

Automatic Test Generation of REST APIs

Automatiserad testgenerering av REST API

Axel Karlsson

Supervisor : Anders Fröberg

Examiner : Erik Berglund

External supervisor : Linus Lindholm

Upphovsrätt

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida <http://www.ep.liu.se/>.

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: <http://www.ep.liu.se/>.

Abstract

One of the most common APIs today in web development are REST APIs. They allow for interoperability between computer systems on the internet. One of the challenges with RESTful systems is that they are currently hard to automatically generate tests for, partially due to the fact that REST does not have an actual defined standard. OpenAPI specification has emerged as a way to standardize APIs and also describe REST API in a documented format. The aim of this thesis is to investigate a way to automatically generate tests for a REST API which is given in the form of a OpenAPI specification. The work was carried out by first conducting a study over available tools which could assist in the generation of the tests, then the tests were set up to be executed against an online web API. Finally an evaluation was to be conducted, with test coverage as a metric, regarding how high test coverage could be achieved from the tests by looking at the returned status code that an operation would return. The results show that although it is possible to set up the tests and make them executable, it is hard to define what the response should be in the generated tests and this makes it difficult to predict what the expected result of a test should be. It is concluded that further work would need to be done in order to properly evaluate the code coverage of the tests.

Acknowledgments

I would first like to thank everyone at Pitch Technologies who have been very welcoming and also for providing a place to actually perform the work during the COVID-19 pandemic. I would also like to specifically thank my supervisor at Pitch, Linus Lindholm, whom first came up with the idea for this thesis and has provided help to keep me going whenever I got stuck. I would also like to thank my examiner Erik Berglund who was always ready to answer questions I had via email. Finally I also give thanks to my family for encouraging and continuously supporting me during the time of the thesis.

Contents

Abstract	iii
Acknowledgments	iv
Contents	v
List of Figures	vii
List of Tables	viii
1 Introduction	1
1.1 Motivation	1
1.2 Aim	1
1.3 Research questions	2
1.4 Delimitations	2
2 Theory	3
2.1 Web API	3
2.2 HTTP	3
2.3 REST	4
2.4 The value of testing	6
2.5 Combinatorial testing	7
2.6 Test coverage	7
2.7 Testing of REST API	7
2.8 OpenAPI Specification	8
2.9 Tcases	10
2.10 Node.js	11
2.11 Mocha and Chai	12
3 Method	13
3.1 Pre-study	13
3.2 Implementation	14
3.3 Evaluation	15
4 Results	17
4.1 Pre-Study	17
4.2 Implementation	18
4.3 Test results	22
5 Discussion	24
5.1 Results	24
5.2 Method	25
5.3 The work in a wider context	26

6 Conclusion	27
6.1 Future work	27
Bibliography	29

List of Figures

2.1	An example of the output produced when testing with Mocha	12
3.1	A snapshot of the Swagger pet store and its interactive UI.	15
3.2	Test Coverage Model: Levels & Criteria.	16

List of Tables

2.1	Example of common status codes	6
4.1	Tests for POST requests	18
4.2	Tests for GET requests	18
4.3	Amount of successful tests for GET requests	23
4.4	Amount of successful tests for POST requests	23



1 Introduction

Today web services play a major role when developing applications and programs, due to their flexibility and also their simplicity. A common way to develop such applications is by using a RESTful web service that may provide all the data via an API over the network by using the HTTP protocol. Testing these REST APIs provide their own set of issues. Inputs and outputs are sequences of HTTP requests and responses to a remote server. REST is not a standard on its own, instead it works more as a guideline on how RESTful APIs should be built. Open API specification is emerging as a popular way of standardising REST APIs and can be used to describe a system that is meant to be designed as RESTful.

1.1 Motivation

Pitch Technologies is a company that provides services and solutions for interoperable distributed simulation systems. Pitch can at the time of writing use their own software, Pitch developer studio (PDS), to generate a Java, C++ or C# based library for interacting with their simulations. PDS can also generate a complete web server in the form of a REST API. This can allow any HTTP capable application, such as a web application for example, to interact with the generated simulations. In addition to this a specification describing the REST API is also generated. What is missing is a way to dynamically test such a system by automatically generating test cases for this API based on the generated specification. The work consists of looking into already existing frameworks and tools that provide automatic testing for REST APIs to see if there exists one which can be used for this purpose. The goal is to have a tool that can provide tests for the generated API, with the help of the generated specification. As a metric for the generated tests, test coverage will be used as the main criterion in order to see how much of the system that has been tested.

1.2 Aim

As previously mentioned, tools and testing frameworks already exists for working with REST APIs and their specifications [19]. The thesis will first focus on investigating what tools can be used to aid in the process of automatically generating test cases from a REST specification.

If a tool is found to be able to aid in the generation of the test cases, then it will be used to automate the generation of a test suite for an example REST API. Then the generated tests will be executed and the results of the tests will be reported. If no suitable tool is found, then the work will instead consist of building a tool from scratch.

Based on the description above an evaluation of the tests and how they performed will be done where the generated tests are evaluated. This evaluation is based on how large part of the REST system that is actually covered by the testing. This is referred to in the report as test coverage.

1.3 Research questions

Given the stated motivation and aim of the thesis, the following research question was conducted:

1. How can automated tests, given an Open API specification of a RESTful API, be implemented in order to provide high test coverage of the specified system?

1.4 Delimitations

This thesis report will only focus on testing the different CRUD operations of an application and not generate tests for e.g websocket connections. This thesis will only consider test coverage as the main evaluation criteria. As discussed later on in the report, coverage does not always guarantee the effectiveness of a test suite [8].



2 Theory

2.1 Web API

Web application programming interface (API) is a set of defined interfaces from where the interactions are performed between an enterprise and the actual application that are using its assets. A web API can be described as a set of functions and procedures that enables the possibility to build upon the functionality of an existing application.

2.2 HTTP

Hypertext transfer protocol (HTTP) is the main protocol for communication on the world wide web and also when interacting with RESTful APIs, which we will be discussed later on. HTTP is an application level protocol which enables collaboration between hypermedia information systems. Originally it was designed for transmitting hypermedia documents, HTML for example, and also to communicate between web browsers and web servers. HTTP is developed after a client-server model meaning the client sends the HTTP request which is handled by the server whom performs actions or some task and returns a response, back to the client. Some of the more common operations include create, read, update and delete (CRUD) [21]

A HTTP message is usually built on four main components:

- Method
 - The operation that is to be done. An operation can be retrieving a certain web page for example.
- Resource path
 - The resource path is an identifier used to specify on what resource an HTTP operation should be applied. This could be the path to a document to fetch, for example.
- Headers
 - Metadata on the form of a list of key/value pairs. One example can be the accept header which advertises which content types the client is able to understand and which should be returned to the client.

- Body
 - The payload that the message can contain. This can for example be the HTML text of a web page that is returned as response to a request.

The HTTP protocol is stateless, meaning that there is no link between two successively carried requests, on the same connection. All of the incoming request must therefore carry all the information that is needed to be processed, since the HTTP protocol does not store any previous information. In order to maintain a state for a user doing several related HTTP requests, cookies have to be employed. Cookies function as HTTP headers with a unique id created by the server in order to keep track of a given user. It is the task of the user to such a header in all of its HTTP requests [21].

