is equivalent to the *alternative* constraint in feature models [3].

For example, in the Twilio SMS API, when retrieving the messages of a particular account, either the parameter `MessagingServiceSid` or the parameter `From` must be included, but not both at the same time, i.e. $OnlyOne$(`MessagingServiceSid`, `From`). Similarly, when deleting a picture in the PicPlz API, only one of the parameters `id`, `longurl_id` or `shorturl_id` must be submitted in the API call, i.e. $OnlyOne$(`id`, `longurl_id`, `shorturl_id`).

AllOrNone. Given a set of parameters $p_1, p_2, \ldots, p_n$, either all of them are provided or none of them, denoted as $AllOrNone(p_1, p_2, \ldots, p_n)$. Very similarly to the Or dependency type, only 6% of the dependencies found belong to this category, nonetheless, they are present in about one third of the APIs under study (30%). These dependencies are typically described with structures such

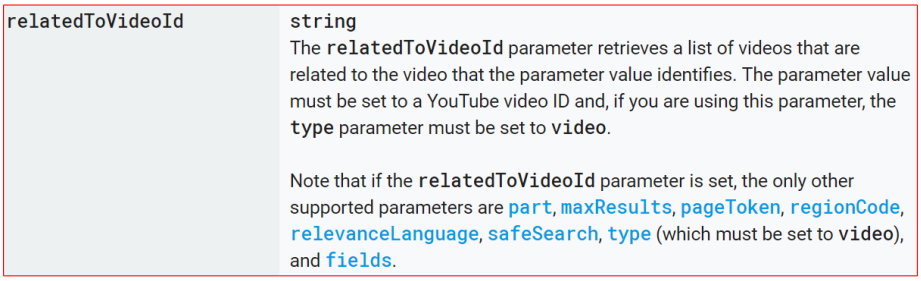


Figure 5: Description of a parameter in the YouTube API that implies multiple *ZeroOrOne* dependencies.

as *"can only be used in conjunction with"*, *"required if X is provided"* and *"(in conjunction) with X, (...)"*.

In the GitHub API, for example, the operation to obtain information about a user accepts two optional parameters, `subject_type` and `subject_id`, and they must be used together, i.e. $AllOrNone$(`subject_type`, `subject_id`). In the payments API Stripe, when creating a Stock Keeping Unit (a specific version of a product, used to manage the inventory of a store), if `inventory.type` is set to `'finite'`, then `inventory.quantity` must be present, and vice versa, i.e. $AllOrNone$(`inventory.type=finite`, `inventory.quantity`).

ZeroOrOne. Given a set of parameters $p_1, p_2, \ldots, p_n$, zero or one can be present in the API call, denoted as $ZeroOrOne(p_1, p_2, \ldots, p_n)$. Figure 4 reveals that this type of dependency is common both in terms of the number of occurrences (18% of the total) and the number of APIs including it (47.5%). Commonly used linguistic patterns for describing this type of dependency are *"not supported for use in conjunction with"*, *"cannot be combined with"*, *"if X is set, the only other supported parameters are"* and *"mutually exclusive with X"*.

Interestingly, about one third of the occurrences of this dependency type were found in YouTube, where filtering by a video ID in the search operation restricts the allowed parameters it can be combined with to only 8, as shown in Figure 5. Since the operation accepts other 22 optional parameters, they are related to the video ID parameter by means of *ZeroOrOne* dependencies, e.g. $ZeroOrOne$(`relatedToVideoId`, `topicId`). Other examples of this dependency type include those where the use of a parameter restricts the allowed values of another parameter, like in the Google Maps API: when searching for places nearby, if `radius` is present, then `rankby` cannot be set to `'distance'`, i.e. $ZeroOrOne$(`radius`, `rankby=distance`).

