RAML-based Mock Service Generator for Microservice Applications Testing

Nikita Ashikhmin[1], Gleb Radchenko[1](✉) and Andrei Tchernykh[1,2]

[1] South Ural State University, Chelyabinsk, Russia
ashikhminna@pvc.susu.ac.ru, gleb.radchenko@susu.ru
[2] CICESE Research Center, Ensenada, México
chernykh@cicese.mx

Abstract. The automation capabilities and flexibility of computing resource scaling in cloud environments require novel approaches to application design. The microservice architectural style, which has been actively developing in recent years, is an approach to design a single application as a suite of small services. Continuous integration approach demands transition from manual testing methods to fully automated methods. The mocking is one of the methods to simplify development and testing of microservice applications. The mock service can be considered as an extension of mock object concept. It simulates the behavior of a web service based on a description of its interface. However, developers need to spend additional efforts on development and support of these mock services. We propose a method that would make it easier to generate mocks for REST services by using RAML specifications of services. Using this approach, we propose an implementation, which provides mock services generation and deployment as Docker containers.

Keywords: Microservice · Testing · Docker · REST · RAML · Mocking container.

1 Introduction

The microservice model describes a cloud application as a suite of small independent services, each running in its container and communicating with other services using lightweight mechanisms. In [1], the following features of microservices are defined:

- *Open Interface* – microservice should provide an open description of interfaces and communication messages format (either API or GUI).
- *Specialization* – each microservice provides support for an independent part of application's business logic.
- *Containerization* – isolation from the execution environment and other microservices, based on a container virtualization approach. Technologies like OpenVZ, Docker and Rocket [2] became de-facto standards for implementation of such an approach.

- *Autonomy* – microservices can be developed, tested, deployed, destroyed, moved and duplicated independently and automatically. Continuous integration is the only option to deal with such a development and deployment complexity.

The complex structure of microservice applications demands that microservices should be independently deployable by fully automated machinery. Continuous integration approach demands transition from manual testing methods to fully automated methods. Newman in [3] describes following three levels of testing of microservice applications:

- *Unit tests* that typically validate a single function or method call.
- *Service tests* that are designed to test individual capabilities of isolated services.
- *End-to-End tests* verify the correctness of an entire system in its integrity.

End-to-End tests cover production codes and provide confidence that the application will behave correctly in the production environment. On the other hand, the feedback time of End-to-End Tests is significant. Finally, when such a test fails, it can be hard to determine which unit has broken.

To simplify and speed up the testing process, the developer must isolate the test of an individual service from the entire system. On the other hand, service testing will not be completed without testing its interaction with other services. To simulate the behavior of the other services in controlled ways developers use the so-called «test doubles», namely mocks.

Test Double is a generic term for any case where one replaces a production object for testing purposes. In [4], the following types of test doubles are defined:

- *Dummy objects* are objects without implemented functionality.
- *Fake objects* provide all the functionality needed by the consumer objects, but not suitable for production implementations because of some limitations in speed or effectiveness.
- *Stubs* provide canned answers to the method calls.
- *Mocks* are pre-programmed objects, which generate answers for method calls, corresponding with the interface specification.

Mock services can be used in the following cases [5]:

- *Development* – at the beginning of the development, we define protocols of the communication between services. Mock services can imitate the behavior of services that has not been implemented. This approach can provide a solution to such a «Chicken or the egg» problem when we need to develop a service which is communicating with such unimplemented services.
- *Testing* – mock services allow testing each service individually in isolation from others. It reduces time and resources required for testing. Additionally, mock services reduce the test coverage to one specific service. It helps developers to find broken functionality faster.

REST [6] is one of the most common approaches for microservice interface implementation. However, REST does not define a standard way for the interface documentation. It requires developers to provide additional information about all endpoints and call parameters using third party methods. There are two design patterns of REST interface specifications. Top-down specifications determine the behavior of the REST service independently of its implementation. On the other hand, bottom-up specifications describe the interface of the REST service based on its source code, and cannot be created independently.