2.3 REST

Representational state transfer (REST) is described to be an architectural style used to create and organize distributed systems. [7] REST is, however, not a standard per se. It is not clearly defined, meaning there does not exist a clear set of rules that can be followed to build a REST system. REST can instead be seen as a set of architectural guidelines for building web services on top of HTTP. The main idea behind REST is that a distributed system organized after a REST style will improve in areas such as performance, scalability, portability, reliability, modifiability, etc. RESTFUL APIs provide a uniform interface to interact with different resources. For example a song in a music player API. These interactions are performed using CRUD operations generally via HTTP.

2.3.1 Constraints

There are two ways to describe a system according to R. Fielding [7]. The first is to start without any initial knowledge of the system being built or using known components until the needs for them are satisfied. The second approach that can be taken, which REST follows, is to start with the full needs of the system and add constraints to each individual component until the influences on the system can interact continuously with each other. In a REST architecture, a null-state is initially defined and the constraints are added one after another.

Fielding [7] also defines six different architectural constraints, that when implemented, enables a web service to be considered a true RESTful API.

Uniform interface

One of the main characteristics of the RESTful API is its uniform interface constraint. Managing a uniform interface between components will simplify the job of the client when it comes to interacting with the system. This will also imply that clients implementation will be independent of each other. By defining a standard and uniform interface, the implementation for independent clients are simplified since they are given a clear set of rules to follow. A concept relevant to achieving a uniform interface is HATEOAS, which will be discussed later on.

Client-Server

The principle that is invoked here is the separation of concern. A divide and conquer strategy that is used in software engineering, separating a program so that each part only addresses a separate concern, keeping apart sections that do not need to be held together [12]. When designing for a RESTful service, it allows for the separation of the front-end code from the server side code which should primarily focus on storage and server-side processing of data. Servers and clients are able to be developed or replaced independently with the condition that the interface between them is not altered.

Stateless

The constraint to be added on top of the previous on mentioned is the stateless constraint. Communication between a client and server should be stateless, that is the client needs to pass all the necessary information when making the request, without taking advantage of any stored data. No information of the latest HTTP request will be stored on the server, it will treat each of the requests as if it is unique.

Cacheable

The purpose with this constraint is that every response to a request must either be implicitly or explicitly set to be cacheable. A well managed caching could eliminate partly or completely, some client-server interactions and improve upon the scalability and performance of the system.

Layered system

The design of REST takes into account the possibility that large amounts of traffic can be experienced on the web, and that the system should be able to handle this traffic. To achieve this, the components are separated into layers and allowing each layer to only to use the one below and at the same time only communicate its output to the one above. For example, the API can be deployed on server A, store data on server B and authenticate requests on server C. A client should not be able to tell if it is connected to the end server directly or to an intermediary. Layered system is the constraint which is however, more important to the deployment of the Web APIs rather than their design.

2.3.2 Resources

One of the main aspects of the REST architecture are the resources. Essentially anything in a REST architecture can be a resource, such as a web page, an image, a person etc. Resources are what define what the services are going to be about as well as what information will be sent back and forth, and their related actions. The structure of a resource is set up so that it has a representation, a way of representing data in different formats. Also an identifier, which works as a URL that retrieves a specific resource at any time. There also exists metadata that represents content-type, time of modification etc. Finally there is also control data which can work as cache control [21].

2.3.3 Resource Identifiers

There should be a method of uniquely identifying the resource and provide a full path to it at any given moment. It is not enough to assume that the resources ID on the storage medium, that is used, to be enough. Instead the individual resources are identified as requests that can be on the form of a uniform resource identifier (URI). The actual resources will defer from what is sent to the client, the format of the internal representation can be different to what is the internal representation for example.

2.3.4 HATEOAS

One of the constraints applied to make REST uniform is "hypermedia as the engine of application state" (HATEOAS). The purpose of HATEOAS is to let the client interact with the REST API through the provided responses from the server. This means that a client can, for example, enter a REST application with the help of a simple URL and the following actions that might be taken by the client are discovered from the returned resource representation from the server. This could also be referred to as a sort of discoverability.

```

1 {
2   "name": "Axel"
3 }

```

Listing 2.1: Example of NON-HATEOAS response

```

1 {
2   "name": "Axel"
3   "links": [{
4     "rel": "self",
5     "href": "http://localhost:5000/example/1"
6   }]
7 }

```

Listing 2.2: Example of HATEOAS response

2.3.5 Status codes

Status codes are another standard that the REST API uses when it is based on HTTP. The status code summarizes the response that is associated to it. Some of the more common ones include 404 (not found) and 200 (OK). These status codes can be helpful for the clients to interpret the response. However, it should not function as a substitute for it. Status codes may be hard to interpret from their number alone as it can be difficult to transmit exactly what has happened. In some cases it can be enough, however further information is useful to help the client solve the issue. While testing RESTful application the status codes can play a vital part in seeing what the results of specific tests were [5].

Table 2.1: Example of common status codes

Code syntax	Status code	Description
2XX	200	OK
	201	Created
	202	Accepted
	203	Non-Authoritative Information
	204	No Content
3XX	300	Multiple Choices
	301	Moved Permanently
	302	Found
	303	See Other
4XX	400	Bad Request
	401	Unauthorized
	402	Payment Required
	403	Forbidden
5XX	404	Not Found
	500	Internal Server Error
	501	Not Implemented
	502	Bad Gateway
	503	Service Unavailable

2.4 The value of testing

When writing and developing programs it is natural that bugs and problems are introduced in the process and also scenarios and situations that haven't been accounted for will at some

point be brought up. The original requirements for the program could experience changes over time and new faulty scenarios could be added when the program or application evolves over time. Testing is in essence the process of executing a program with the intent of finding errors. [16]

Detecting errors early in the process can save a lot of value both in time and assets. If a fault instead is detected at live version of a system this can have more dire consequences and could potentially damage the reputation of the program or application.

2.5 Combinatorial testing

Combinatorial testing, also known as t-way testing, is a method for software testing. Combinatorial testing can effectively detect failures that are triggered by the interactions of parameters in the software under test (SUT). The method consists of testing all possible discrete combinations for each pair of input parameters. It is generally too large to test all of the possible combinations in the SUT, therefore it is normal to use a type of chosen test vectors in order to faster test the system than by using an exhaustive method. This requires a well defined model of the test parameters and is critical in order to effectively perform combinatorial testing [17].

2.6 Test coverage

Test coverage is used when testing in order to see how much of the system is actually covered by the tests. This will be basis and main criteria when evaluating how well the automated testing can perform. It should be noted that high coverage does not indicate that the test suite is well tested. L. Inzomtseva et al. [9] found that the correlation between coverage and effectiveness is low / moderate when the size of the test suite is controlled. They also conclude that high levels of coverage alone, is not enough to determine the effectiveness of a test suite.