Arithmetic/Relational. Given a set of parameters $p_1, p_2, \ldots, p_n$, they are related by means of arithmetic and/or relational constraints, e.g. $p_1 + p_2 > p_3$. As shown in Figure 4, this type of dependency is the most recurrent across the subject APIs, being present in half of them. Moreover, 17% of the dependencies found are of this type. These dependencies are typically implicit by the meaning

Value	Description
browse	Find venues within a given area. Unlike the `checkin intent`, browse searches an entire region instead of only finding venues closest to a point. A region to search can be defined by including either the `ll` and `radius` parameters, or the `sw` and `ne`. The region will be circular if you include the ll and radius parameters, or a bounding box if you include the sw and ne parameters.

Figure 6: Complex dependency present in the `GET /venues/search` operation of the Foursquare API.

of the parameters. For example, in a hotel booking, the `checkout` date should be later than the `checkin` date.

As an example, in the GeoNames API, when retrieving information about cities, the `north` parameter must be greater than the `south` parameter for the API to return meaningful results, i.e. `north` > `south` (`north`, `east`, `south` and `west` are the coordinates of a bounding box conforming the search area). In the payments API Forte, when creating a merchant application, this can be owned by several businesses, in which case the sum of the percentages cannot be greater than 100, i.e. `owner.percentage` + `owner2.percentage` + `owner3.percentage` + `owner4.percentage` <= `100`.

Complex. These dependencies involve two or more of the types of constraints previously presented. Based on our results, they are typically formed by a combination of *Requires* and *OnlyOne* dependencies. As illustrated in Figure 4, we found 4% of complex dependencies, being present in 7.5% of the subject APIs.

For example, in the Tumblr API, when creating a new post, if the `type` parameter is set to `'video'`, then either `embed` or `data` must be specified, but not both, i.e. `type=video` $\rightarrow$ *OnlyOne*(`embed`, `data`). Figure 6 shows an extract of the documentation of the search operation in the Foursquare API. As illustrated, if `intent` is set to `'browse'`, then either `ll` and `radius` are present or `sw` and `ne` are present, i.e. `intent=browse` $\rightarrow$ *OnlyOne*((`ll` $\wedge$ `radius`), (`sw` $\wedge$ `ne`)).

4 Threats to validity

The factors that could have influenced our study and how these were mitigated are summarised in the following internal and external validity threats.

Internal validity. This concerns any factor that might introduce bias. The main source of bias is the subjective and manual review process conducted for identifying dependencies among input parameters in the online documentation of the subject APIs. It is possible that we missed some dependencies or that we misclassified some of them. To mitigate this threat, the documentation of each API was carefully checked several times recording all the relevant data for its later analysis, and also to enable replicability. This was an extremely time-consuming process, but it was somehow alleviated by the familiarity of the authors with web APIs—all the authors have years of experience in the development of service-oriented systems for teaching, research and industrial purposes.

The impact of possible mistakes was also minimised by the large number of APIs reviewed (40 APIs and 2,557 operations), which makes us remain confident of the overall accuracy of the results.

External validity. Threats to external validity relate to the degree to which we can generalise from the experiments. Our study is based on a subset of RESTful web APIs, and thus our results could not generalise to other APIs. To minimise this threat, we systematically selected a large set of real-world APIs from multiple application domains. This set includes some of the most popular APIs in the world with millions of users worldwide.