We highlight three popular methods for REST interface specifications description [7].

- *SWAGGER* [8] – is a format and framework for the definition of RESTful APIs. It is used to generate server-side API code, client code, and API documentation. SWAGGER is designed as the *bottom-up specification*.
- *RAML (RESTful API Modeling Language)* [9] – is a REST-oriented non-proprietary, vendor-neutral open top-down specification language based on YAML. It focuses on the description of resources, methods, parameters, responses, media types, and other HTTP constructs. It has user-friendly syntax and is contributed by many companies, like Cisco and VMware.
- *API Blueprint* [10] – is a top-down API specification language for web APIs, based on the markdown [11] format. It requires third party server codes and specifically focuses on C++.

The aim of this work is to describe the architecture and implementation of the system, which would provide generation of mock services based on the RAML specification in the form of deploy-ready Docker containers.

We choose the RAML language for several reasons:

- this specification format is human-readable because it based on YAML language;
- it has a big community;
- it is a top-down specification so that users can generate mock service before the development of the real one.

2 Related Work

There are several systems that support the generation of mock services. Some of them allow automatic mock services generation based on an interface specification, while others use special types of "request-response" configuration to emulate service behavior.

Mountebank [12] is an open source tool, which provides cross-platform, multi-protocol test doubles for network services. An application, which is supposed to be tested, should point to the IP or URL of a Mountebank instance instead of the real dependency.

Mountebank supports HTTP, HTTPS, TCP and SMTP protocols. To define the behavior of the network service, Mountebank requires a configuration, where all request and response messages for the services are specified.

SoapUI [13] is an open-source web service testing application for service-oriented architecture (SOA) and representational state transfer (REST) applications. SoapUI can generate SOAP mock service based on WSDL specification, while REST mock services must be configured by Groovy scripts.

API Designer [14] is an application that provides a web-based graphical environment for design, documentation, and testing of APIs in a web browser. API Designer creates REST mock service using RAML specification. However, this service cannot be used by Continuous Integration systems because the generated mock services disappear when the user closes the application.

All solutions discussed above do not create deploy-ready mock services. These applications make REST mock service creation easy, but they do not make this process fully automated. Our approach generates the REST mock service based just on the RAML specification file. Furthermore, our approach would support delivery of mock services as Docker containers.

3 Mock Service Generator Requirements

The mock service generator has one functional and two nonfunctional requirements. The functional requirement is the ability to generate mock services. Nonfunctional requirements are the ability to get the file with RAML specification v0.8 as input and return created microservices as Docker images.

Use case diagram is shown in Figure 1.

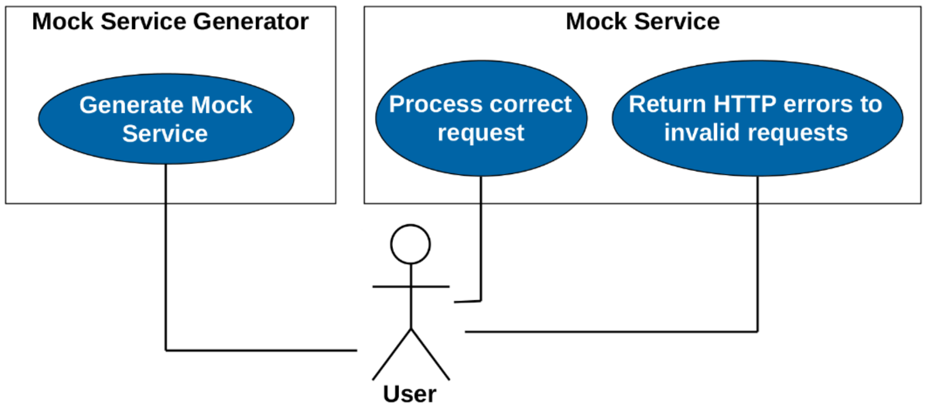

Fig. 1. The mock service generation system Use Case diagram.

