IDLGen: Automated Code Generation for Inter-parameter Dependencies in Web APIs

Saman Barakat[1][0000−0002−7714−3742], Ana Belén Sánchez[1][0000−0003−1473−0955], and Sergio Segura[1][0000−0001−8816−6213]

SCORE Lab, I3US Institute,
Universidad de Sevilla, Seville, Spain
{salias,anabsanchez,sergiosegura}@us.es

Abstract. The generation of code templates from web API specifications is a common practice in industry. However, existing tools neglect the dependencies among input parameters (so-called inter-parameter dependencies), extremely common in practice and usually described in natural language. As a result, developers are responsible for implementing the corresponding validation logic manually, a tedious and error-prone process. In this paper, we present IDLGen, an approach for the automated generation of validation code for inter-parameter dependencies in web APIs. Specifically, we exploit the IDL4OAS extension for specifying inter-parameter dependencies as a part of OpenAPI Specification (OAS) files. To make our approach applicable in practice, we present an extension of the popular OpenAPI Generator tool ecosystem, automating the generation of Java and Python code for the management of inter-parameter dependencies in web APIs. Evaluation results show the effectiveness of the approach in accelerating the development of APIs, generating up to 9.4 times more lines of Java code than the current generator. This leads to average time savings ranging from 16 to 24 minutes when implementing API operations including between one and three dependencies, when compared to manual coding. More importantly, the generated code mitigates human errors, making web APIs significantly more reliable.

Keywords: Web APIs · Open API and Scaffolding · Code generation

1 Introduction

Web Application Programming Interfaces (APIs) enable communication between heterogeneous devices and systems over the Web. They have gained significant interest in the software industry as the *de-facto* standard for software integration. API directories such as Rapid [32] index over 40K web APIs from different domains such as shopping, finance, and social networks. Web APIs can be categorized into various types based on application designs and the communication protocols they use. Hypertext Transfer Protocol (HTTP) APIs, arguably the de-facto standard, use the HTTP protocol to interact—typically through CRUD (Create, Read, Update, and Delete) operations—with resources (e.g., a *video* in the YouTube API [40] or an *invoice* in the PayPal API [30]). HTTP APIs often

implement the principles of the REST architectural style for distributed systems, being referred to as RESTful APIs [10]. Henceforth, we will use the term web API to refer to RESTful web APIs or, more generally, to HTTP APIs.

Web APIs are commonly described using the OpenAPI Specification (OAS) format [29,12]. An OAS document describes a web API in terms of the operations supported, as well as their input parameters and the possible responses. OAS documents are heavily used nowadays for automating certain tasks in the API lifecycle. One of these applications is *scaffolding*: generating code templates for both API servers and clients from the OAS specification of the API. There are multiple tools available for code generation, including AutoRest [2], Codegen [6], NSwag [25], and OpenAPI Generator [28], among others. OpenAPI Generator is a popular open-source code generation tool ecosystem for OAS. It is developed in Java and offers over 50 generators for clients and servers in different programming languages.

Web APIs typically include inter-parameter dependencies. These are constraints that restrict the way in which two or more input parameters can be combined to form a valid call to the service [20,21]. For example, when searching for businesses in the Yelp API[1], the parameter `location` is "required if either `latitude` or `longitude` is not provided", and both parameters are "required if `location` is not provided". A recent study revealed that dependencies are extremely common and pervasive in industrial web APIs: they appear in 4 out of every 5 APIs across all application domains and types of operations [21]. However, the current version of OAS provides no support for the formal description of these types of dependencies, despite being a highly demanded feature by practitioners[2]. Instead, users are encouraged to describe them informally using natural language[3]. As a result, current scaffolding tools do not support the generation of validation code for inter-parameter dependencies, as these are not specified in OAS documents. Therefore, the validation code associated to these dependencies must be manually implemented. In the previous example, for instance, developers should write the required assertions to make sure that `latitude` and `longitude` parameters are used together when the `location` parameter is not provided in an API call, both in clients and servers. This is not only tedious, but also error-prone, making validation failures very common in practice [22].

In this paper, we present IDLGen, an approach for the automated generation of code for validating inter-parameter dependencies in web API servers. For this, we leverage IDL4OAS, an OAS extension for specifying *inter-parameter dependencies* as a part of OAS documents using the Inter-parameter Dependency Language (IDL) [17,20]. To make our approach readily applicable in practice, we present an extension of the OpenAPI Generator tool ecosystem, enabling the automated generation of Java and Python code for the validation of inter-parameter dependencies in API servers. Evaluation results show that our approach generates up to 940% more lines of code (LoC) for Java, 491% on average. This im-

[1] https://www.yelp.com/developers/documentation/v3/business_search
[2] https://github.com/OAI/OpenAPI-Specification/issues/256
[3] https://swagger.io/docs/specification/describing-parameters

provement is even more noticeable for Python, where the OpenAPI Generator generates an empty method for each API operation, whereas IDLGen generates up to 68 LoC, 37.2 on average, Additionally, the results of an empirical study with 81 participants revealed that IDLGen saves, on average, between 16 and 24 minutes per API operation (including between one and three dependencies) when compared to manual coding. More importantly, our results show that IDLGen effectively avoids human mistakes, common in practice, making web APIs significantly more reliable.