2.7 Testing of REST API

Testing of REST APIs provides difficulties of its own. The inputs and outputs of the RESTful application correspond to the request and responses from the server. They are also centered around the fact that web-services are distributed, lacks a user interface and are based on late data binding. An additional issue is also testing the restfulness of the application. That is, testing if the application actually fulfills the characteristics of traditional RESTful web-services e.g uniform interface, caching, client-server structure, etc. Chakrabarti et al. [4] presents an article for actually testing the connectedness of a RESTful API. This is done via first developing a formal notation that was used to specify the RESTful web-service. With the help of this specification, automated testing could be achieved on the web-service.

Seijas et al [11] brought forth a technique to generate tests for a RESTFUL API. They first developed a property-based test (PBT) model which gave an idealised model of A REST web-service. The approach proposed in this work is however, mainly a manual approach.

Most testing for web-Applications is done via black box testing, meaning that tests are performed not knowing the inner details of the system and instead only performing tests from the outside perspective. A. A. Andrews et al. [1] addresses the problem of automatic as well as black box testing of web applications. The approach taken applies a criteria to a structured model of the application. Tests for entire web applications are created by creating sub-tests that are developed by using models from the lower level abstractions of the application. The papers main focus is the state dependent behaviour of the web applications. The paper addresses that the usage of finite state machines (FSM) provide a suitable method to model software behaviour. However, when modelling web applications with FSM even the simplest web sites can suffer from the problem of state explosion. The main issue is that

there exists several possibilities for different input fields and forms. Therefore an FSM-based testing method should only be used if there are techniques that can generate tests that are considered descriptive enough so that the tests are effective yet small enough to be practically useful.

An alternative approach to testing REST APIs was taken by A. Arcuri [2]. Here the author proposes a technique for automatically collecting white-box information from the web service. Then such information is used to generate test cases with the help of an evolutionary algorithm. This approach has been implemented in a tool written with Java and Kotlin, called EvoMaster. The experiments were run on different web services that would range from 2 to 10 thousand lines of code. The technique proposed by the paper managed to generate test cases which managed to detect a range of bugs in the web services. When comparing to existing test cases, however, it was noted that the metric for code coverage was lower from the tool. The possibilities of why this is the case is discussed in the paper and also addresses the main reasons to why these results were obtained. For example one of the issues is string constraints, since strings are complex to handle and the tool does not support techniques needed to handle them properly when unit testing. Another issue is access to databases and external web services. There is no possibility to check what is inside a database or external resource at the moment, also it is not possible to generate or modify such data.

2.7.1 Coverage criterion for REST APIs

Different coverage criterias are used to determine what level of coverage that the tests of a certain RESTful system actually provide. There exists different types that are more suitable depending if the tests are performed from a white-box or black-box perspective. A. Martin-Lopez, et al. [14] discusses different types of test coverage criteria that are suitable to use when writing tests for REST APIs. There is first the input coverage where elements such as path coverage, operation coverage or parameter coverage can be examples of criterions used to determine how well the tests function. Then there is also the output coverage criterion which can relate more to the actual responses that the API can produce when a request is sent. An example of this can be status codes that will tell whether an operation was successful or not.

2.8 OpenAPI Specification

The OpenAPI specification (OAS) (formerly known as the Swagger API) is an API documentation specification for RESTful web-services. It enables the creation of machine-readable interface descriptions enabled for documenting, producing, consuming and visualizing APIs that are based on HTTP. An OpenAPI definition can be used by tools that perform documentation generation in order to display the API. It can also use tools for code generation in order to generate servers and clients in various programming languages, testing tools, etc [20]. S. Karlsson, et al [10]. investigates a method to use automatic property-based tests which are produced from OpenAPI documents that describe a REST API under test. OAS can be used in this case when generating the test cases. If the REST API follows a standard which allows the description of an OAS, then tests could be generated based on this and could also provide a standard for RESTful applications to follow. This was done by H. Ed-douibi [6] where OpenAPI and test suits are used to generate test cases. The approach was implemented in a plugin which managed to provide code coverage on an average of 76.5% of the elements that were tested from the description.

One of the issues that can arise with OAS is describing dependencies. Web services often come with dependency constraints that may, for example, restrict two or more input parameters that together can form valid calls to the service. Currently OAS provide little or no support for describing the dependencies among parameters. This is discussed by A. Martin-Lopez, et al [13]. where they draw the conclusion that some sort of dependency of this kind is extremely common.

An example of an OpenAPI specification can be seen in Listing 2.3. The example shows the definition of a pet store, created by OpenAPI. Here the different paths and parameters can be seen. The different CRUD operations that can be used to call the REST api are also shown as well as the expected responses.

```

1 paths:
2   /pets:
3     get:
4       summary: List all pets
5       operationId: listPets
6       tags:
7         - pets
8       parameters:
9         - name: limit
10          in: query
11          description: How many items to return at one time (max 100)
12          required: false
13          schema:
14            type: integer
15            format: int32
16       responses:
17         '200':
18           description: A paged array of pets
19           headers:
20             x-next:
21               description: A link to the next page of responses
22               schema:
23                 type: string
24           content:
25             application/json:
26               schema:
27                 $ref: "#/components/schemas/Pets"
28         default:
29           description: unexpected error
30           content:
31             application/json:
32               schema:
33                 $ref: "#/components/schemas/Error"
34     post:
35       summary: Create a pet
36       operationId: createPets
37       tags:
38         - pets
39       responses:
40         '201':
41           description: Null response
42         default:
43           description: unexpected error
44           content:
45             application/json:
46               schema:
47                 $ref: "#/components/schemas/Error"
48   /pets/{petId}:
49     get:
50       summary: Info for a specific pet
51       operationId: showPetById
52       tags:
53         - pets
54       parameters:
55         - name: petId
56          in: path
57          required: true
58          description: The id of the pet to retrieve
59          schema:
60            type: string
61       responses:
62         '200':
63           description: Expected response to a valid request
64           content:
65             application/json:
66               schema:
67                 $ref: "#/components/schemas/Pet"
68         default:
69           description: unexpected error
70           content:
71             application/json:
72               schema:
73                 $ref: "#/components/schemas/Error"

```

Listing 2.3: OpenAPI excerpt of Petstore in YAML

2.9 Tcases

The OAS project contains a large amount of open source tools involving different categories for working with the OAS specifications. This includes both generating code from a given

OAS specification and also generating a specification itself [19]. One of these open source tools is Tcases which is a model-based test case generator. The tool is primarily for designing black box test, thus the coverage criteria is instead guided from the input space of the system. With Tcases there is the possibility to define the input space for your system under test and the level of coverage that you want. Then Tcases will generate a minimal amount of test cases that meets the requirements specified. Tcases has the ability to generate test cases directly from a given OAS specification. The generated test cases, which are generated in the form of either YAML or JSON, can be seen as models from which actual tests can be generated. The generated tests provide enough information to be able to build tests that can guarantee that the minimum requirements of the system are covered, meaning that 100% coverage is guaranteed for the specified requirements. At the moment of writing, Tcases can only generate these building blocks for the test cases and not generate test cases that are ready for execution, alas there is no way of running these generated tests and this will have to be further implemented by the programmer. The test cases can still be used to transform and guide the construction of actual test suites however [22].

