HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

# swaggertest: Property-based testing framework for Open APIs

Bachelor's Thesis

by

Carlo Schackow

born in
Neuss

submitted to

Professorship for Computer Networks
Prof. Dr. Martin Mauve
Heinrich-Heine-University Düsseldorf

July 2018

Supervisor:
Christian Meter, M. Sc.

# Abstract

API development is an error-prone, complex task. Yet, with the growing market for mobile apps and micro-services, building and maintaining stable REST APIs is becoming a vital part of many software operations. The *OpenAPI Initiative* (OAI) aims at standardizing the description of REST APIs by way of their *OpenAPI Specification*, a standardized specification used to describe REST APIs, as well as a collection of tools designed for working with the specification. This thesis seeks to implement an automatic testing tool for REST APIs described with the OpenAPI specification, providing an asset to be used continuously during development and subsequently on live REST APIs. A command line tool, *SwaggerTest*, was implemented in the programming language Java. The use of generative testing ensures minimal workload for the user and extensive tests for the API. The tool was then tested on several live REST APIs, as well as *D-BAS*, a webservice currently in development by the department of computer networks at Heinrich Heine University. All of the tested APIs failed multiple tests, revealing flaws within the implementation and documentation. This shows the value of SwaggerTest for the use case of maintaining a live REST API. SwaggerTest's practicality when developing a new API from start to finish remains to be tested.

# Acknowledgments

A lot of people supported me during my work on this thesis to whom I wish to express my gratitude. Thanks go out to Julia and Tinki, who proofread my work.

Thanks also to Christian Meter, who brought up the concept for this thesis, and advised me through the whole process.

# Contents

**1 Motivation 1**

1.1 Context . . . 1

1.2 Scope . . . 2

**2 REST APIs and the OAS 3.0 Document 3**

2.1 REST APIs . . . 3

2.2 The OAS 3.0 Document . . . 4

2.2.1 Components . . . 4

2.2.2 Paths . . . 6

2.2.3 Security . . . 6

**3 Why Generative Testing? 7**

3.1 Unit Testing . . . 7

3.2 Generative Testing . . . 8

**4 Development Process 9**

4.1 Parser . . . 9

4.2 Test Generator and Tester . . . 10

4.3 Adapting SwaggerTest to the Live Tests . . . 11

**5 Program Structure and Features 13**

**6 Testing Live APIs 15**

6.1 Adyen Checkout Service . . . 16

6.2 Amadeus Travel Innovation Sandbox . . . 17

6.3 GeoMark Web Service . . . 20

6.4 BC Laws . . . 22

6.5 D-BAS . . . 22

6.6 Adafruit IO . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
6.7 Giphy API . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
6.8 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

**7 Conclusion 31**
7.1 Accomplishments . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
7.2 Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
7.3 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

**Bibliography 35**

# Chapter 1

# Motivation

## 1.1 Context

The OpenAPI Initative was formed in 2015, by Google, IBM, Microsoft and SmartBear, among others. The group adopted the Swagger 2.0 specification for REST APIs and set out to standardize API design and documentation practices. Today, it is a widely used open source REST API development framework, providing tools for documentation, code generation and testing. Being open source, however, many of these are not updated regularly. Specifically, most tools still only support the Swagger 2.0 specification and have not yet integrated the *OpenAPI Specification 3.0* (OAS 3.0), released in March 2017. *D-BAS*, a Dialog-Based Online Argumentation System by the department of computer networks at Heinrich-Heine-Universität [KMB+18] operates using a REST API which is documented in OAS 3.0. To facilitate further development of the project, a new testing tool was required.

## 1.2 Scope

The testing tool, *SwaggerTest*, was to be a simple, one-call command-line tool, to be easily embedded in the *Continuous Integration* (CI) of the project. Any additional options where to be submitted via a settings file, keeping the program call simple. A drawback of the *Swagger-Hub Integrations* [int], the testing suite developed by the Swagger team, is that they require the manual definition of tests, specifying input for routes. To improve upon this, SwaggerTest was conceptualized to use generative testing, reading out all necessary information from the OAS 3.0 file of the API and generating all tests and all input for the tests automatically, based on the definition of the API. SwaggerTest was then to be used experimentally on live APIs which implement the OAS 3.0, with the intent of uncovering flaws and inconsistencies in their implementation or documentation. The test results were to be evaluated to determine the usefulness of SwaggerTest for D-BAS specifically, as well as the OpenAPI development community in general.

# Chapter 2

# REST APIs and the OAS 3.0 Document

This shall serve as a quick introduction to the functionality of REST APIs and the OpenAPI Specification 3.0, as used in this thesis.

## 2.1 REST APIs

A *Representational State Transfer* (REST) *Application Programming Interface* (API) is a web service designed to break down large transactions into smaller ones, constisting of one request and one response. REST APIs are designed to be stateless. This means that with every call the client provides all the information necessary to process that call, and the server is not required to store any context data between calls. Additionally, the server encourages the client to avoid unneccessary calls to the API by enabling caching of received data on the client side, e.g. via a “valid-until” flag. This not only reduces the traffic load, but also keeps the logic on the server side as simple as possible, by shifting the burden of storing the state of the interaction to the client. REST APIs, unlike other concepts, such as the *Simple Object Access Protocol* (SOAP), are not bound to *Extensible Markup Language* (XML), but return a wide variety of media types, including XML, *JavaScript Object Notation* (JSON), or plain text.