A very preliminary version of this work, restricted to code generation for Java, was presented in [3]. This paper extends our previous work in several directions, including a significantly larger evaluation with 14 industrial API operations, code generation for Java and Python, and an empirical study with 81 participants.

2 Background

This section introduces key concepts to contextualize our proposal, namely, IDL, the IDL4OAS extension, and the OpenAPI Generator tool ecosystem.

2.1 Inter-parameter dependency language (IDL)

The Inter-parameter Dependency Language (IDL) is a textual domain-specific language used to describe dependencies among input parameters in web APIs [20]. IDL was created based on the results of a study of more than 2.5K operations of 40 real-world APIs. Specifically, IDL supports expressing seven types of inter-parameter dependencies widely used in practice. As an example, Listing 1 shows a fragment of an IDL document describing the inter-parameter dependencies found in the Google Maps Places API [13]. In what follows, we briefly introduce the types of dependencies supported by IDL including references to Listing 1:

```
1  // Operation: Search for places within specified area:
2  ZeroOrOne(radius, rankby=='distance');
3  IF rankby=='distance' THEN keyword OR name OR type;
4  maxprice >= minprice;
5
6  // Operation: Query information about places:
7  AllOrNone(location, radius);
8  Or(query, type);
9  maxprice >= minprice;
10
11 // Operation: Get photo of place:
12 OnlyOne(maxheight, maxwidth);
13
14 // Operation: Automcomplete place name:
15 IF strictbounds THEN location AND radius;
```

Listing 1: IDL specification of Google Maps Places API.

– **Requires**. This type of dependency emerges when the presence of a parameter *p1* in a request requires the presence of another parameter *p2*. For example, line 3 indicates that if the parameter *rankby* of the search operation in Google Maps is set to 'distance', then at least one of the following parameters must be present: *keyword*, *name* or *type*.

- **Or**. Given a set of parameters, one or more of them must be included in the request. As an example, in the Google Maps Places API, when searching for places (line 8), both *query* and *type* parameters are optional, but at least one of them must be used.
- **OnlyOne**. Given a set of parameters, one and only one of them must be included in the request. For example, line 12 indicates that only one of the parameters *maxheight* and *maxwidth* must be used.
- **AllOrNone**. Given a set of parameters, either all of them must be included in the request, or none of them. For example, as expressed in line 7, either both *location* and *radius* are used, or none of them.
- **ZeroOrOne**. Given a set of parameters, zero or at most one must be included in the request. For example, line 2 indicates that if the parameter *radius* is used, then *rankby* cannot be set to 'distance' and vice versa.
- **Arithmetic/Relational**. Relational and arithmetic dependencies relate two or more parameters using standard relational and arithmetic operators. For example, as stated in line 4, the parameter *maxprice* must be greater than or equal to *minprice*.
- **Complex**. These dependencies are specified as a combination of the previous ones.

We refer the reader to [17,20] for a detailed description of the language, including its grammar.

2.2 IDL4OAS

Web APIs are commonly described using the OAS [29] format, arguably the industry standard. OAS documents describe web APIs in terms of the elements it comprises, namely, paths, operations, resources, request parameters, and responses. IDL4OAS [20] is an OAS extension for describing inter-parameter dependencies within OAS using the IDL language. This makes it possible to process dependencies automatically and leverage them, for example, for automated test case generation [22].

IDL4OAS supports specifying inter-parameter dependencies at the operation level. As an example, Listing 2 shows an excerpt of an OAS document in YAML format extended with IDL4OAS, corresponding to the Get Bussiness operation of the Yelp API [38]. As illustrated, the property "x-dependencies" has been added to the "GET /businesses/search" operation. This property is an array of elements, where each element—preceded by a hyphen— represents a single dependency following the syntax of IDL.

```
1  paths:
2    /businesses/search:
3      get:
4        parameters:
5          - name: location [...]
6          - name: latitude [...]
7          - name: longitude [...]
8          - name: open_now [...]
9          - name: open_at [...]
10         - name: limit [...]
11         - name: offset [...]
```

```
12          - [...]
13        [...]
14        x-dependencies:
15          - Or(location, latitude AND longitude);
16          - ZeroOrOne(open_now, open_at);
17          - offset + limit <= 1000;
18          - IF offset AND NOT limit THEN offset <= 980;
```

Listing 2: OAS document of the Get Businesses operation from the Yelp API extended with IDL4OAS

2.3 OpenAPI Generator

OpenAPI Generator is a set of tools that automatically generate API clients library, server stubs, configuration, and documentation files based on a given OAS definition of the API [28]. It is developed in Java, with over 50 generators for clients and servers that generate code for different programming languages.