Docker [15] is a lightweight mechanism, allowing to run pre-configured system images. Docker represents an implementation of container technology that is considered as an alternative to complete virtualization approach, providing a well-defined application execution environment at the operating system level. Instead of starting a complete operating system on top of a host system or a hypervisor, a container shares the kernel with the host system, which largely eliminates overheads while maintaining isolation between applications. Docker container wraps up a service inside isolated filesystem together with all required system libraries.

The mock service, created by the mock service generator, must satisfy the following functional requirements:

- process correctly GET, POST, PUT, and DELETE requests and return valid responses based on RAML specification of the service;
- return appropriate errors for all incorrect requests.

Furthermore, nonfunctional requirements for created mock services are:

- return responses based on response body examples or response body JSON schema;
- be packed inside a single Docker image.

4 System Architecture

Our system consists of two subsystems – the mock service generator and mock services. The communications between them are shown in Figure 2.

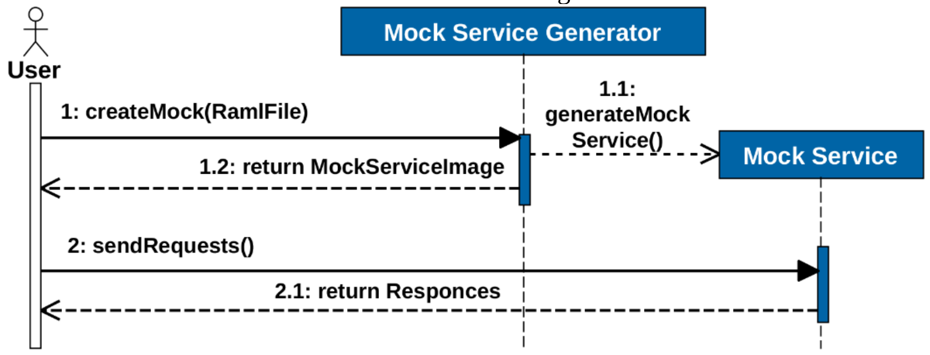

Fig. 2. Sequence diagram of mock service generation and usage.

Mock Service Generator processes user commands to generate mock services. The user can be represented by a continuous integration system that uses the mock service generator to implement testing procedures. The mock service generator processes the createMock request with one argument – a link to an RAML file that describes the interface of the service, endpoints, the format of valid requests, and expected responses for these requests (see Figure 3).

The generator validates this file and sends the user an error message if the RAML file is incorrect. As a result, the mock service generator creates a Docker container that contains a template of mock service with the received RAML specification file and sends a link to this file to the user.

Mock Service is a service generated by the mock service generator. All mock services have the same architecture, and its behavior depends only on the RAML specification of the service. We define four components in the mock service architecture: *Gateway*, *Path resolver, Request validator, and Response generator* (see Figure 4).

```
/employees:
  get:
    queryParameters:
      department:
    responses:
      200:
        body:
          application/json:
         example: |
           ...
  post:
  delete:
  /{employee}:
   get:
     responses:
       200:
         body:
          application/json:
            schema: |
            { "$schema": "http://json-
              schema.org/schema",
              "type": "object",
              "properties": {
                "fullName": { "type": "string",
                              "format": "fullname"},
                "department": { "type": "string",
                                "format": "word"}
                "email": { "type": "string",
                           "format": "email"}
              },
            }
```

Fig. 3. An example of REST service specification in an RAML format.

The *Gateway* implements the facade pattern and provides a single entry point for all user requests to the mock service. The Gateway receives HTTP-requests sent by a user and calls the Path resolver and Request validator to check the correctness of the received query. Further, if the results of query analysis conducted by these components indicate that the request corresponds to the RAML specification, the Gateway calls the Response generator to generate a body of the response. Finally, the Gateway forms the HTTP response and returns it to the client.