## 2.2 The OAS 3.0 Document

This is intended as a short overview of the OAS 3.0 document. For an in-depth introduction, please refer to the official documentation [oas]. The OAS 3.0 is intended as a standardized format for the description and documentation of REST APIs. Valid data types for this document are JSON and *Yet Another Markup Language* (YAML). The root document object consists of up to 8 sub fields:

*Openapi*, the version number of the document
*Info*, metadata about the API
*Servers*, the servers the API can be contacted on
*Paths*, the paths and operations of the API
*Components*, definitions of objects used throught the document
*Security*, the security definitions
*Tags*, used to categorize the API
*ExternalDocs*, external Documentation about the API

The *Components*, *Paths* and *Security* fields shall be examined further here, as they dircetly pertain to the functionality of SwaggerTest.

### 2.2.1 Components

*Components* contain the definition of objects referenced elsewhere in the document. Any referenceable object can be defined where it is used or in the Components and then referenced multpile times at the appropriate locations in the rest of the document, saving space and ensuring consistency across the document. The referenceable objects relevant to SwaggerTest are:

**Schemas** *Schemas* are used to desribe the specific structure of a parameter or an object of a specific media type. The Schema definition largely parallels that of *JSON Schema* [jso], defining the schema type, its sub schemas and various properties.

**Responses** The expected *Responses* consist of a map, mapping a HTTP response code to a media type, each of which holds a *Schema* describing its structure.

**Parameters** A *Parameter* is uniquely identified by its name and location. Parameters have four possible locations:
*path*, where the parameter is part of the path name of the resource,
*query*, where the parameter is appended to the call as a query string,
*header*, where the parameter is submitted via the HTTP headers,
*cookie*, where the parameter is submitted via the special HTTP cookie headers.
Each parameter holds a *Schema* which describes the values it holds and how they are to be encoded into the call.

**Request Bodies** A *Request Body* describes the actual content of the HTTP request. The request body can be described in multiple media types, each holding a *Schema* describing its structure.

**Security Schemes** A *Security Scheme* describes a single instant of a security measure implemented on the API, which has to be met to qualify any call it pertains to. In general this means submitting the agreed upon security token in the appropriate place with each call. Possible types are *HTTP*, *APIkey*, *OAuth2*, and *OpenID Connect Discovery*. It is referenced via the *security* field on the API-, path-, and operation-level.

### 2.2.2 Paths

The *Paths* field of the document contains a map, mapping path names to the path items. Each path is tied to one resource or one set of resources the API provides access to. A path item may define additional *Servers*, from which it may be accessed, a list of *Parameters* used to modify the call, and up to 8 HTTP operations, one for each of the HTTP operations:

- GET, return the specified resource
- PUT, store the resource supplied by the client
- POST, submit the resource to the server for processing
- DELETE, delete the specified resource
- OPTIONS, return the allowed methods for this resource
- HEAD, return relevant meta-information about the resource
- PATCH, update the specified resource
- TRACE, return the call as received, enabling the client to detect changes made by transitional servers

Each operation defines further *Servers*, *Security*, the *Request Body*, expected *Response* types, and additional *Parameters* to apply to the call.

### 2.2.3 Security

The *Security Requirements* field describes which of the *Security Schemes* apply to the given resource. Referencing them like this, and not defining them directly on each method, renders them reusable on multiple paths and methods.

# Chapter 3

# Why Generative Testing?

This section is to serve as an introduction to the concept of *Generative Testing* and an explanation for its use in this thesis.

## 3.1 Unit Testing

In traditional *Unit Testing*, or example based testing, the user defines preconditions for a function, executes the function and then verifies preconceived assumptions about the function. This has the advantage of being a repeatable process, in which deterministic functions always pass the same tests. But it also requires the user to manually write these tests. For large datasets this can quickly become a nuisance. More so, the “edge cases”, often counterintuitive combinations of input, may go completely untested, because they are never conceived of by the programmer. Apart from the inefficiency, writing exhaustive test cases manually is often impossible.

## 3.2 Generative Testing

A striking difference to example based testing is the generation of the tests. They are defined not as concrete values, but as a logical *property*, or in more complex cases, groups of *properties*. These properties can be seen as a set of rules that define the input and the behavior of the test subject. Input values for each test case are then derived from these properties and the properties are expected to hold true. A result of this approach is the possibility of *shrinking*. When a property is falsified, the input for the failing test case is shortened, or "shrunk", while still adhering to the bounds of the property and the test is repeated. Eventually, a minimized example that still falsifies the property will be returned. This may not be the overall minimal falsifying value, but will often be shorter than the first falsifying value. A shorter value in turn helps the user better understand the input and why it produced the fail state. SwaggerTest is built to accept every valid OAS 3.0 document and test against the corresponding APIs. Writing example based tests for all the possible APIs would be impossible. Input and output of a REST API can take varying forms and consist of a multitude of media types. Accounting for all of them is simply not feasible. How the OAS 3.0 document is defined, however, allows extraction of some basic properties for each method of the API. The relationship of parameters and request body to the expected responses can be expressed as properties, e.g., "With input of this form the API is expected to return this kind of response.". Unexpected responses and error messages from the API can then induce a fail state, causing the input to be shrunk to find minimal test cases. In conclusion, generative testing works best for both possible use cases of SwaggerTest:
Live APIs, where fail states can be considered an exception, undergo a large number of tests with varying and exhaustive input, a lot of which may never appear in the regular use of the API, uncovering previously unknown errors.
In the case of APIs in development, where fail states are probably the norm, the developer will receive minimized input causing failure in the API, enabling them to efficiently debug the API.

# Chapter 4

# Development Process

The implementation of SwaggerTest occured in three distinct phases, first the development of the parser for the OAS 3.0 document, secondly the development of the test generator and tester. Third, the program was adapted for use with different APIs. While originally the test generator and tester were planned to be developed seperately, they soon revealed themselves as much to interrelated to be effectively implemented in isolation. SwaggerTest was written in the programming language Java.