OpenAPI Generator has transforming logic as well as templates for each generation of code. Built-in templates are written in Mustache [24], which is a template system with multiple implementations for different languages and technologies. The templates contain common code, independent of the specific API, and have variables that are replaced with the parsed data from the OAS file. As an example, Listing 3 (Lines 1, 18-28), shows the code generated when running OpenAPI Generator on the *Search Business* operation within the Yelp API, as shown in Listing 2. It basically consists of a method including some media type checks and returning an HTTP error (501 - "Not implemented") to let the developer know that (s)he must implement the functionality of the operation.

3 Approach: IDLGen

We propose IDLGen, an approach for the automated generation of validation code for inter-parameter dependencies in Web APIs. Given an API request, the generated code automatically checks its conformance to the inter-parameter dependencies of the API, returning an informative error in case a violation is detected. By automating this process, our approach not only saves development time but also eliminates potential bugs caused by programming mistakes, which are prevalent in the validation code of web APIs [21,23].

Figure 1 depicts an overview of the approach. IDLGen generates code from the API specification in OAS format, arguably the industry standard. This makes our approach readily applicable and language-independent. Specifically, we leverage the IDL4OAS extension for extending OAS files with a rigorous specification of inter-parameter dependencies using the IDL language. Hence, given an input OAS file enriched with IDL4OAS, we propose transforming each dependency into a fragment of executable code that checks whether the incoming API request satisfies it or not, returning an informative message and an HTTP error status code in case it is violated. We propose using code templates for the generation of code, making our approach easily customizable.

```
default ResponseEntity<BusinessesResult> getBusinesses(. . .) {
+ // Check dependency: Or(location, latitude AND longitude);
+ if(DependencyUtil.doNotSatisfyOrDependency((location != null),(latitude != null) &&
    (longitude != null))){
+    return new ResponseEntity("Dependency not satisfied: Or(location, latitude AND
     longitude);", HttpStatus.BAD_REQUEST);
+ }
+ // Check dependency: ZeroOrOne(open_now, open_at);
+ if(DependencyUtil.doNotSatisfyZeroOrOneDependency((openNow != null),(openAt != null))){
+    return new ResponseEntity("Dependency not satisfied: ZeroOrOne(open_now, open_at);",
     HttpStatus.BAD_REQUEST);
+ }
+ // Check dependency: offset + limit <= 1000;
+ if(!(!(offset != null && limit != null) || (offset+limit<=1000.0))){
+    return new ResponseEntity("Dependency not satisfied: offset + limit <= 1000;",
     HttpStatus.BAD_REQUEST);
+ }
+ // Check dependency: IF (offset AND NOT limit) THEN offset <= 980;
+ if(!(!((offset != null) && !(limit != null)) || (offset != null && offset<=980.0))){
+    return new ResponseEntity("Dependency not satisfied: IF (offset AND NOT limit) THEN
     offset <= 980;", HttpStatus.BAD_REQUEST);
+ }
 getRequest().ifPresent(request -> {
   for (MediaType mediaType: MediaType.parseMediaTypes(request.getHeader("Accept"))) {
   if (mediaType.isCompatibleWith(MediaType.valueOf("application/json"))) {
    String exampleString = ". . .";
    ApiUtil.setExampleResponse(request, "application/json", exampleString);
    break;
        }
   }
 });
 return new ResponseEntity<>(HttpStatus.NOT_IMPLEMENTED);
}
+ public static boolean doNotSatisfyOrDependency(boolean... assertions){
+   boolean result = false;
+   for (int i=0;i<assertions.length;i++){
+     result = result || assertions[i];
+     if (result)
+       return false;
+   }
+   return true;
+ }
+ public static boolean doNotSatisfyZeroOrOneDependency(boolean... assertions){
+   boolean result = true;
+   for (int i=0;i<assertions.length;i++){
+     result = result && ZeroOrOneAllOrNoneElement(i,assertions,false,true);
+     if (!result)
+       break;
+   }
+   return !result;
+ }
+ private static boolean ZeroOrOneAllOrNoneElement(int i,boolean[] allElements,
   boolean negateElement, boolean negateRemainingElements){
+   boolean element = negateElement ? allElements[i]:!allElements[i];
+   if (element)
+     return true;
+   boolean result = true;
+   for (int j=0;j<allElements.length;j++){
+     if (i!=j){
+       boolean otherElement = negateRemainingElements ? !allElements[j]:allElements[j];
+     result = result && otherElement;
+     if (!result)
+       return false;
+     }
+   }
+   return true;
+ }
```

Listing 3: Code generated by IDLGen for the Get Businesses operation (Yelp API)