Path resolver. This component validates endpoints of requests based on the RAML-based specification of the service. The Path resolver determinates the correctness of requests endpoints.

The *Request validator* component responds for the validation of parameters of the request, based on the RAML specification of the service. The component checks the compliance of the received parameters with the limitations of the RAML specification. The example of an RAML description of parameters is shown in Figure 5.

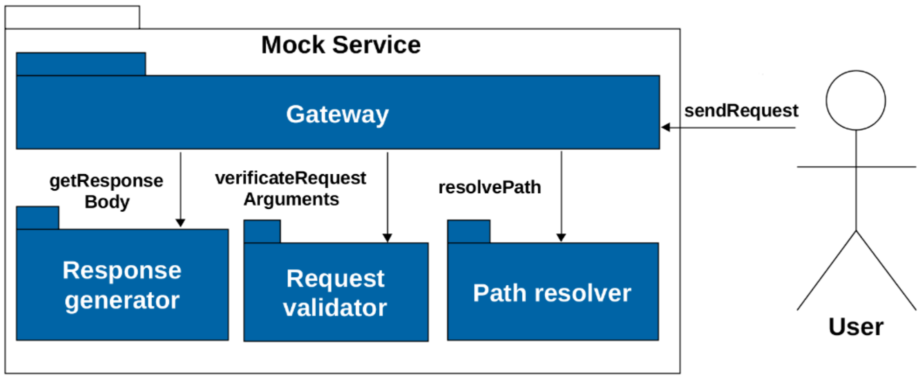

Fig. 4. The architecture of the Mock Service.

The *Response generator* component generates a body of the response for the request. The RAML language provides two ways to describe the response body. The first way is to declare an example of valid response in JSON format. The second way is to specify a JSON Schema [16], a special format that allows defining the structure of JSON documents. The body generation component to generate the response body by JSON Schema if JSON Schema exists. Otherwise, the component returns an example of response specified in the RAML file.

```
/{documentId}
  uriParameters:
    id:
      description: document identification number
      type: string
      minLength: 20
      pattern: ^[a-zA-Z]{2}\-[0-9a-zA-Z]{3}\-\d{2}$
```

Fig. 5. An example of the RAML definition of parameters of a request.

This component uses Elizabeth library [17] for generation dummy human-readable data. Users can define "format" parameter for string and use one of following integrated formats: *ipv4/ipv6, email, URI, date, time, name, username, surname, word* (Figure 3).

5 System Implementation

5.1 Mock Service Generator

Mock service generator is a standalone command line application that creates Docker images of mock services. It consists of Python script that generates containers, and operates according to the following procedure:

```
FROM python:3-onbuild
EXPOSE 5000
ENV PYTHONPATH .
CMD ["python", "./server/main.py"]
```

Fig. 6. The Docker file for mock service.

- the user runs the generator with the following parameters: the link to RAML file, and the name of resulting Docker container;
- mock service generator creates a temporary folder and copies the template mock services files;
- generator downloads the RAML specification file by the user's link, and adds it to the folder;
- the app generates a Docker file (see Figure 6);
- the app runs Docker build command that creates the Docker container. This container includes the Python interpreter, all required libraries and isolates the mock service from the other system;
- mock service generator returns a link to the container to the user.