The generated JSON file will provide the path to the server that is tested and the operation which is needed in order to request it. This information is of course dependent on what has been previously defined in the OAS specification provided. For each resource, an amount of test cases will be generated depending on the different cases that need to be covered. These test cases will provide an ID for the test case that is run for the operation to the specific path. It will also provide the different properties that the test cases consists of and which are to be used when building the test cases. It is also defined whether there exists any values and if so, what types these values are as well as what they are actually defined as. With the help of the parameters specified, the test cases can be built and generated automatically [23].

An example of how the JSON schema can look like can be seen in listing 2.4

```

1  "GET_pets-petId": {
2    "has": {
3      "path": "/pets/{petId}",
4      "server": "http://petstore.swagger.io/v1",
5      "operation": "GET",
6      "version": "1.0.0"
7    },
8    "testCases": [
9      {
10       "id": 0,
11       "has": {
12         "path": "/pets/{petId}",
13         "server": "http://petstore.swagger.io/v1",
14         "operation": "GET",
15         "version": "1.0.0",
16         "properties": "petId,petIdValue"
17       },
18       "path": {
19         "petId.Defined": {
20           "has": {
21             "style": "simple"
22           },
23           "value": "Yes"
24         },
25         "petId.Type": {
26           "value": "string"
27         },
28         "petId.Value.Length": {
29           "value": "> 0"
30         }
31       }
32     }
33   ]
34 }
```

Listing 2.4: Test case for a GET request for the Petstore in JSON format

2.10 Node.js

Node.js is a run time environment for JavaScript. It is open-source and designed for building scalable web applications [18]. It contains a large community and there exists a huge library

of modules that can be installed with the help of the Node package manager (NPM). NPM allows free publication of packages shared publicly and also allows collaboration of packages and project. The modules that can be installed can vary in size and both contain large server frameworks to also containing smaller functions.

2.11 Mocha and Chai

Mocha is a javascript test framework which allows the writing of asynchronous unit tests[15]. Chai is an assertion library suited for behavior-driven development (BDD) or test-driven development (TDD) which can be combined to work together with Mocha [3]. Both Mocha and Chai are both using Node.js, a library which allows JavaScript to be run outside of browsers. Mocha is the part of the testing that actually triggers the tests and can report the results. Chai handles the assertion part of the tests. Both of the tools are open source and can be accessed via the node package manager (NPM). Combined together they can be used to automatically generate test suites used to interact with web services. Test suites that can provide information and knowledge of what is being tested and what errors that have been encountered can easily be generated.



```
> mocha

Testing the pet store
  Testing different gets
    ✓ status (327ms)
    ✓ findbyStatus (1337ms)
    ✓ findByTags (317ms)
    ✓ petId (408ms)

4 passing (2s)
```

Figure 2.1: An example of the output produced when testing with Mocha



3 Method

In this chapter the method will be described and the work that has been carried out will be explained thoroughly. More so the pre-study, implementation and evaluation phases will be elaborated. The pre-study was performed in order to evaluate different tools that could work with OAS and read out valuable data from it. It also includes investigating how to properly evaluate testing of RESTful services. The implementation part will describe how the tools are meant to be used and integrated to perform the work that was to be done. Finally the evaluation phase would elaborate on how to interpret the results from the tests that were generated.

3.1 Pre-study

As mentioned earlier there exists several frameworks that are available to work with RESTful services and there are also tools available that can be used to work with OAS. Therefore a pre-study was conducted to investigate if there existed anything in the area of test generation for REST APIs based on an OAS. From the theory it can be seen that tools exist for testing RESTful services but there does not seem to be as many options provided for completely automated test generation. This opens the possibility of writing a program, with the help of already existing tools, that completes the full process. The goal of the pre-study would therefore be to focus first on finding a tool or framework which could aid in generating tests or at the least use the provided information from the OAS which could be used in order to automatically generate a test suite. A first approach in finding a suitable tool is to look at the ones that are proposed by the OpenAPI Initiative [19] where there is a section that lists tools suitable for working with testing based on a OAS. This is where Tcases was first discovered and considered as a possible option.

3.1.1 Requirements

Before starting the actual implementation of the test generation, there are some basic requirements that need to be fulfilled for the automatically generated tests.

The requirements for the generated test suites included:

- The tests need to run against a server with a RESTful interface.

- This REST server should have a corresponding OAS which describes all the paths, operations, formats, data, etc.
- For the actual testing, some sort of data must be checked when the tests are run. There needs to be a request and an expected response for each test case, This is to assert if the test has succeeded or failed and also helps to evaluate which parts of the system that has been tested. This should be provided by the corresponding OAS.

There was also the need to consider which programming languages to use and if there existed additional software or tools that could potentially be useful when working with testing of REST APIs.

The goal of a generated test should in this case be to perform an operation against a specified path in the REST API. The test should, based on the data provided in the OAS, perform a request for a certain operation against a path of the REST system. Then the test checks if it was successful or not based on the response given from the server. The test should expect some type of response from the server before making the request and will evaluate if the test succeeded or not based on if the expected result matched the actual result. This evaluation is discussed in the sections that follows.

3.2 Implementation

The implementation phase would consist of developing the automatic testing with the help of the tool Tcases. Tcases performs part of the work meaning it can, given an OAS, produce a JSON file that describes test cases and their input values. From the documentation, it is also stated that the generated test cases will provide high coverage of the system defined in the given specification, if needed [23]. From the produced JSON file, the information to build test cases can be obtained to see their needed inputs, if they should fail, parameter values and more. The next step is to generate the tests from the given JSON file. This will be written in JavaScript and with the help of NPM. The reason for choosing to implement this with JavaScript is that if the tool would prove to be useful in the future, then it could be published in the NPM network publicly.

As a proof of concept that it is actually possible to automatically generate the test cases from the given specification, a prototype is implemented. Building the tests consists of constructing http requests to a web server and using CRUD operations to test the different functionalities that are specified in the OAS. The tests are built with the help of Mocha and Chai in order to be able to structure and run the tests while at the same time getting feedback from the executed tests. Mocha and Chai provides the ability to assert results to check if they are correct. The JSON file that is generated from Tcases contains the necessary information to set up the requests to be done for each test. For each operation that is specified in the OAS Tcases will be able to generate simple tests based on the specified input and the values. This means that successful tests are generated but also tests that will be considered to fail.

For the prototype that is being built, a web-server that is hosted by the OAS community is used in order to perform the tests against a live RESTful service. The same web service is also described by a OpenAPI specification that is used to generate the tests. This allows the most essential CRUD operations to be performed and will give a good first basis of what the generated tests will look like. To compare the the requests made by the generated tests, OpenAPI/Swagger provides an interactive UI that can be used to manually send requests to the pet store server. The requests that are being sent via the automated tests can also be manually executed against the sample pet store server by using this UI in order to compare them.