To make our approach applicable in practice, we have developed IDLGen, an extension of the widely recognized OpenAPI Generator tool suite [28], complementing its functionalities to generate Java and Python code to deal with dependencies in web API servers. To achieve our objectives, we created a fork

of the OpenAPI Generator project on GitHub [15]. Within this fork, we extended the functionality of two generators, responsible for generating projects for "Spring" and "FastAPI" frameworks. Additionally, we developed a Mustache template that incorporates the necessary logic for different types of dependencies, including *Or*, *OnlyOne*, *AllOrNone*, and *ZeroOrOne*. Leveraging the extended classes, the IDL parser [16], and the Mustache template, we translated dependencies expressed using IDL4OAS [20] into conditional blocks for each assertion within an operation. If the condition specified by the dependency is not met, the code returns a bad request HTTP status code (400), accompanied by descriptive error messages.

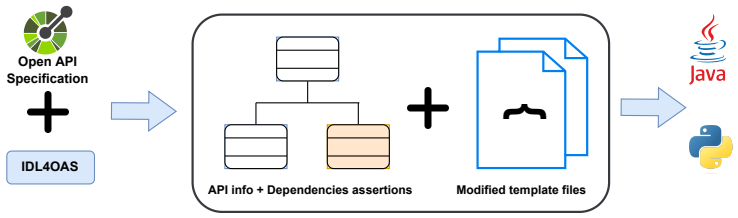

Fig. 1: IDLGen overview

As an example, Listings 3 depicts the Java server code generated by IDLGen for the Get Bussiness operation of the Yelp API. The tool receives as input the specification of the API operation, shown in Listing 2, including the description of dependencies in an IDL4OAS block. Lines of code generated by IDLGen are highlighted in green and preceded by the symbol '+' for illustrative purposes. Lines not highlighted are those generated by the original OpenAPI Generator. As illustrated, the code generated by IDLGen consists of conditional statements checking whether or not each dependency is violated (lines 2-17), plus some auxiliary methods for checking non-trivial dependencies (lines 29-61). Overall, IDLGen generates 55 LoC without counting comments, whereas the original OpenAPI Generator generates 10 LoC. This means an improvement of 450% on the amount of generated code.

4 Evaluation

We aim to answer the following research questions:

- **RQ1**: *What are the gains of using IDLGen in terms of the amount of generated code?* We aim to quantify the gains of our approach in terms of the lines of code automatically generated in comparison with standard code generators.
- **RQ2**: *What are the benefits of IDLGen in terms of development time and ratio of failures?* We aim to study the benefit of using our approach in reducing development time and faults in the validation code of inter-parameter dependencies compared to manual implementation.

4.1 Subject APIs

We used a dataset of 14 API operations from 10 real-world APIs previously used in the context of web API testing [33]. These operations represent a diverse set in terms of domains, sizes, and dependencies, including all the types of inter-parameter dependencies identified in [21] (c.f. Section 2.1). Table 1 shows the web API operations used in this study. For each operation, the table shows the name and reference of the API it belongs to, name, number of parameters, number of dependencies, and number (and percentage) of parameters involved in its dependencies (column PD(%)). For each API, we used the OAS specification file provided in [33], which includes the specification of IDL dependencies using IDL4OAS. Since the experiments were run locally, we slightly modified each OAS file changing the server URL and removing security-related configuration details, e.g. OAuth. The resulting OAS files used in our experiments are available as part of the supplementary material [4].

4.2 Experiment 1: Code generation

In this experiment, we aim to answer RQ1 by evaluating the amount of code generated by our approach in comparison to standard specification-driven tools.

Experimental setup. We used IDLGen—our extension of the OpenAPI Generator tool—to generate Java server code (Spring) and Python server code (FastAPI) for the API operations listed in Table 1. Specifically, for each operation, we generated code using the standard OAS specification files—with no information about inter-parameter dependencies—and the OAS files enriched with IDL4OAS—describing inter-parameter dependencies using IDL. Then, we computed the number of generated lines of code on each scenario, with and without dependencies. To make the results more accurate, we restricted the counting of LoC to the method implementing the API operation (and corresponding auxiliary methods), excluding imports and class definitions code, since it is common in both cases. The generated projects are available as part of the supplementary material of the paper [4].

Experimental results. Table 1 shows the result of our experiment on 14 real-world API operations. On the one hand, the original OpenAPI Generator tool—with no support for dependencies—generated exactly 10 LoC of Java for each API operation. This is because the code generated is always the same: a method template with some basic media type checks (see lines 18-28 from the example in Listing 3). Analogously, it generated an empty method in Python for each API operation. On the other hand, IDLGen—including support for dependencies–generated between 13 and 104 LoC of Java (59.1 LoC on average) and between 2 and 68 LoC of Python (37.2 on average). This means IDLGen generated between 30% and 940% more Java code (491% on average) than the popular OpenAPI Generator tool (recall that the tool generates a method with 10 LoC for all API operations). The improvement is even more noticeable in Python, where OpenAPI Generator generated an empty method for each API operation, whereas

IDLGen generated up to 68 LoC. As expected, this improvement seems to be proportional to the number of dependencies of the operation. As an example, the largest portion of code (94 LoC for Java, 68 LoC for Python) was generated for the API operation with more dependencies, YouTube Search, with 31 parameters and 15 dependencies (c.f. Table 1).