5.2 Mock service

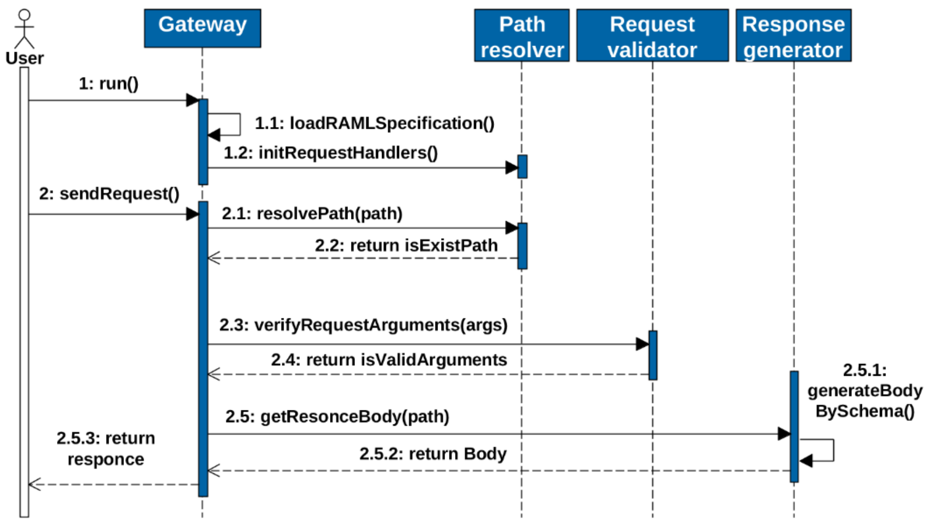
User
Gateway
Path resolver
Request validator
Response generator
1: run()
1.1: loadRAMLSpecification()
1.2: initRequestHandlers()
2: sendRequest()
2.1: resolvePath(path)
2.2: return isExistPath
2.3: verifyRequestArguments(args)
2.4: return isValidArguments
2.5.1: generateBody BySchema()
2.5: getResonceBody(path)
2.5.3: return responce
2.5.2: return Body

Fig. 7. Mock service user interaction sequence diagram.

The mock service is a web service based on the Flask framework [18]. The processing of user requests by the mock service is shown in Figure 7.

The source code of all mock services generated by the generator is identical. The behavior of mock services depends only on RAML specification file loaded on startup. The mock service uses ramlification [19] to parse RAML specification into Python objects. Currently, mock services support RAML v0.8.

At the startup of the service, the Gateway parses the RAML specification file and bind all routers to one of four functions that will handle GET, POST, PUT, and DELETE requests. The part of this code is shown in Figure 8.

```
def _init_url_rules(self):
  for resource in self.parser.resources:
    self.app.add_url_rule(
     rule=self._transrofm_path_raml_to_flask(resource.path),
     endpoint=resource.name,
     view_func=self.get,
     methods=['GET']
```

Fig. 8. Initialization of GET request handlers.

The request validator is implemented as a set of separated functions that validate parameters. The example of a validation function is shown in Figure 9.

```
def validate_string(self, value, param):
  if param.min_length and param.min_len > len(value):
         raise Exception(message.ERR_STRING_MIN)
  if param.max_length and param.max_len < len(value):
            raise Exception(message.ERR_STRING_MAX)
```

Fig. 9. An example of validation function.

There are five main function implemented in the Gateway component: initialization function and four functions that handle HTTP requests. The implementation of the GET function is shown in Figure 10.

```
def get(route, **kwargs):
    endpoint = Resolver.get_endpoint(route)
    Validator.validate_params(endpoint, request.args)
    body = BodyGenerator.generate_body(endpoint,'get')
    header = getHeader(endpoint,
                        resources.query_params,'get')
    return header, body
```

Fig. 10. The implementation of GET requests handler.

To create a body of the response, the response generator parses the JSON Schema in the RAML file into a tree and performs a direct traversal of all the nodes in the tree, corresponding to the following procedure.

1. `get_node` function identifies the type of current node and call the special function for this JSON type (for instance, it can be a `get_array`, `get_object` or P`get_string` function);
2. functions for generation objects and arrays calls `get_node` functions for the node children to fill inner data (see Figure 11);
3. the function generates the JSON data with the constraints imposed by the JSON schema to the current node.

```
def get_array(self, node):
  n = utils.generate_int(self._array_min_count,
                         self._array_max_count)
  items = [self._get_node(node['items']) for _ in
                                         range(n)]
  return items
```

Fig. 11. The implementation of the get_array method.