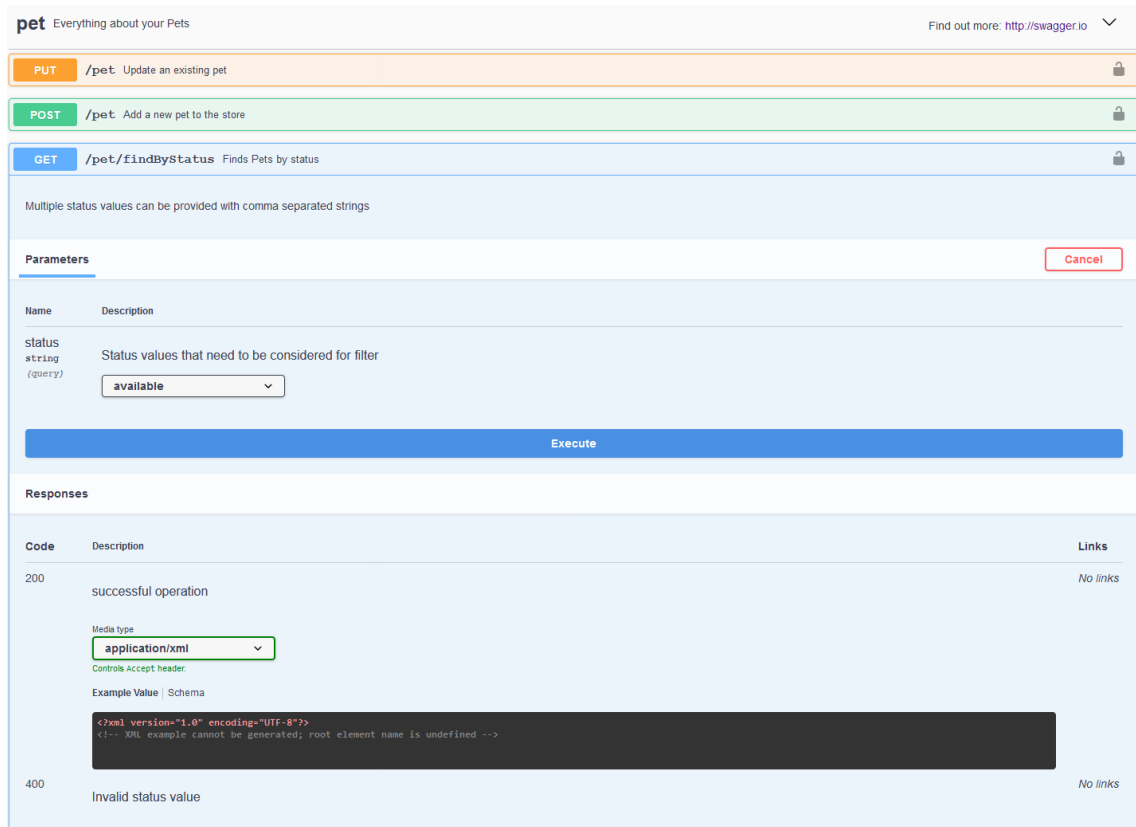



Figure 3.1: A snapshot of the Swagger pet store and its interactive UI.

3.3 Evaluation

The aim of the evaluation phase is to measure the actual coverage of the tests produced. In this work the only criteria used for the tests is code coverage, meaning that if a certain aspect of the REST system is interacted with, it will be considered tested. Therefore the way the system is assessed is that the tests will look at the output coverage or more specifically, the status codes produced when making certain request. If an operation is successful it will give a 200 response and if not, a 404 might be produced for example. As discussed in the theory, in chapter 2, the status codes can be used as an output coverage criteria. If nothing else is stated by the test case itself, then 2xx status codes can be considered correct and 4xx or 5xx is considered to be erroneous. It should be noted that some of the tests will be generated to fail, and therefore a 4xx result in that case could prove to be correct.

The tests will be performing CRUD operations against the routes that are declared in the OAS specification and evaluating each of the responses from the server. Tests will be built by parsing the generated JSON file with the help of JavaScript and then using Mocha and Chai to perform the requests. All of the generated tests will contain the needed parameters to make the request and different test cases will modify these parameters to test different scenarios. One example could be sending a null value instead of an integer. This will result in different responses depending on if the operation is allowed or not. In the pet store example, retrieving a certain pet could result in either the pet not existing (404 NOT FOUND) or successfully retrieving the pet (200 OK) for example. In the specification, the expected response for each route is defined. Thus making it possible to see which responses that the server can output and which can be accounted for when testing each route.



TCL	Input criteria	Output criteria
0		
1	Paths	
2	Operations	
3	Content-type	Content-type
4	Parameters	Status code classes
5		Status codes
6		Response body properties
7	Operation flows	

Figure 3.2: Test Coverage Model: Levels & Criteria.

The system is also evaluated to see the effectiveness of the test generation, and that all the tests that are specified from Tcases are generated properly in their JSON format. This is to ensure that they can be accurately generated and actually made executable. Other areas such as scaling and optimization can also be commented briefly on but won't be the main subject of investigation in this report.



4 Results

This chapter will present the results of the carried out method. It will go through and motivate the choice of the tool being used, the actual usage of said tool and also the outcome of the testing.

4.1 Pre-Study

In the pre-study phase, different tools were investigated and evaluated in order to see if they fit the criterion that were given. The choice fell on Tcases since it provided a method to automatically generate ready tests from the given specification. The remaining work to be done consisted of making the tests executable which was made possible since the generated tests were given in a JSON format. The documentation of Tcases was also determined to consist of enough information to be able to properly understand it. Finally it was also considered that Tcases was open source, making its source code available as well as being free of usage. Tcases also provided the possibility to generate more customised test cases since making the test cases executable requires some work done by the user in order to make them fully functional. This could therefore open up the possibility to use other testing tools when generating the tests, to fit the needs of the test suites being built.

Once the tool to work with had been chosen, the pre-study phase would go over to focus on reading the documentation of how tcases worked together with OAS and JSON. This would explain how Tcaess generated its tests in order to provide a high coverage of the system models. The input spaces for every request defined in the OpenAPI spec will be covered in the way that there are variations so that every input parameter is used at least one time. This should include both valid and also invalid values which are expected to give a faulty response. The method that is used here is a sort of combinatorial testing. The default for Tcases is 1-tuple coverage and will translate the system inputs into a minimal set of test cases which uses every value of every variable at least once, while still satisfying all the constraints.

For the requirements to be able to run the test suites against a server, the OpenAPI projects own server was used and was able to provide a sample API which described a pet shop. This sample API was also described in a OAS format which could be used in order to generate tests from Tcases. The described endpoints were checked to see if they existed and provided the described responses first via their web-based api, which can be seen in figure 3.1, and also

using the testing tool Postman. This verified that the output was identical to the one described in the OAS. As described in the method chapter, the returned data that was considered to be of most value when evaluating the coverage was status codes.

4.2 Implementation

The implementation phase consisted of constructing tests with the help of Tcases, whom generated the tests from the given OpenAPI specification, and then make the tests executable. The tests were written using JavaScript and the work consisted of parsing the produced JSON file in order to set up the tests.

The routes in the given OAS that were tested, consisted of POST requests and GET requests. There were also PUT and DELETE operations but since they in this scheme functioned in the same way as the POST and GET requests, they were not implemented in order to save time for actually evaluating and focusing on generating the tests. In table 4.1 and 4.2 the total number of testcases for each path can be seen. In some instances for the POST request, the body would be in the form of "application-xml" which were discarded since for the tests in this work, the body would only be in JSON format when the requests were sent. This is motivated by the fact that testing in different formats is not of interest in this work and it is instead the generation of tests that is relevant. The test cases would still perform the same task in each format. This is a reason for a higher number of tests for the POST requests and not the actual number of tests that were run.