API Operation	**Parameters**	**Deps**	**PD(%)**	**Java**	**Python**
Amadeus - HotelOffers [1]	27	8	11 (41%)	**93**	**62**
Box - FoldersItems [5]	9	3	5 (56%)	**43**	**25**
DHL - FindByAddress [8]	10	1	2 (20%)	**22**	**9**
Foursquare - VenuesSearch [11]	17	3	7 (41%)	**53**	**32**
Ohsome - ElementArea [26]	11	3	7 (64%)	**86**	**59**
Ohsome - ElementAreaRatio [26]	15	4	9 (60%)	**89**	**61**
OMDb - Search [27]	9	1	3 (33%)	**47**	**29**
Travels - TripsUser [35]	6	1	2 (33%)	**13**	**2**
Tumblr - BlogLikes [36]	5	1	3 (60%)	**37**	**21**
Yelp - BusinessesSearch [38]	14	4	7 (50%)	**55**	**34**
Yelp - TransactionsSearch [38]	3	1	3 (100%)	**22**	**9**
YouTube - Comments [40]	6	3	4 (67%)	**77**	**52**
YouTube - CommentThreads [40]	11	6	9 (82%)	**86**	**58**
YouTube - Search [40]	31	15	26 (84%)	**104**	**68**
Mean				**59.1**	**37.2**

Table 1: Java and Python #LoC generated by IDLGen. Deps = Dependencies. PD(%) = Number and percentage of parameters involved in the dependencies.

To check the correctness of the generated code we performed a sanity check as follows. We used the open-source framework RESTest [22,33] for automatically generating test cases for the API operations under test. Specifically, for each subject API operation, we used RESTest to generate 1000 valid test cases—satisfying all the dependencies described in the OAS file—and 1000 invalid test cases—API requests violating one or more inter-parameter dependencies. Then, we ran the test cases against the validation code generated and confirmed that they were correctly processed. Specifically, valid calls were handled returning a 200 HTTP status code[4], whereas invalid calls were correctly identified, returning proper 400 HTTP status codes and descriptive error messages. This supports the validity of the generated code. The generated test cases are also available as a part of the supplementary material [4].

Response to RQ1

IDLGen generates between 1.3 and 9.4 times more Java code than a standard code generator. The improvement is even more noticeable in Python, where OpenAPI Generator generates no code (empty method), and IDLGen generates between 2 and 68 LoC.

[4] We configured the tool such that the generated code returns a 200 status code (rather than 501 - "Not implemented") when all dependencies are satisfied.

4.3 Experiment 2: Implementation time and faults

In this experiment, we aim to address RQ2 by evaluating the time required by developers to implement the validation code for checking inter-parameter dependencies, as well as the failure rate of the produced code.

Experimental setup. We conducted an experiment with 81 second-year students of the Software Engineering Bachelor Degree at *University of Seville*. Specifically, the experiment was conducted in the course on Software Architecture and Integration. The experiment took place at the end of the course when students had gained experience consuming and implementing REST APIs using the Spring framework. Participants were tasked with implementing inter-parameter dependencies for two API operations, keeping a record of the invested time. Then, the resulting code was analyzed by the authors, who ran a thorough test suite on each participant project to identify failures.

The 81 participants were divided into four groups, who attended a session of 1 hour and 50 minutes each. The authors of the paper conducted each session. At the beginning of each session, the instructors briefly introduced inter-parameter dependencies and IDL. Then, participants were asked to download two template Java projects (Spring Boot) from GitHub, where they had to implement the inter-parameter dependencies of two API operations: a mock version of the search business transactions operation in the Yelp Fusion API, and a mock version of the folder listing operation in the Box API. The Yelp Fusion API project featured a single dependency, whereas the Box API project had three dependencies, depicted in IDL format in Listings 4 and 5, respectively. Participants were asked to implement and test their code and then submit it through the university virtual platform, indicating the starting and ending time for each project as a part of the submission. In a later step, each submission was thoroughly analyzed by the authors, running a test suite on each participant project. These test suites were carefully crafted by the authors trying to cover the most relevant input combinations and including both valid and invalid API requests. As a sanity check, we used IDLGen for generating code for both API operations and confirmed that the generated code passed both test suites.