## 4.1 Parser

The implementation of the parser for the OAS 3.0 document required an in-depth introduction to the OAS 3.0. As such, it lasted about 4 weeks, which was longer than anticipated, but it fascilitated much of the planning for the rest of SwaggerTest. After trying out the *FASTERXML jackson library*, originally chosen for its ability to handle the mime types YAML as well as tradtional JSON, it was found to feature a more ground level parser for YAML, too basic for this endeavour. After researching other available libraries, the *snakeyaml* YAML library was ultimately settled on. This library required only the definition of the Java domain objects and the submission of some top level type definitions for the document, leaving the actual parser quite short. The domain objects were modelled to mirror the structure of the document as closely as possible.

A major roadblock in development turned out to be the *Callbacks* object on the document, used to define asynchronous requests the API sends to specified addresses in response to certain actions performed on it. This element is defined in the *Components* section of the document as a map, mapping strings to *Callback* objects. The Callback object itself contains only a map, mapping strings to instances of *Path Item*. This meant the Callback could not be directly adapted as a Java object, as that requires all properties to be named. Coding the entire Callbacks object as a map, mapping strings to maps, which in turn map strings to *Path Item* objects, left the parser unable to infer the content of the inner map. This caused the Path Item to be rendered as a structure of unspecified maps, instead of the desired Java object. Eventually, the `Constructor` class of the snakeyaml library was extended and special behaviour was implemented to manually parse the Callbacks object. The definition of *Examples* could also be solved through this method: The OAS 3.0 allows for examples of parameters and schemata to be defined as either a string or a YAML or JSON representation of the parameter or schema. Java only allows properties to be of one specific type, so examples could not be represented as Java objects this way. Again, the solution was to override the behaviour in the `Constructor` class and to perform a manual check on the type of the example, and then fill in the corresponding property, either a map or a string, in the Java object.

## 4.2 Test Generator and Tester

The original plan was to program the test generator and the tester one after the other, but this turned out to be impractical. During development of the parser, the OAS 3.0 document could always be used as a reference. For the test generator, tests were being produced without specific knowledge of how they would finally be applied. So the plan was adapted to first implement a basic test generator and tester, and gradually and simultaneously add in more features in the generator and tester, as development goes on. After an initial attempt at building the tests without using a generative testing framework, SwaggerTest was reworked to integrate the *QuickTheories* java library for generative testing. This library allows for granular generation of test values as well as automatic shrinking, so it was anticipated to be a valuable resource for testing.

A problem with the library was soon found within the shrinking mechanism, as the first example it produces is an empty string. Path parameters, for example, do not allow empty values, so the test would fail immediately and output the test case with the empty value as the minimal found value, without trying longer values to see if they, too, failed. This completely annulled the testing, by producing only a single test which is bound to fail. The solution was to add a predicate to the string generator for path parameters and non-nullable string schemas, which states that the produced string is non-empty. The library would then only consider non-empty strings for testing.

The implementation of the security schemes was a much larger task than anticipated. It required the introduction of a settings file, in which the user could specify how to handle logins for the API, and set specific security tokens directly. The login function required writing another client and parsing response bodies in search of the security tokens. In the end, only the two most common security methods, APIKey and HTTP were implemented.

## 4.3 Adapting SwaggerTest to the Live Tests

During development of SwaggerTest, the program had only ever been tested on D-BAS, the API provided by the chair of computer networks. All of this testing was done against a dockerized instance of the API on a local disk. When testing on live APIs over the internet, some problems arose. Most of them could be fixed by manually checking the existence of objects in the documentation previously assumed to be present by SwaggerTest. The type of a schema, for example, while technically optional, was assumed to always be present. Now, if it is empty, the schema is assumed to be of the type *object*. Another problem were extendable objects. Many of the objects within the OAS 3.0 can be extended with custom properties, beginning with “x-”, e.g. “x-company-name”. While these properties generally do not play into the functionality of SwaggerTest, they had nonetheless to be accounted for, as the parser would otherwise fail upon encountering them. An interface, `Extendable` was introduced, and the appropriate behaviour in the snakeyaml `Constructor` class was overridden, detecting these custom properties and storing them in a Map on each extendable object. This enables future developers to easily access their desired properties by name.

A problem specific to testing on live APIs over the internet turned out to be the number of tests run. The maximum number of skrinking cycles in QuickTheories is 100,000, and for complex methods with 10 parameters, SwaggerTest would send around 70,000 requests to the API. Not only did this result in much too extensive test lengths, there was also the fear that the tests would be perceived as a *Denial-of-service* (DoS) attack, an attack on a server where the goal is to incapacitate the server with an inordinate amount of requests. The large number of tests would also cause certain defined methods to fail testing, as the "429: Too Many Requests" code is generally not defined as an expected response to a given method. The solution was to add more functionality to the settings file. An upper limit for the number of tests was introduced, as well as stop- and pause-signals, where the user could specify HTTP response codes to either pause or stop testing for the specific branch.

# Chapter 5

# Program Structure and Features

This section will briefly explain the program structure and its features.

SwaggerTest can be started from the console. The first argument has to be the path to a valid OAS 3.0 YAML document. Alternatively, the program is set up so a JSON parser can be implemented. The optional second argument, if present, has to be the path to a valid settings YAML file. Both YAML files are parsed and the corresponding `APISpecification` and `APISettings` Java objects are created. Next, the servers are tested for availability. If specified in the settings, the tests are either run on the first, or all active servers. The test generation then occurs in two phases. First, test build options are generated from the parsed OAS 3.0 document for each method on each path , applying all the relevant path information, parameters and the request body, as well as the expected responses from the API. The security schemes for each test build option are then applied, using the APISettings. If specified, logins are performed and security headers added. Secondly, the QuickTheories library is used to generate the specific test cases containing concrete values for each parameter and the request body. If specified in the settings, the number of tests will be limited, otherwise up to 100,000 tests will be generated per method. Using QuickTheories to generate the whole test case, as opposed to only single values, enables the tester to access every parameter in each test case, which allows for appropriate shrinking. The tester then creates a HTTP request from each test case, adding the parameters and security headers, and performs it on the API and evaluates the response. If the response code matches one of the expected responses, the body of the response is tested against the schema in the documentation. *Body testers* for the mime type “application/json” and a default body tester were implemented to search the response body for any missing expected properties, or unexpected properties.