5 Related work

Two related papers have addressed the issue of parameter dependencies in contemporary web APIs. Wu et al. [21] presented an approach for the automated inference of parameter dependencies in web services. As a part of their work, they studied four popular RESTful web APIs and classified the dependencies found into six types, four of which are specific instances of the *Requires* dependency presented in our work. Oostvogels et al. [12] proposed a Domain-Specific Language (DSL) for the description of inter-parameter constraints in OAS. They first classified the dependencies typically found in web services into three types: *exclusive* (called *OnlyOne* in our work), *dependent* (*Requires* in our work), and *group constraints* (*AllOrNone* in our paper). Then, they looked for instances of those types of dependencies in the documentation of six popular APIs by searching for specific keywords such as "either" or "one of". Compared to theirs, our work presents a much larger and systematic study: we have manually reviewed 40 APIs from different domains, whereas they have jointly studied 7 "popular" APIs. As a result, the conclusions drawn from our investigation differ sharply from those derived from their papers. Among other differences, we identified a richer set of dependencies (e.g. Oostvogels et al. [12] identified three out of the seven types of dependencies found in our work), and collected a much larger amount of data (e.g. Oostvogels et al. [12] found 19 dependencies in YouTube while we found 82). Consequently, the general trends observed in our paper also differ, e.g. Wu et al. [21] found that an average of 21.9% of service operations had dependency constraints, while in our study that percentage is 9.7%. As a further difference, our work comprises a much more thorough analysis of dependencies including aspects such as their arity, frequently used linguistic patterns and correlations. Overall, however, the three papers complement each other and support the need for supporting inter-parameter dependencies in web APIs.

Several authors have addressed the problem of input dependencies in web services using the Web Services Description Language (WSDL). Xu et al. [22] analysed multiple service specifications to extract different types of constraints that enable syntax, workflow and semantic testing. One type of constraint they were able to infer is inter-parameter dependencies, but no details were given regarding their type and number of occurrences. Cacciagrano et al. [4] identified

three types of constraints present in input parameters that hinder the automated invocation of services, one of them being inter-parameter dependencies (e.g. the value of a parameter being conditioned to the value of some other), and proposed an XML-based framework for their formalisation. Gao et al. [9] integrated information about parameters, error messages and testing results to infer data preconditions on web APIs that sometimes are not correctly specified in their documentation. They studied two web services and identified constraints involving one parameter (e.g. an integer that must be lower than certain value) or several parameters (e.g. two parameters that cannot be used together). Compared to them, our work is the first systematic and large-scale study of input constraints in modern web APIs, including a catalogue of the types of constraints most commonly found in practice.

Finally, our work is related to testing approaches for web services where dependency management is a key point to generate valid test cases. Recent contributions on testing of RESTful services [1,2,6,18] have succeeded to automatically generate test cases to some extent, however, none of them support the automated management of dependencies among input parameters. What is more, checking the existence of inter-parameter dependencies could be considered a black-box test coverage criterion to fulfil when testing RESTful APIs [11]. This would, in turn, enable the automatic generation of more thorough test suites. This paper takes a step further to address these challenges.

6 Conclusions and future work

In this paper, we reviewed the state of practice on the existence of inter-parameter dependencies in RESTful web APIs. To the best of our knowledge, this is the first systematic study on the topic, and the largest one, with 40 real-world APIs and more than 2.5K operations reviewed. Our results show that dependencies are extremely common and pervasive—they appear in 85% of the APIs under study across all application domains and types of operations. The collected data helped us to characterise dependencies identifying their most common shape—dependencies in read operations involving two query parameters—, but also exceptional cases such as dependencies involving up to 10 parameters and dependencies among different types of parameters, e.g. header and body parameters. We also identified some correlations pointing at the number of operations and the number of parameters as helpful estimators of the amount of dependencies in a web API. As the main result of our study, we present a catalogue of seven types of inter-parameter dependencies consistently found in all the subject APIs. We trust that the results of this study will provide the basis for future research contributions on modelling and analysis of input constraints in web APIs, enabling a more precise description of their capabilities and opening a new range of possibilities in terms of automation in areas such as code generation and testing.

Several challenges remain for future work. On the one hand, it would be desirable to perform an empirical study assessing the validity of the conclusions drawn from our investigation. On the other hand, the results of our study set

the ground for approaches for modelling dependencies among input parameters in web APIs. Such proposals should ultimately reach industrial standards, as in the case of tools such as SLA4OAI [8], an OAS extension to model and manage Service Level Agreements (SLAs) in APIs. Our work enables the creation of multiple tools of this kind, namely: a DSL for the description of dependencies; a documentation analyser for the automatic inference of inter-parameter dependencies based on the linguistic patterns found; a tool for the automatic detection of dependencies at *run-time*; and a dependency analyser for the discovery of inconsistencies between multiple dependency constraints, e.g. a *dead* parameter that can never be selected.