Among the 162 submissions received (two projects per participant), five of them were empty and were discarded, resulting in 77 projects for the Yelp API, and 80 for the Box API. All submissions, duly anonymized, are included in the supplementary material [4], as well as the test suites used.

```
- Or(location, latitude AND longitude);
```

Listing 4: IDL specification of the Search Business Transactions operation (Yelp API)

```
- IF marker THEN usemarker == true;
- IF (usemarker == true AND folder_id == '0') THEN NOT sort;
- ZeroOrOne(usemarker == true, offset);
```

Listing 5: IDL specification of the Folder Listing operation (Box API)

Experimental results. As summarized in Table 2, participants took between 2 and 42 minutes (15.6 minutes on overage) to implement the validation code

for the Yelp API operation (one dependency), and between 8 and 62 minutes (24.3 minutes on average) for the Box API operation (three dependencies). In sharp contrast, IDLGen took less than one second to automatically generate the validation code of both API operations. In terms of faults, more than half

		Time			
API operation	**Projects**	**Min**	**Max**	**Avg**	**Failures (%)**
Box - FoldersItems	80	**8**	**62**	**24.3**	**92.5**
Yelp - TransactionsSearch	77	**2**	**42**	**15.6**	**51.9**

Table 2: Average implementation time (min) and percentage of projects with failures

(51.9%) of the Yelp API projects did not pass one or more of the test cases created by the authors. On the other hand, a significantly higher percentage (92.5%) of the Box API projects failed at least one test case. Upon analyzing the test results, it was observed that 44.3% of Yelp API projects failed when making a valid request that included the *location* parameter. It appears that the participants either misunderstood the logic behind the *Or* dependency or did not adequately test their code for valid requests. This is because when the *location* dependency is passed, the request should be valid regardless of the *latitude* and *longitude* values. In the case of the Box API, we found that a significant portion of the failures (78%) were due to a null value not properly checked (dependency *(IF marker THEN usemarker==true;)*) throwing a Null Pointer Exception. These results support previous findings revealing that many of the failures revealed in Web APIs are due to faults in the input validation logic [23]. Again, this is in sharp contrast with our approach, where valid code is automatically generated, discarding potential human mistakes.

As expected, the time and percentage of failures observed in manual coding seem to increase with the number and complexity of dependencies. This suggests that the benefits of IDLGen would be significantly more noticeable in highly-constrained API operations, e.g., 25 out of the 31 input parameters of the search operation in the YouTube API are involved in at least one dependency [20].

The results also revealed the potential of IDLGen to improving code maintainability. For example, we observed that some of the participants tried to check all the dependencies in a single long if statement, making the code error-prone and hard to understand. In contrast, code generated by IDLGen addresses each dependency independently, showing descriptive error messages for each of them.

Response to RQ2

IDLGen saves, on average, between 16 and 24 minutes in API operations with between one and three dependencies. More importantly, the generated code mitigates human error, making Web APIs substantially more reliable. Savings are expected to be more noticeable as the number and complexity of dependencies increases.

5 Related work

Several papers have addressed the problem of automated code generation of web APIs. Ed-douibi et al. presented an approach called EMF-REST that takes Eclipse Modeling Framework (EMF) data models as input to generate REST APIs [9]. Gómez et al. introduced a proposal called CRUDyLeaf based on Domain-Specific Languages (DSL). The tool takes an entity with CRUD operations (Create, Read, Update, Delete) to generate Spring Boot REST APIs [14].

Queirós presented Kaang, an automatic generator of REST Web applications. Its goal is to reduce the impact of creating a REST service by automating all its workflow, such as creating file structuring, code generation, dependencies management, etc. [31]. This tool is based on Yeoman [39], an open-source, client-side development stack consisting of tools and frameworks intended to help developers build web applications.

Deljouyi et al. introduced MDD4REST [7], a model-driven methodology that uses Domain-Driven Design (DDD) to produce a rich domain model for web services. Also, it designs REST web services using modeling languages and supports automatic code generation through a transformation of models. The authors in [37] used UML class diagrams to model a set of NoSQL database collections, and then automate the generation of common database access functions and the wrapping of these functions within a set of REST APIs.

Li et al. proposed a Navigation-First Design approach to make a REST API navigable before implementing any service actions [19]. This approach is based on REST Chart [18], which is a model and language to design and describe REST APIs without violating the REST constraints. Rossi [34] proposed a model-driven approach to develop a REST API. First, they used modeling of the API with specific profiles. Then, a model transformation exploited REST API Modeling Language (RAML) as an intermediate notation that could be used to produce documentation and code for various languages automatically.

In contrast to related papers, this is the first work addressing code generation for inter-parameter dependencies in web APIs. Evaluation results show that this leads to important gains in terms of productivity and reliability. Our work is based on exploiting an enriched version of the OAS specification— arguably the de-facto standard in the industry—making it easy to integrate our approach into related tools.

6 Threats to validity

In this section, we discuss the potential validity threats that may have influenced our work and how these were mitigated.