Additionally, body testers for any mime type can be implemented and registered to extend this capability and cover any special case that may occur. The possible errors are:

- No servers: No servers were listed in the documentation
- Unreachable servers: None of the listed servers could be reached.
- Unknown response: returned HTTP code is not in the expected responses
- Unknown Media type: returned media type is not in the expected responses
- No media type: the document is missing any media type for this response
- Body tester: error in the testing of the response body
- Server error: a server error occured
- Client error: a client error occured
- Content unparseable: The response body could not be read
- Unknown error: An exception was thrown while executing the code

Listing 5.1: Definition of possible errors

When an error occurs, the QuickTheories tester is triggered to keep testing and shrink the values, resulting in a minimized value that still produces a fail state. All the test results are then sorted by HTTP response code and output to the console. Then the tests for the next test build options are generated and run, until all methods have been tested.

# Chapter 6

# Testing Live APIs

SwaggerTest was tested on several public REST APIs. The D-BAS specification was the focus of testing and development, the other APIs were chosen based on the size and completeness of their respective specifications, as many other available specifications described small APIs or were poorly defined. The specification YAML files for the APIs were taken from APIs.guru [api]. The specification for Adafruit IO, Amadeus Travel Innovation Sandbox, and Giphy API were converted using the Mermade Swagger 2.0 to OpenAPI 3.0.0 converter [mer]. For each entry, the API itself will be briefly described and any noteworty characteristics explained. Then any failing test results will be presented. This may include details of the error messages, the smallest found falsifying value, sample response bodies, and parts of the definitions from the appropriate specification. Afterwards, their merit for the improvement of either the API or of the API's documentation will be discussed. All test results presented in this chapter were trimmed and formatted for readability. The full test results, along with the API documentation files can be found on the accompanying CD.

| Name | Message |
|---|---|
| No Media Type | The documentation contains no mediatype for the expected response. |
| Unexpected Media Type | The API returned a mediatype not listed in the expected responses. |
| Unexpected Response | Response X did not fit the expected responses and no default was given. |
| Invalid JSON | The response JSON structure could not be read, may be invalid. |
| Unknown Property | Property X has unknown property Y. |
| Missing Property | Required property X not found. |
| Client Error | The following client error occurred: HTTP code : message. |
| Server Error | The following server error occurred: HTTP code : message. |

Table 6.1: All received errors with their respective message

## 6.1 Adyen Checkout Service

*From the description in the info section of the specification:*
"A web service to initiate and authorise payments with Adyen Checkout. You can use the same integration for payments made with cards (including One-Click and 3D Secure), mobile wallets, and local payment methods (e.g. iDEAL and Sofort). For more information, refer to [ady]."

Adyen Checkout Service is comparatively small, consisting of only two paths, each with one POST operation: */setup POST* and */verify POST*. No *Security Schemes* fields are present in the specification, so no settings YAML file was created.

Both tests that were run on the API returned the error code 401 with the message *No Media Type* (see table 6.1). This means the code 401 is listed in the expected responses for the operation, but the media type is missing and the response body can not be parsed. Additionally, the HTTP code 401 reflects an unauthorized call. Any "Unauthorized" error is unexpected, since the specification contains no *Security Schemes* field, so all operations should allow unrestricted access. The OAS 3.0 specification allows for certain fields to be left blank or left out completely to allow additional access restriction while still providing all non restricted information. It may be the case that the *Security Schemes* field was simply omitted. A quick comparison of the reqeust body of the smallest found falsifying value against the specification reveals the required property "merchantAccount", which is likely being used to authorize the request. In any OAS 3.0 document, any authorization element should be clearly marked as such and documented in the *Security Schemes*. At least the existence of an authorization element, via an empty *Security Schemes* field, should be included. This test result can be used to refactor the implementation of *Security Schemes* in the API to better adhere to this OpenAPI design concept.

## 6.2 Amadeus Travel Innovation Sandbox

*From their website [ama]:*
"Our easy-to-use REST/JSON APIs gives you access to test and prototype with one of the world's largest repositories of Travel Records, as well as industry leading flight search, hotel search and much more"

Amadeus Travel Innovation Sandbox API is extensive, with 23 paths with one GET method each, and uses an API key for security, applied to all paths. Of the ten tests run on the methods, five failed.

The first two tests returned *404: Invalid JSON* (see table 6.1). What this indicates is that the API returned the HTTP error code *404: Not Found*, conforming to the listed, expected responses in the documentation. However, the response body, advertised as JSON in the content-type header by the server, was not valid JSON. A look at the sample response body reveals the mime-type of the response body to be HTML, not JSON:

```
<h1>Not Found</h1>
<p>The requested URL /v1.2/travel-record/!?iuG3vk?
    was not found on this server.</p>
```

Listing 6.1: Sample Response Body

In the *Responses* section of both methods, only the HTTP code 200 and a default response labelled "Client or server error" are explicitly set. This test result can be used to either extend the definitions of responses in the documentation to account for responses returning HTML instead of JSON, or better yet, to implement the most common client and server errors to return JSON bodies, as advertised in the documentation.