Table 4.1: Tests for POST requests

Test resource	Number of tests
POST_pet	112
POST_pet-petId	10
POST_store-order	65
POST_user	71
POST_user-createWithList	27

Table 4.2: Tests for GET requests

Test resource	Number of tests
GET_pet-findByStatus	7
GET_pet-findByTags	9
GET_pet-petId	6
GET_store-order-orderId	6
GET_user-login	7
GET_user-username	5

The parsed tests were used together with Mocha and Chai to send requests to the pet store server and assert the resulting status code. Mocha managed to build the actual test structure and Chai was used to set up the headers, content-type and also assert the response that was to be expected.

4.2.1 Parsing JSON file

The JSON file produced by Tcases was structured in a certain way so that the tests would have to be extracted and built before being able to be executed. The JSON file defined the parameters in a certain format and the parameters and their values could be extracted with the help of the following keywords:

- Defined
 - The parameters of the request are first defined whether it is used or not. If it is not used it will be defined as "NA". The defined part also defines if the test should fail, due to an erroneous value being sent for example.
- Type
 - The type of the parameter that is being used.
- Value
 - The actual value of the parameter. Can contain "Value.Is" which states a value that the parameter should have. "Value.length" which states the length of the parameter, if it is a string or an array for example. It can also contain "Value.Properties", in the case of an object, which defines the different properties that the object parameter contains.

These were consistent for the parameters that a test were supposed to contain.

GET Tests

The GET requests send the information needed primarily via the URL either with a single value such as an ID for example or by using a query with several values. One of the generated test cases for a GET request can be seen in listing 4.1. The test case sets up a query with the value status. It defines the type of value it has and also the actual value itself. This was iterated through and set up together with the URL so that the GET request could be made against the server with the values that the test case contained.

```

1      {
2          "id": 0,
3          "has": {
4              "path": "/pet/findByStatus",
5              "server": "/v3",
6              "operation": "GET",
7              "version": "1.0.6-SNAPSHOT",
8              "properties": "status,statusValue"
9          },
10         "query": {
11             "status.Defined": {
12                 "has": {
13                     "style": "form"
14                 },
15                 "value": "Yes"
16             },
17             "status.Type": {
18                 "value": "string"
19             },
20             "status.Value": {
21                 "has": {
22                     "default": "available"
23                 },
24                 "value": "available"
25             }
26         }
27     }

```

Listing 4.1: A generated test case for a GET request

POST Tests

The post tests would contain more values since the data could be sent via a body and also added at the path the same way as GET requests functioned. This resulted in more testcases as previously mentioned and in most cases, more values that could be sent. The format for the POST tests would consist of a larger format as can be seen in listing 4.2. Here the test case first defines a body of the request which contains parameters that are being sent. The values could be different types of data values, integer, chars etc, as in the GET requests. But

they could also consist of containers such as arrays and objects which needed to be handled and set up in order to execute the test.

```

1      {
2          "id": 3,
3          "has": {
4              "path": "/user",
5              "server": "http://petstore.swagger.io/v2",
6              "operation": "POST",
7              "version": "1.0.0",
8              "properties": "application-json, application-jsonEmail, application-jsonEmailValue, application-
                jsonFirstName, application-jsonFirstNameValue, application-jsonId, application-jsonIdValue
                , application-jsonLastName, application-jsonLastNameValue, application-jsonPassword,
                application-jsonPasswordValue, application-jsonPhone, application-jsonPhoneValue,
                application-jsonProperties, application-jsonUsername, application-jsonUsernameValue,
                application-jsonUserStatus, application-jsonUserStatusValue, application-jsonValue,
                Content"
9          },
10         "request": {
11             "Body.Defined": {
12                 "value": "Yes"
13             },
14             "Body.MediaType": {
15                 "has": {
16                     "mediaType": "application/json"
17                 },
18                 "value": "application-json"
19             },
20             "Body.application-json.Type": {
21                 "value": "object"
22             },
23             "Body.application-json.Value.Property-Count": {
24                 "value": "> 0"
25             },
26             "Body.application-json.Value.Properties.id.Defined": {
27                 "value": "Yes"
28             },
29             "Body.application-json.Value.Properties.id.Type": {
30                 "value": "integer"
31             },
32             "Body.application-json.Value.Properties.id.Value.Is": {
33                 "has": {
34                     "format": "int64"
35                 },
36                 "value": "> 0"
37             },
38             "Body.application-json.Value.Properties.username.Defined": {
39                 "value": "Yes"
40             },
41             "Body.application-json.Value.Properties.username.Type": {
42                 "value": "string"
43             },
44             "Body.application-json.Value.Properties.username.Value.Length": {
45                 "value": "0"
46             },
47             "Body.application-json.Value.Properties.firstName.Defined": {
48                 "value": "Yes"
49             },
50             "Body.application-json.Value.Properties.firstName.Type": {
51                 "value": "string"
52             },
53             "Body.application-json.Value.Properties.firstName.Value.Length": {
54                 "value": "0"
55             },
56             "Body.application-json.Value.Properties.lastName.Defined": {
57                 "value": "Yes"
58             },
59             "Body.application-json.Value.Properties.lastName.Type": {
60                 "value": "string"
61             },
62             "Body.application-json.Value.Properties.lastName.Value.Length": {
63                 "value": "0"
64             },
65             "Body.application-json.Value.Properties.email.Defined": {
66                 "value": "Yes"
67             },
68             "Body.application-json.Value.Properties.email.Type": {
69                 "value": "string"
70             },
71             "Body.application-json.Value.Properties.email.Value.Length": {
72                 "value": "0"
73             },
74             "Body.application-json.Value.Properties.password.Defined": {
75                 "value": "Yes"
76             },
77             "Body.application-json.Value.Properties.password.Type": {
78                 "value": "string"
79             },
80             "Body.application-json.Value.Properties.password.Value.Length": {

```

```

81         "value": "0"
82     },
83     "Body.application-json.Value.Properties.phone.Defined": {
84         "value": "Yes"
85     },
86     "Body.application-json.Value.Properties.phone.Type": {
87         "value": "string"
88     },
89     "Body.application-json.Value.Properties.phone.Value.Length": {
90         "value": "0"
91     },
92     "Body.application-json.Value.Properties.userStatus.Defined": {
93         "value": "Yes"
94     },
95     "Body.application-json.Value.Properties.userStatus.Type": {
96         "value": "integer"
97     },
98     "Body.application-json.Value.Properties.userStatus.Value.Is": {
99         "has": {
100             "format": "int32"
101         },
102         "value": "0"
103     },
104     "Body.application-json.Value.Properties.Additional": {
105         "value": "Yes"
106     }
107 }
108