Internal validity. *Are there factors that might affect the results of our evaluation?* A potential threat is the possibility of implementation errors within the IDLGen extension, which could compromise the accuracy and reliability of the generated code. To mitigate this threat, we conducted extensive testing and validation throughout the development process. More importantly, we ran 28K

automatically generated test cases (2K test cases per API operation) on the code generated for the 14 subject API operations revealing no failures.

The validity of the experiment with people may be compromised due to the lack of experience of the students who participated in the study. To mitigate this threat, we conducted the experiment at the end of the course, when students had gained extensive experience in consuming and implementing REST APIs. We also simplified the examples by excluding parameters unrelated to dependencies, which allowed students to focus exclusively on the implementation process. In addition, we provided thorough explanations of the dependencies for both examples to ensure that students understood the tasks effectively. Overall, the results show that implementing the validation code for inter-parameter dependencies is time-consuming and error-prone, supporting the value of IDLGen to generate error-free code in a matter of seconds.

External validity. *To what extent can we generalize the findings of our investigation?* The generalizability of our findings may be limited due to the specific set of API operations evaluated. To mitigate this threat, we carefully selected a diverse sample of 14 operations from 10 industrial APIs with millions of users worldwide. Similarly, we focused on a specific and highly popular code generator, OpenAPI Generator, and therefore our results may not be generalized further. To the best of our knowledge, however, none of the state-of-the-art generators for web APIs supports the generation of validation code for inter-parameter dependencies, and therefore the gain reported in our paper should be analogous when considering similar tools.

7 Conclusions and future work

This paper presents IDLGen, an approach for the automated generation of validation code for inter-parameter dependencies in web APIs. Specifically, our approach leverages the IDL4OAS extension for describing dependencies as a part of OAS files. The generated code can automatically detect whether or not incoming API calls satisfy the dependencies among input parameters, returning informative errors in case they are violated. To implement our approach, we extended the well-known OpenAPI Generator tool ecosystem to automate the generation of Java and Python code for inter-parameter dependencies in web APIs. The evaluation results show that IDLGen generates up to 9.4 times more LoC for Java servers than a state-of-the-art code generator (5 times more LoC on average), with similarly noticeable savings in Python. The results of an empirical study with 81 participants revealed that IDLGen saves an average of between 16 minutes (one dependency) and 24 minutes (three dependencies) per API operation. More importantly, the code generated minimizes the possibility of making mistakes, making APIs significantly more robust and reliable.

Several challenges remain for future work. We plan to address the automated generation of documentation for inter-parameter dependencies. Also, we aim to obtain feedback from the core team of the OpenAPI Generator project for eventually merging our approach into the official tool ecosystem.

Acknowledgements

This work has been partially supported by grants PID2021-126227NB-C22 and TED2021-131023B-C21, funded by MCIN/AEI/10.13039/501100011033 and by European Union "NextGenerationEU"/PRTR». Ana B. Sánchez was supported by the VI Plan Propio de Investigación y Transferencia of Universidad de Sevilla 2021 [VI PPIT-US].

References

1. Amadeus Hotel Search API, `https://developers.amadeus.com/self-service/category/hotel/api-doc/hotel-search/api-reference`, accessed: July 2023
2. AutoRest, `https://github.com/Azure/autorest`, accessed: June 2023
3. Barakat, S., Roque, E.B., Sánchez, A.B., Segura, S.: Specification-Driven Code Generation for Inter-parameter Dependencies in Web APIs. In: Troya, J., Mirandola, R., Navarro, E., Delgado, A., Segura, S., Ortiz, G., Pautasso, C., Zirpins, C., Fernández, P., Ruiz-Cortés, A. (eds.) Service-Oriented Computing – ICSOC 2022 Workshops. pp. 261–273. Springer Nature Switzerland, Cham (2023)
4. Barakat, S., Sánchez, A.B., Segura, S.: [Supplementary material] IDLGen: Automated Code Generation for Inter-parameter Dependencies in Web APIs (July 2023), `https://doi.org/10.5281/zenodo.8138633`
5. Box API, `https://developer.box.com/reference/`, accessed: July 2023
6. Swagger Codegen, `https://swagger.io/tools/swagger-codegen/`, accessed: June 2023
7. Deljouyi, A., Ramsin, R.: MDD4REST: Model-Driven Methodology for Developing RESTful Web Services. In: MODELSWARD. pp. 93–104. Scitepress (2022)
8. DHL Location Finder API, `https://developer.dhl.com/api-reference/location-finder`, accessed: July 2023
9. Ed-Douibi, H., Izquierdo, J.L.C., Gómez, A., Tisi, M., Cabot, J.: EMF-REST: Generation of RESTful APIs from Models. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing. vol. 04-08-April-2016, pp. 1446–1453. Association for Computing Machinery (2016)
10. Fielding, R.T.: REST: Architectural Styles and the Design of Network-Based Software Architectures. Doctoral dissertation, University of California (2000)
11. Foursquare Search for Venues API, `https://developer.foursquare.com/reference/v2-venues-search`, accessed: July 2023
12. Gamez-Diaz, A., Fernandez, P., Ruiz-Cortes, A.: Automating SLA-Driven API Development with SLA4OAI. In: Yangui, S., Bouassida Rodriguez, I., Drira, K., Tari, Z. (eds.) Service-Oriented Computing. pp. 20–35. Springer International Publishing, Cham (2019)
13. Google Maps API, `https://developers.google.com/maps/documentation/places/web-service/search`, accessed: July 2023
14. Gómez, O.S., Rosero, R.H., Cortés-Verdín, K.: CRUDyLeaf: A DSL for Generating Spring Boot REST APIs from Entity CRUD Operations. Cybernetics and Information Technologies **20**(3), 3–14 (2020)
15. IDLGen. https://github.com/ssegura/openapi-generator/tree/IDLGen-extension, accessed: July 2023
16. IDL Parser. `https://github.com/isa-group/IDL-mvn-dep`, accessed: July 2023