The third test returned three HTTP error codes:

- 400 Bad Request
- 502 Bad Gateway
- 504 Gateway Timeout

For code 400 the error is *Unknown Property* (see table 6.1):

```
Property 'Error' has unknown property 'request_id'.
Response body was:
  {"status": 400,
   "request_id": "8c54b38a-e295-457b-b7d4-4047bf830e79",
   "message":
     {"return_date": "Invalid value: '!!!7|u=P-D`=>q~p3k'"}}
```

Listing 6.2: Unknown Property in Response Body

For code 504 the errors are *Unknown Property* and *Missing Property* (see table 6.1):

```
Required Property 'message' not found.
Required Property 'status' not found.
Property 'Error' has unknown property 'fault'.
Response body was:
  {"fault":
    {"faultstring": "Gateway Timeout",
     "detail":
       {"errorcode": "http.flow.GatewayTimeout"}}}
```

Listing 6.3: Properties Not Found in Response Body

This shows that the codes 400 and 504 were expected by SwaggerTest, but there were unexpected properties present and required properties were missing on the JSON response body. A quick comparison between the output response bodies and the components/schemas/Error (see listing 6.4) in the documentation validates these messages.

```
Error:
  properties:
    message:
      type: string
    more_info:
      type: string
    status:
      type: integer
  required:
    - status
    - message
```

Listing 6.4: Definition of components/schemas/Error

Indeed, the error definition is missing the properties“request_id” and “fault”, and the properties “message” and “status” are required. This test result can be used to either refactor the API to better adhere to the documentation, or update the documentation to reflect the real return values of the method.

The test result for the code 502 was *Unexpected Media Type* (see table 6.1), indicating that the server error 502 returned a mime-type other than JSON, which is the only media type defined in the documentation. This test result can be used to update the documentation to expect the actual media type of the response body.

Additionally, the server errors 502 and 504 reflect a problem in the inner workings of the API. If this behaviour persists, the smallest found falsifying values can be used to reproduce the fail state in the server and investigate its causes.

The test for the fourth failing method also returned *Unexpected Media Type* (see table 6.1). Again, this test result can be used to update the documentation to expect the media type of the response body.

Finally, the fifth failing test returned *Unknown Property* and *Missing Property* (see table 6.1):

```
Required Property 'message' not found.
Required Property 'status' not found.
Property 'Error' has unknown property 'Error'.
Response body was:
  {"Error":
    "Latitude / Longitude should be geolocations
     between -180 and 180, specified in Decimal Degrees:
     could not convert string to float: !"}
```

Listing 6.5: Error Message with Missing and Unknown Proeprties and Response Body

Firstly, the sample response body reveals that the API expected Latitude and Longitude to be decimal numbers between -180 and 180. Yet, in the documentation both parameters are simply of type string, with no technical format given.
Secondly, the response body does not fit the *Definition of components/schemas/Error* (see listing 6.4) found in the documentation. This test result can be used to update the documentation to provide machine-readable formats for parameters where the API expects them, as well as refactor the API to return the defined error.

## 6.3 GeoMark Web Service

*From their website [geo]:*
"The Geomark Web Service allows you to create and share geographic areas of interest over the web in a variety of formats and coordinate systems."

GeoMark Web Service has seven paths, with one method each and no security mechanisms in place. Tests run on five methods each returned 404: *No Media Type* (see table 6.1). This test result can be used to extend the documentation to contain media types for all expected error codes.

The test run on the sixth method returned 302: *Unexpected Response* (see table 6.1) and 504: *Server Error* (see table 6.1).

Both results indicate that their respective HTTP codes were not defined as responses in the documentation of the method. *302: Found* is a URL redirection, meaning the requested resource was moved. This test result can be used to integrate the 302 error code into the documentation, and to investigate why the resource is not at the expected place. *504: Gateway Timeout* means that the API serves as a proxy and received a time out while waiting for the upstream server, and this response could be integrated into the documentation as well.

The tests that were run on the seventh method returned *No Media Type* (see table 6.1) for the HTTP codes 400 and 500. This indicates the API returned *400: Bad Request* and *500: Internal Server Error*, both expected by the documentation, but with no media type defined. The smallest found falsifying value was:

```
Path: https://apps.gov.bc.ca/pub/geomark/geomarks/new
Operation type: POST
Body: bufferMitreLimit=0
    &failureRedirectUrl=!
    &redirectUrl=!
    &resultFormat=json
    &multiple=false
    &format=json&body=!
    &bufferSegments=0
    &srid=4326
    &bufferJoin=ROUND
    &bufferMetres=0
    &allowOverlap=false
    &bufferCap=ROUND
    &callback=!
```

Listing 6.6: Smallest Falsifying Value for geomarks/new

This shows that the request conforms to the method definition in the documentation. This test result can be used to extend the definition of the method to reflect the constrictions on the parameters. Additionally, the smallest found falsifying value can be used to trigger the internal server error 500 and investigate its causes. What differentiates a 500 error from the other server errors, is that it does not reflect a specific issue, but is more general, a sort of “catch all”. This warrants speculation that it is an unexpected error on the server side, possibly still unknown, and therefore highly relevant to the developers of the API.

## 6.4 BC Laws

*From their website [bcl]:*
“BC Laws is an electronic library providing free public access to the laws of British Columbia. BC Laws is hosted by the Queen’s Printer of British Columbia and published in partnership with the Ministry of Justice and the Law Clerk of the Legislative Assembly.”

BC Laws has seven paths with one method each. All seven tests returned 200: *No Media Type* (see table 6.1). This test result can be used to extend the definition of the responses to include the media type of the expected responses.

Five tests additionally returned 400: *Client Error* (see table 6.1). This indicates that the documentation does not contain the error code 400 in the expected responses. Additionally, the smallest found falsifying values show that the requests conform to their respective definitions, so a code *400: Bad Request* is not expected. This test result can be used to extend the definition of the method to reflect the constrictions on the parameters.