```

Listing 4.2: A generated test case for a POST request

Notable is that in some cases the "value" of a certain parameter is not always set. If the Open API specification defines a value to be "more than one" then there will be generated tests that will define this in a general way. This is done by setting the value to "> 1" for example. Tests can also be generated as a variant of the same test where the value for instance is instead "< 1". In order to perform these tests the chosen solution was to choose a random value between the given value and 10 or -10, these values were chosen simply for keeping it trivial and being able to make the tests executable.

```

1  function assignIntegerValue(value) {
2      var parameterValue
3
4      if (value == '1') {
5          parameterValue = 1;
6      }
7      else if (value == "> 0" || value == "> 1") {
8          parameterValue = Math.ceil(Math.random() * 10)
9      }
10     else if (value == "< 0") {
11         parameterValue = -Math.ceil(Math.random() * 10)
12     }
13     else if (value == '< 1' || value == '0') {
14         parameterValue = 0
15     }
16     else {
17         parameterValue = value
18     }
19
20     return parameterValue
21 }

```

Listing 4.3: Function to generate a random integer

There was also the case when the "type" could be defined as a "string" and the length of the string was set to "> 1". Then the process would be similar with the only difference that a random string would be generated with the length being chosen at random. The maximum length of strings here were also set to 10.

```

1  function assignStringValue(value) {
2
3      if (value == "0") {
4          length = 0;
5          string = generateString(length)
6      }
7      else if (value == "> 0") {
8          length = Math.ceil(Math.random() * 10)
9          string = generateString(length)
10     }
11     else if (value == "1") {

```

```

12     length = 1;
13     string = generateString(length);
14 }
15 else if (value == "> 1") {
16     length = Math.ceil(Math.random() * 10)
17     string = generateString(length)
18 }
19 else if (value == "< 1") {
20     length = -Math.ceil(Math.random())
21     string = generateString(length)
22 }
23 else if (typeof(value) == "string") {
24     string = value
25 }
26 else {
27     string = ""
28 }
29 return string
30 }

```

Listing 4.4: Function used in order to assign a random string

```

1 function generateString(length) {
2     var result = '';
3     var characters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789';
4     var charactersLength = characters.length;
5     for (var i = 0; i < length; i++) {
6         result += characters.charAt(Math.floor(Math.random() * charactersLength));
7     }
8     return result;
9 }

```

Listing 4.5: Function to generate a random string

In the case where the parameter already had a pre-defined "value", this would simply be used instead. The possible value that a parameter can have are defined in the OpenAPI specification.

The parameters and their values were extracted and set up as a JSON body in an array in order to easier set up the requests that were being made. An example of what the information for test could look like can be seen in listing 4.6. Here all the information is given in order to set up and execute a test.

```

1 {
2   operation: 'POST',
3   path: '/pet',
4   shouldFail: true,
5   TestId: '38',
6   body: {
7     category: { id: -3, name: 0 },
8     name: 'ds0',
9     photoUrls: [ '' ],
10    status: 'pending'
11  }
12 }

```

Listing 4.6: An example of the extracted test

For each path there exists test that pass when the correct values are given, but in cases where parameters are excluded or changed to different values this can also cause the test to fail.

4.3 Test results

The tests managed to be executed properly with the requests being sent to the pet store server. In most cases the tests would fail, simply put return a 4xx response code due to some erroneous parameter being sent. For example the path /pet would run 112 POST tests in total where 84 would respond with the expected result. When the test cases that were supposed to result in a failure, Tcases currently does not provide a way of showing what the expected responses should be. Some of the tests generated will be defined with values expected to fail

by sending an erroneous value meaning that these tests should be expected to fail, however there is still no way of knowing which exact status code that the response will contain.

For example there can be a case where the test sends an erroneous value as a parameter. Then there exists both the possibility that there can be a server issue that will respond with a 5xx response and also that there can be a error in the request which could grant a 4xx response. This is not possible to differentiate from in the current state of Tcases.

This can be seen in the two tables 4.3 and 4.4 where the amount of successful tests can be seen. Successful in this case means a test where we received the same status code from the resulting operation, as we expected.

Table 4.3: Amount of successful tests for GET requests

Test resource	Successful/Total
GET_pet-findByStatus	6/7
GET_pet-findByTags	6/9
GET_pet-petId	6/6
GET_store-order-orderId	6/6
GET_user-login	3/7
GET_user-username	3/5

Table 4.4: Amount of successful tests for POST requests

Test resource	Number of tests
POST_pet	84/112
POST_pet-petId	5/10
POST_store-order	19/65
POST_user	54/71
POST_user-createWithList	20/27



5 Discussion

This chapter will discuss the results that were obtained during the work and evaluate these. The carried out method will also be discussed and criticized. Finally the work will also be discussed in a wider context.

5.1 Results

The test cases were generated as expected and could function well enough in order to execute the tests. Although there is an issue with expecting a response from the server since Tcases does not provide this. This makes the test cases incomplete since they will not be able to actually give any valuable info regarding if the system has properly been tested or not. In this case the status codes is what is evaluated when considering coverage of the tests but it is still hard to determine to what degree the actual code coverage results in, at least practically in this case. Although it can in theory be able to cover different instances and achieve high coverage due to the fact that Tcases follows the method practiced in combinatorial testing, mentioned in chapter 2. The specification will regardless determine in what ways the different paths can be tested in but it was deemed hard in this work to actually determine the coverage provided by Tcases, or at least confirm that it was high enough to be of significance.

One can also note the amount of test cases that were generated, which might be considered ambiguous in some cases due to the fact that they might be testing cases which are not needed. This is of course determined by one's own needs when testing and some cases might have to use all the input variables available. Tcases also provides different options for managing the amount of tests, but this is not further investigated in this report. It can be noted however, that the pet store example that is used here is not of very large scale compared to other systems and generating test cases for a much larger specification might have different results.

This work only used the pet store API and its OpenAPI specification in order to generate the tests and make them executable. If a different OpenAPI specification would have been used the resulting test cases that were generated, would of course have been different. One could note that it would be of interest to run Tcases with different OpenAPI specifications but this was disregarded in this work due to lack of time.

5.2 Method

The objective of this work was to investigate a method for automating testing for REST APIs, given a OpenAPI specification. In the beginning of the work, it was considered wheather to use a framework to perform the test generation or to implement such a functionality from scratch. It was however, quite soon discovered that doing so would be far too much work and there would not be time to implement all of these functionalities properly in order to generate and actually execute the tests. Therefore the choice to use an already existing tool seems to have been better in terms of time management. There would still be a large amount of time spent on actually making the tests, which where generated from Tcases, executable.

The pre-study phase would consist of both going through available tools and also to try to immerse one self in how the chosen one, Tcases in this case, would function and work. This would mean that there would not be enough time to make a full scale evaluation of different OpenAPI specification tools. The focus on Tcases came from reading the information that was available on the Github repository as well as the information that was specified on the official site. There were other tools and frameworks that where briefly considered but since they where only vaguely read upon, they where not brought up in this work.

Parsing and constructing the tests in order to make them executable would take longer time than expected. Mostly due to the fact that the format of the JSON file would at times be hard to understand and was at a start not clear how to go forward. This can of course be attributed to the programmer rather than the lack of documentation and a more thorough read through of the existing documentation could have been needed. In the end the extraction and setup of the tests was successful even if it took a longer amount of time than was first expected.