Acknowledgements

This work has been partially supported by the European Commission (FEDER) and Spanish Government under projects BELI (TIN2015-70560-R) and HORATIO (RTI2018-101204-B-C21), and the FPU scholarship program, granted by the Spanish Ministry of Education and Vocational Training (FPU17/04077). We would also like to thank Enrique Barba Roque and Julián Gómez Rodríguez for their help in analysing the documentation of some of the APIs considered for this study.

References

1. Arcuri, A.: RESTful API Automated Test Case Generation with EvoMaster. ACM Trans. on Software Engineering and Methodology **28**(1), 3 (2019)
2. Atlidakis, V., Godefroid, P., Polishchuk, M.: REST-ler: Automatic Intelligent REST API Fuzzing. Tech. rep. (April 2018)
3. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated Analysis of Feature Models 20 Years Later: A Literature Review. Information Systems **35**(6), 615 – 636 (2010)
4. Cacciagrano, D., Corradini, F., Culmone, R., Vito, L.: Dynamic Constraint-based Invocation of Web Services. In: 3rd Intern. Workshop on Web Services and Formal Methods. pp. 138–147 (2006)
5. Inter-Parameter Dependencies in RESTful APIs [Dataset] (2019), `https://bit.ly/2wvv1m1`
6. Ed-douibi, H., Izquierdo, J.L.C., Cabot, J.: Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach. In: IEEE 22nd Intern. Enterprise Distributed Object Computing Conference. pp. 181–190 (2018)
7. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis (2000)
8. Gamez-Diaz, A., Fernandez, P., Ruiz-Cortés, A.: Automating SLA-Driven API Development with SLA4OAI. In: 17th Intern. Conference on Service-Oriented Computing (2019)
9. Gao, C., Wei, J., Zhong, H., Huang, T.: Inferring Data Contract for Web-based API. In: IEEE Intern. Conference on Web Services. pp. 65–72 (2014)
10. Jacobson, D., Brail, G., Woods, D.: APIs: A Strategy Guide. O'Reilly Media, Inc. (2011)

11. Martin-Lopez, A., Segura, S., Ruiz-Cortés, A.: Test Coverage Criteria for RESTful Web APIs. In: Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST '19) (2019)
12. Oostvogels, N., De Koster, J., De Meuter, W.: Inter-parameter Constraints in Contemporary Web APIs. In: 17th Intern. Conference on Web Engineering. pp. 323–335 (2017)
13. OpenAPI Specification, `https://github.com/OAI/OpenAPI-Specification`, accessed March 2019
14. ProgrammableWeb API Directory, `http://www.programmableweb.com/`, accessed March 2019
15. RESTful API Modeling Language (RAML), `http://raml.org/`, accessed March 2019
16. RapidAPI API Directory, `https://rapidapi.com`, accessed March 2019
17. Richardson, L., Amundsen, M., Ruby, S.: RESTful Web APIs. O'Reilly Media, Inc. (2013)
18. Segura, S., Parejo, J.A., Troya, J., Ruiz-Cortés, A.: Metamorphic Testing of RESTful Web APIs. IEEE Trans. on Software Engineering **44**(11), 1083–1099 (2018)
19. Swagger, `http://swagger.io/`, accessed March 2019
20. Text Analyzer - Text analysis Tool, `https://www.online-utility.org/text/analyzer.jsp`, accessed April 2019
21. Wu, Q., Wu, L., Liang, G., Wang, Q., Xie, T., Mei, H.: Inferring Dependency Constraints on Parameters for Web Services. In: Proceedings of the 22nd Intern. Conference on World Wide Web. pp. 1421–1432 (2013)
22. Xu, L., Yuan, Q., Wu, J., Liu, C.: Ontology-based Web Service Robustness Test Generation. In: IEEE Intern. Symp. on Web Systems Evolution. pp. 59–68 (2009)