## 6.5 D-BAS

*From their website [dba]:*
“Welcome to D-BAS - the Dialog-Based Argumentation System. It offers a clear discussion structure, a modern web interface, a decentralized moderation system and is easy to use for everyone!”

D-BAS has ten paths, eight with one method each and two with two methods each, and uses an API key for security, applied to all paths, except the login path. The tests that were run on two methods returned 405: *Client Error* (see table 6.1). The message of the 405 error was:

```
405 Method Not Allowed
The method POST is not allowed for this resource.
```

Listing 6.7: 405: Not Allowed

This test result can be used to modify the documentation to better reflect the state of the API, or to debug the API, in the case that the 405 error is not the intended behaviour.

The tests for two methods returned 400: *Client Error* (see table 6.1). The 400 error had the following message:

```
{"status": "error",
  "errors":
    [{"location": "path",
      "name": "Invalid slug for issue",
      "description": null}]}
```

Listing 6.8: 400: Bad Request

This test result indicates that the API returns a *400: Bad Request* error whenever the path parameter “slug” does not match an existing value. This test result can be used to extend the documentation to account for the 400 error, or to refactor the API to return a *404: Not Found* error, which better reflects the fact that the request is valid, but that the specified resource is not available.

The tests for two methods returned 400: *Client Error* (see table 6.1). The 400 error had the following message:

```
{"status": "error",
  "errors":
    [{"location": "header",
      "name": "Received invalid or empty
          authentication token",
      "description": "Please provide the
          authentication token in the
          X-Authentication field!"}]}
```

Listing 6.9: 400: Bad Request

A look at the smallest found falsifying values for both paths shows the authentication token to be present in both cases:

```
Path: http://localhost:4284/api/!!/justify/0/agree/undermine
Operation type: POST
Headers: X-Authentication=
      {"nickname":"Walter",
       "token":"57f122c89c55d9f6754484eea0492"}

Path: http://localhost:4284/api/!/justify/0/agree
Operation type: POST
Headers: X-Authentication=
      {"nickname":"Walter",
       "token":"57f122c89c55d9f6754484eea0492"}
Body: {"reason":"!"}
```

Listing 6.10: Smallest falsifying values for agree/undermine and agree

This test result can be used to debug the API using the smallest found falsifying values, in order to determine the causes of this malfunction.

## 6.6 Adafruit IO

*From their website [ada]:*
"The Adafruit IO HTTP API provides access to your Adafruit IO data from any programming language or hardware environment that can speak HTTP."

Adafruit IO is the largest API of the ones tested, with 69 methods across 34 paths, which uses an API key for security, applied to all paths. The test run on the first of the 69 methods returned 200: *Unknown Property* (see table 6.1). This shows that the API returned the expected 200 response, but the response body differs from the definition in the documentation. This is the response body:

```
{"id": 289177,
 "name": "Carlo Schackow",
 "color": "#ff0000",
 "username": "schackow",
 "role": "user",
 "default_group_id": 184940,
 "default_dashboard_id": null,
 "time_zone": "Europe/Berlin",
 "created_at": "2018-06-20T12:52:45Z",
 "updated_at": "2018-07-02T13:49:31Z",
 "subscription": {"plan": {"name": "Free"}}}
```

Listing 6.11: Reponse body

And this is *#/components/schemas/User*:

```
User:
  properties:
    color:
      readOnly: true
      type: string
    created_at:
      format: dateTime
      readOnly: true
      type: string
    id:
      readOnly: true
      type: number
    name:
      readOnly: true
      type: string
    time_zone:
      readOnly: true
      type: string
    updated_at:
      format: dateTime
      readOnly: true
      type: string
    username:
      readOnly: true
      type: string
  type: object
```

Listing 6.12: Definition of #/components/schemas/User

A comparison between the response body and the documentation shows that the unknown properties “role”, “default_group_id”, “default_dashboard_id”, “subscription” and “beta_flags” are not specified in the documentation. This test result can be used to extend the documentation to include the previously undocumented properties of the schema.

68 of the 69 tests run on the methods returned 404: *No Media Type* (see table 6.1). This indicates that the documentation is missing the media type for the 404 response code. For the sake of testing, the defintion of the 404 response in these operations was extended to reflect the received responses:

```
schemas:
  404Error:
    properties:
      error:
        type: string
    type: object
responses:
  404Response:
    content:
    application/json:
      schema:
        $ref: '#/components/schemas/404Error'
    '#<Mime::NullType':
      schema:
        type: string
```

Listing 6.13: The Modified Definition of the 404 Response

Then the tests on these operations were run again, and all of them passed the test without displaying errors. It should be noted that the second mime type *#<Mime::NullType* is ill defined and should not be included in the documentation as is. Rather, the API should be updated to always return JSON for 404 errors.

## 6.7 Giphy API

*From their website [gip]:*
“Welcome to the GIPHY API. Natively embed all the best features of the world’s largest and most powerful GIF library into your app.”