17. Inter-parameter Dependency Language (IDL), https://github.com/isa-group/IDL, accessed: July 2023
18. Li, L., Chou, W.: Design and Describe REST API without Violating REST: A Petri Net Based Approach. In: 2011 IEEE International Conference on Web Services. pp. 508–515 (2011)
19. Li, L., Tang, T., Chou, W.: Automated Creation of Navigable REST Services Based on REST Chart. Journal of Advanced Management Science pp. 385–392 (2016)
20. Martin-Lopez, A., Segura, S., Muller, C., Ruiz-Cortes, A.: Specification and Automated Analysis of Inter-Parameter Dependencies in Web APIs. IEEE Transactions on Services Computing pp. 1–14 (2021)
21. Martin-Lopez, A., Segura, S., Ruiz-Cortés, A.: A Catalogue of Inter-parameter Dependencies in RESTful Web APIs. In: Yangui, S., Bouassida Rodriguez, I., Drira, K., Tari, Z. (eds.) Service-Oriented Computing. pp. 399–414. Springer International Publishing, Cham (2019)
22. Martin-Lopez, A., Segura, S., Ruiz-Cortés, A.: RESTest: Black-Box Constraint-Based Testing of RESTful Web APIs. In: Kafeza, E., Benatallah, B., Martinelli, F., Hacid, H., Bouguettaya, A., Motahari, H. (eds.) Service-Oriented Computing. pp. 459–475. Springer International Publishing, Cham (2020)
23. Martin-Lopez, A., Segura, S., Ruiz-Cortés, A.: Online Testing of RESTful APIs: Promises and Challenges. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 408–420. ESEC/FSE 2022, Association for Computing Machinery, New York, NY, USA (2022)
24. Logic-less templates, https://mustache.github.io/, accessed: July 2023
25. NSwag toolchain, https://github.com/RicoSuter/NSwag, accessed: June 2023
26. Ohsome API, https://docs.ohsome.org/ohsome-api/v1/, accessed: July 2023
27. OMDb API, http://www.omdbapi.com/, accessed: July 2023
28. OpenAPI Generator, https://openapi-generator.tech/, accessed: July 2023
29. OpenAPI Specification, https://www.openapis.org/, accessed: July 2023
30. PayPal Invoicing API, https://developer.paypal.com/docs/api/invoicing/v1/#invoices, accessed: July 2023
31. Queirós, R.: Kaang: A RESTful API Generator for the Modern Web. In: 7th Symposium on Languages, Applications and Technologies SLATE 2018. vol. 62, pp. 1:1–1:15. Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2018)
32. RapidAPI Hub, https://rapidapi.com/hub, accessed: March 2022
33. RESTest: Automated Black-Box Testing of RESTful Web APIs, https://github.com/isa-group/RESTest, accessed: July 2023
34. Rossi, D.: UML-based Model-Driven REST API Development. In: WEBIST 2016 - Proceedings of the 12th International Conference on Web Information Systems and Technologies. pp. 194–201 (2016)
35. Travel API, https://github.com/isa-group/RESTest/tree/master/src/test/resources/Travel, accessed: July 2023
36. Tumblr API, https://www.tumblr.com/docs/en/api, accessed: July 2023
37. Wang, B., Rosenberg, D., Boehm, B.W.: Rapid Realization of Executable Domain Models via Automatic Code Generation. In: 2017 IEEE 28th Annual Software Technology Conference (STC). pp. 1–6 (2017)
38. Yelp API, https://docs.developer.yelp.com/reference, accessed: July 2023
39. Yeoman, https://yeoman.io/, accessed: July 2023
40. YouTube Data API, https://developers.google.com/youtube/v3/docs, accessed: July 2023