The idea of measuring the coverage by looking at the status seemed like a reasonable solution in theory but it is hard to assess how well it could have worked in practice. Tcases states in its documentation that it has no way of determine the responses yet, but the idea was that it would still be possible to make out successful test cases from the status codes that where returned. If more time had been spent on developing some way of detecting when a certain status code would be given, then it is possible that it might have yielded better results but as of this work, there was not enough time to focus on this.

While the work was ongoing, Tcases would release a new version of the software which actually generated tests that where executable as well. Since the work of parsing the JSON file was already started this was not considered in this report but if the same work was to be done again an interesting approach could be to look into these executable tests and save time from actually having to parse and make the tests executable by hand. Then more time could be spent on the evaluation part of the thesis and could possibly discover other findings.

5.2.1 Replicability, Reliability and Validity

By using Tcases in this report it should be able to generate test cases that can be parsed as has been done in this work. The generated tests will of course be different if a different OpenAPI specification is used as discussed previously. Using the same specification should however, lead to similar results even though the actual implementation part of making the tests executable can be done in with different languages or using other testing tools for example. The actual result received is that the tests can be generated but they do not at the current stage actually perform valuable testing to any degree. Therefore the actual coverage is hard to measure in practice and get a confirmation on, as has previously been discussed.

5.2.2 Source criticism

Most of the sources for REST APIs cite the work of Fielding [7] and this is also the primary source for reading about RESTful systems since Fielding is the one who coined the term.

When searching for information about REST the sources that have been cited the most and are also citing Fielding have been considered first. For the information about Tcases mostly the documentation that exists on Tcases site and their JSON guide [23] has been used as well as their own Github repository [22]. Overall automating testing of RESTful systems is a subject which calls for more studies and for Tcases there have not been any excessive amount of research done as of yet, at least not in the manner of which it is using the OpenAPI specification.

5.3 The work in a wider context

A large part of software production involves testing and the more of this that can be automated the more time can be saved. Although it is still at an early stage regarding automatic testing of RESTful systems, it can still be an important step to take in the right the direction in order to automate the process as much as is possible. From a subjective point of view it is also the experience that testing is in fact not a very favorable step in software development and can in many cases be skipped. Since it can still be a very vital part of the process it can however still be argued that it is needed and automating the process could potentially contribute to it being done more often.



6 Conclusion

The purpose of the work carried out was to investigate how tests could be automatically generated for a REST api. The REST API was given in the format of a OpenAPI specification and the evaluation of the tests was to be done using code coverage as a metric. The research question that was set to answer was the following:

- How can automated tests, given an OpenAPI specification of a RESTful API, be implemented in order to provide high test coverage of the specified system?

This was done by first investigating different tools and frameworks that worked together with OpenAPI specifications and which could aid in the generation of the tests. This way it could rid the need to have to write the whole program from scratch. The selected tool would come to be Tcases which could generate test cases for an OpenAPI spec in a JSON-format. The JSON-file only described how the test cases were to be set up and were not directly executable. This was left for the programmer to implement.

The test cases were set up and made executable and were tested against a pet store API which was provided by the OpenAPI community. The part that was missing from Tcases was a way to predict the response from the test cases that were produced. This led to the fact that even if the tests were possible to make executable, they were not able to function properly enough in order to measure any coverage in practice. The tests could provide information in some cases when they were guaranteed to fail, but this was not enough to predict the actual response which was given in the form of status codes. The results show that it is possible to generate test cases for a REST API that is given in the form of an OpenAPI specification, but there is still more work to be done in order to achieve actual coverage of the system that is valuable.

6.1 Future work

For future implementations it would of course be of interest to further improve Tcases, which is an open source project, in order to properly define the responses of the defined system. Then it could be possible to further evaluate and possibly achieve a high coverage of the tested system. Adding to this it could also be interesting to look into if there is a way to complement Tcases so that the responses can be known.

Another interesting aspect that could be investigated regarding Tcases, is evaluating the latest release. Since the work began, Tcases released a new version which allows the generated tests to be executed directly. Looking into this further could be of further interest and might assist in finding a way to evaluate the tests generated by Tcases. This could extend to also comparing the Open source alternatives, such as Tcases, to the other tools that exist for handling test case generation and see how they compare in regards of both test coverage and also other criteria which could be of interest.



Bibliography

- [1] Anneliese A Andrews, Jeff Offutt, and Roger T Alexander. “Testing web applications by modeling with FSMs”. In: *Software & Systems Modeling* 4.3 (2005), pp. 326–345.
- [2] Andrea Arcuri. “RESTful API automated test case generation”. In: *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE. 2017, pp. 9–20.
- [3] Chai. <https://www.chaijs.com/>. 2020.
- [4] Sujit Kumar Chakrabarti and Reswin Rodriquez. “Connectedness testing of restful web-services”. In: *Proceedings of the 3rd India software engineering conference*. 2010, pp. 143–152.
- [5] Fernando Doglio. *Pro REST API Development with Node.js*. Apress, 2015.
- [6] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. “Automatic generation of test cases for REST APIs: a specification-based approach”. In: *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE. 2018, pp. 181–190.
- [7] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine Irvine, 2000.
- [8] Laura Inozemtseva and Reid Holmes. “Coverage is not strongly correlated with test suite effectiveness”. In: *Proceedings of the 36th international conference on software engineering*. 2014, pp. 435–445.
- [9] Laura Inozemtseva and Reid Holmes. “Coverage is not strongly correlated with test suite effectiveness”. In: *Proceedings of the 36th international conference on software engineering*. 2014, pp. 435–445.
- [10] Stefan Karlsson, Adnan Causevic, and Daniel Sundmark. “QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs”. In: *arXiv preprint arXiv:1912.09686* (2019).
- [11] Pablo Lamela Seijas, Huiqing Li, and Simon Thompson. “Towards property-based testing of RESTful web services”. In: *Proceedings of the twelfth ACM SIGPLAN workshop on Erlang*. 2013, pp. 77–78.
- [12] Philip A Laplante. *What every engineer should know about software engineering*. CRC Press, 2007.

-
- [13] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. “A catalogue of inter-parameter dependencies in restful web APIs”. In: *International Conference on Service-Oriented Computing*. Springer. 2019, pp. 399–414.
 - [14] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. “Test coverage criteria for RESTful web APIs”. In: *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 2019, pp. 15–21.
 - [15] Mocha. <https://mochajs.org/>. 2020.
 - [16] Glenford J Myers. *The art of software testing*. John Wiley & Sons, 2006.
 - [17] Changhai Nie and Hareton Leung. “A survey of combinatorial testing”. In: *ACM Computing Surveys (CSUR)* 43.2 (2011), pp. 1–29.
 - [18] Node.js. <https://nodejs.org/en/>. 2020.
 - [19] OpenAPI specification. <https://openapi.tools/>.
 - [20] Ron Ratovsky. *OpenAPI Specification*. <https://github.com/OAI/OpenAPI-Specification.git>. 2020.
 - [21] Alex Rodriguez. “Restful web services: The basics”. In: *IBM developerWorks* 33 (2008), p. 18.
 - [22] Tcases. <https://github.com/Cornutum/tcases>. 2020.
 - [23] Tcases: The JSON Guide. <http://www.cornutum.org/tcases/docs/Tcases-Json.htm>. 2020.