The API consists of ten paths with one GET method each. The API defines an API key that is applied to all methods. One of the tests returned 404: *No Media Type* (see table 6.1). This test can be used to extend the documentation to include the media type of the 404 error.
Six of the tests returned 400: *No Media Type* (see table 6.1). This is the smallest found falsifying value for the GET method on *gifs/search*:

```
Path: https://api.giphy.com/v1/gifs/search
    ?q=%21%21%21%21%21%21%21%21%21%21%21%21
    %21%21%21%21%21%21%21%21%21%21%21%21%21
    %21%21%21%21%21%21%21%21&limit=0
    &offset=0&rating=%21&lang=%21
    &api_key=G323e76KQf3K0Bi4ECF7f8yX7ebYIKkq
Operation type: GET
```

Listing 6.14: Smallest Found Falsifying Value for gifs/search

The request conforms to the definition in the documentation, as there are no restrictions defined for any of the parameters used. It is possible, that the long instance of the query parameter “q” in the requests, with 33 characters, is a cause for this error. Another problem may be the parameter “limit=0”, as this setting would result in zero return values. Based on the definition of the parameters, however, the 400 error is unexpected. This test result can be used to debug the API, to determine the error-causing parameters, and extend the documentation to reflect the restrictions on them. This test result can also be used to extend the documentaion to include the media type of the 400 error.
Two of the tests that were run returned the HTTP code 200: *Unknown Property* (see table 6.1). This shows that the API returned the expected 200 responses, but the response body differs from the definition in the documentation. Both the sample response bodies and the definition of the schemas in the documentation were too long to be presented here (see the accompanying CD), but a comparison between them shows that the unknown properties are not defined. This test result can be used to extend the documentation to include the properties.

| Name | HTTP Code | Occurences |
|---|---|---|
| No Media Type | 200 | 7 |
| | 400 | 7 |
| | 401 | 2 |
| | 404 | 74 |
| | 500 | 1 |
| Unexpected Media Type | 502 | 2 |

Table 6.2: Received errors that report incomplete documentation, arranged by number of occurences per HTTP code

## 6.8 Summary

SwaggerTest was tested on seven REST APIs with a total of 130 methods. 102 of these methods failed the test and nine of these produced two ore more errors, for a total of 116 distinct errors. This shows the utility of SwaggerTest in exposing previously unfound errors, even in long running projects.

The errors are distributed as follows:

As table 6.2 shows, 93 of the error messages were of the types *No Media Type* and *Unexpected Media Type* (see table 6.1).
These errors indicate that the expected responses for the tested methods were insufficiently defined. The HTTP response codes received were expected, but the expected response either contained no media type, or was missing the received one. This left *SwaggerTest* unable to parse the response body and an error was triggered.

Ten of the error messages were of the types *Invalid JSON*, *Unknown Property* and *Missing Property* (see table 6.1). Errors of these types are produced by the `JSONBodyTester`, the `BodyTester` customarily defined to parse JSON response bodies: this means the call was sucessfully executed, the response was expected, and its response body was of the correct media type, but errors within the response body were detected. In the case of *Invalid JSON*, the received reponse body could not be properly read. *Unknown Property* and *Missing Property* indicate that a required property was missing, or that there was an unexpected property within the response body.

*Client Error*, *Server Error* and *Unexpected Response* (see table 6.1) made up the last 13 of the error messages. These errors show that the API returned an HTTP code which was not defined in the expected responses of the method.

In general, since the calls all conformed to the documentation, the received response codes 400, 401 and 405 all indicate missing or incorrect documentation of parameters and security schemes. The received responses 500, 502 and 504 indicate that some calls to the server caused internal errors, an occurence that no input should be able to induce, and highly relevant to the developers of the API.

# Chapter 7

# Conclusion

In this thesis, the possibility of using generative testing on REST APIs was explored. With ever more developers embracing the OpenAPI design philosophy and working with the tools provided by the OpenAPI community, any tool that aids the development process will aid in the emergence of better documented, more standardized REST APIs. To this end, SwaggerTest was developed to automatically generate and execute tests based on the OpenAPI 3.0 specification of any given REST API. It derives basic logic properties from the definition of the API and checks them against the results returned by the API. These properties describe which types of input the API is expected to accept, and which output the API is expected to produce.

## 7.1 Accomplishments

SwaggerTest performs tests with randomly generated input: its strength lies in finding API breaking input and ill-constructed or missing definitions within the documentation. All the tests performed on the APIs ran sucessfully, and with **minimal time and effort** required. For each API, and a majority of the tested methods, errors were found.

**Minimal Error Producing Input** The use of generative testing proved sucessful in producing **counterintuitive input** that causes the API to produce errors, and in **minimizing the input** to aid the user in determining what specifically caused these errors. Finding these "edge cases" is an important task. Any REST API needs to be completely reliable to be effectively integrated into any software project. The *shrinking* ability of the generative testing framework was an additional advantage, as it allowed the error producing inputs to be minimized, removing irrelevant parts of the input that do not directly contribute to producing the error. Shorter and more specific inputs enable quicker debugging.

**Inclomplete API Definitions** In an API community survey conducted by ProgrammableWeb in 2013 [pro], "Complete and accurate documentation" was rated as the highest "Important factor" by API consumers. **SwaggerTest effectively finds gaps and incongruences in the definition of the tested API**, encouraging the developers to extend and improve the document. To facilitate developer's work with the API, especially for the auto-generation of client SDKs, the detailed description of parameters, response types and errors is an essential part of the documentation of the API. How an auto-generated client will react and adapt to client and server errors depends on the ability to parse them in the first place. Where a developer could resort to guessing the media type, or determining it by triggering the specific error, an auto-generated SDK could only leave the media type blank and in turn not parse the corresponding errors. The same principle applies to the parameters used in calls to the API. Certain limitations are easy to intuit by humans, e.g. that a parameter of "desired number of results" to a call must be equal to or greater than zero, or that a "time zone" parameter must correspond to a real time zone. To make these parameters accessible to programs using them, the definitions would need to be extended, with a minimal value in the case of "number of results", and a regular expression or an enumeration of all possible values for "time zone". In this way, a complete definition of all parameters, response types and errors can go a long way towards cleaner code and better useability.

**Extensive Testing** As the input was generated on the basis of the predefined properties of the API, it was easy to test the API against large datasets of input with varying lengths and contents. This means that **SwaggerTest ensures an exhaustive testing process**. It also made it simple to limit the testing to a fixed ammount of calls, should the need arise.

**Extendability** As this thesis was developed in a set time frame, only the essential media types and tests were performed. However, **the code is set up for future users to define their own media types and test them as they please**. Hopefully, this will make SwaggerTest more useable to a wider audience, despite its limitations.

## 7.2 Limitations

**Complex Testing** All properties that can be extracted from the OAS 3.0 document are very simple. This excludes more complex properties, e.g. which input produces which output, whether or not two identical calls to the same resource produce the same output, or whether a series of calls to the server produces a desired output. These kinds of properties are not common enough to be applicable to all or most REST APIs described with OpenAPI 3.0. To test these properties, developers will still have to manually define tests.

**Overloading the Server** The large number of tests run on the servers can cause problems. The process of shrinking the values, once error producing input has been found, can run for up to 100,000 iterations. In the case of complex calls with multiple parameters, that is not an unrealistic number of calls. This mass of requests can incapacitate the server to the point of shutdown, and be perceived by administrators as a DoS attack. An upper limit for the number of calls, as well as stop and pause signals for the testing, were introduced. This however severely limits the ability of SwaggerTest to exhaustively test and effectively minimize the input. A solution may be to always test against seperate, local builds of the API, where a large number of tests run does not affect other users. Otherwise, testing will be severely limited, or pose the threat of overloading the server.

**Positive Results** The use of randomly generated input, with testing limited to 100,000 tests causes a large portion of the methods of the APIs to exclusively return client errors, as the chance of generating valid input is small to nonexistent. Of course, for most of the input, the API will not have a specific resource to match, and so will return *404: Not Found* or *400: Bad Request*. This means that the “successful” return values of the API may go completely untested with SwaggerTest. During testing, the only branches that returned *200: Successful* were search branches, which accept any input.

## 7.3 Future Work

**Extending SwaggerTest** SwaggerTest, as it was developed for this thesis, fullfills the testing needs of D-BAS specifically. It was developed to work on any REST API that can be defined in a OAS 3.0 document. From time constraints, however, arose the need to omit certain capabilities from the program and focus on the essential features. Among the functions left out of development were cookie parameters, the implementation of multiple security schemes, and any media type besides *JSON* for requests as well as responses. Additionally, the ability to induce all expected response types from each method, would help in ensuring an exhaustive and balanced testing of the API. As this information cannot be read from the OAS 3.0 document, perhaps an addition to the settings file could offer a solution here.

**Publishing SwaggerTest** From the outset, the plan was to publish the program resulting from this thesis as an Open Source project on SwaggerHub, if it performed well enough to be of use in the OpenAPI community. With the current state of SwaggerTest, this plan will likely soon come to fruition. This publication may initially generate feedback on the program, and furthermore lead to extensions of some of the less developed features of SwaggerTest.

**Extending the OAS 3.0 Document** The ability to describe more complex interactions with the API, i.e. common *use cases*, would be a valuable addition to the specification. This could include examples for the order in which specific paths and methods are accessed, and which outputs are reused as input for subsequent calls. The interconnectivity between paths, and the special importance of paths such as *login* and *logout*, would be highlighted with this addition.

# Bibliography

[ada] *The internet of things for everyone*. https://io.adafruit.com/api/docs/#.
Accessed: 09.07.2018

[ady] *Checkout documentation*. https://docs.adyen.com/developers/checkout.
Accessed: 09.07.2018

[ama] *Travel Innovation Sandbox*. https://sandbox.amadeus.com/.
Accessed: 09.07.2018

[api] *APIs.guru's Wikipedia for Web APIs*. https://github.com/APIs-guru/openapi-directory.
Accessed: 09.07.2018

[bcl] *BCLaws API*. http://www.bclaws.ca/civix/template/complete/api/index.html.
Accessed: 09.07.2018

[dba] *D-BAS The solution for large-scale Online-Discussions*. https://dbas.cs.uni-duesseldorf.de/.
Accessed: 09.07.2018

[geo] *Geomark Web Service*. https://www2.gov.bc.ca/gov/content/data/geographic-data-services/location-services/geomark-webservice.
Accessed: 09.07.2018

[gip] *Welcome to the GIPHY API*. https://developers.giphy.com/.
Accessed: 09.07.2018

[int] *SwaggerHub Integrations*. https://swagger.io/tools/swaggerhub/integrations/.
Accessed: 09.07.2018

[jso] *JSON Schema*. http://json-schema.org/.
Accessed: 09.07.2018

[KMB+18] KRAUTHOFF, Tobias; METER, Christian; BETZ, Gregor; BAURMANN, Michael; MAUVE, Martin: Dialog-Based Online Argumentation. In: *Computational Models of Argument*, Warsaw, 2018.

[mer] *Mermade Swagger 2.0 to OpenAPI 3.0.0 converter*. https://mermade.org.uk/openapi-converter.
Accessed: 09.07.2018

[oas] *OpenAPI Specification*. https://swagger.io/specification/.
Accessed: 09.07.2018

[pro] *API Consumers Want Reliability, Documentation and Community*. https://www.programmableweb.com/news/api-consumers-want-reliability-documentation-and-community/2013/01/07.
Accessed: 09.07.2018

# Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 13. Juli 2018 Carlo Schackow

Hier die Hülle

mit der CD/DVD einkleben

**Diese CD enthält:**

- eine *pdf*-Version der vorliegenden Bachelorarbeit
- die LaTeX- und Grafik-Quelldateien der vorliegenden Bachelorarbeit samt aller verwendeten Skripte
- die Quelldateien der im Rahmen der Bachelorarbeit erstellten Software *SwaggerTest*
- den zur Auswertung verwendeten Datensatz
- die Websites der verwendeten Internetquellen