6 Testing

Testing of mock service is conducted using the unit and end-to-end tests. Unit tests are developed using standard Python unit test framework. 40 Python unit tests have been developed to check the source code of the project.

To provide integration testing, we developed a series of tests that create a mock service by the RAML specification, and imitate mock service usage, sending a set of REST requests, and comparing received responses with expected ones (see Figure 12). The example of the request and the response to it is shown in Figure 13.

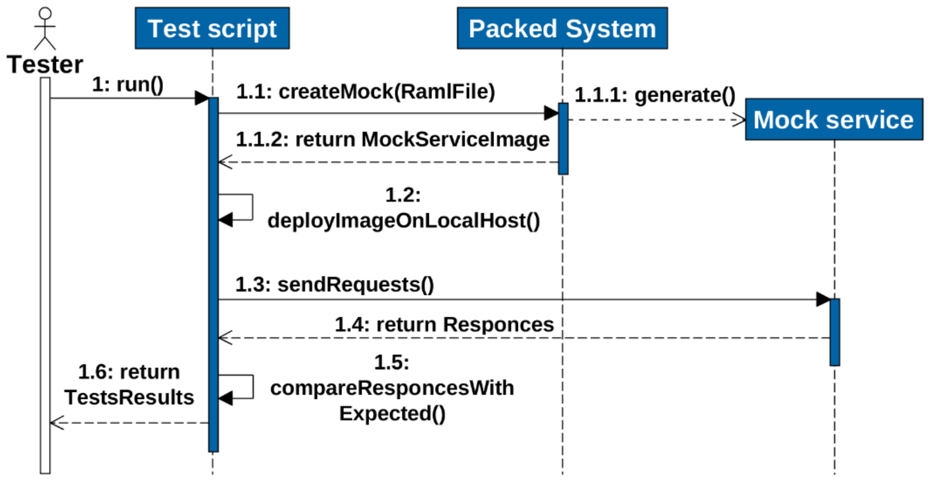

Fig. 12. Integration testing sequence diagram.

Finally, this system was integrated into continuous integration system of the Naumen Service Desk project [20]. This project includes about 200 Selenium tests for the Android application. During the process of testing, the mobile client sends a series of REST requests to the server. All tests without mock service are completed within 60-80 minutes in one node. After integrating the mock service, the time required for testing has decreased by about 35%.

```
request:
GET http://127.0.0.1:5000/employees/1
response:
{
"fullName":"John Smith",
"department": "wood",
"email": "smith-toronto-1765@gmail.com"
}
```

Fig. 13. An example of test request and response.

7 Conclusion

This article presents the design, architecture and implementation of the automatic mock service generation system. It provides generation of mock services based on the RAML specification in the form of deploy-ready Docker containers that considered as an alternative to complete virtualization approach providing a lightweight application execution environment. They share the kernel with the host system, which eliminates overheads while maintaining isolation between applications. We define four components in the mock service architecture: Gateway, Path resolver, Request validator, and Response generator. We describe the mock service generator algorithm. The developed system is tested with the unit and end-to-end tests. Services are verified for correct functioning in a real project. The source code of our application is available on our GitHub page [21].

8 Acknowledgment

The work was supported by the RFBR research project No. 15-29-07959 and by Act 211 Government of the Russian Federation, contract No. 02.A03.21.0011.

9 References

1. Savchenko, D., Radchenko, G.: Microservices Validation: Methodology and Implementation. In: 1st Ural Workshop on Parallel, Distributed, and Cloud Computing for Young Scientists (Ural-PDC 2015), vol. 1531, pp. 21-28. CEUR Workshop Proceedings, Yekaterinburg, Russia (2015).

2. Pahl, C.: Containerization and the PaaS Cloud. In: IEEE Cloud Computing, pp. 24–31 (2015).
3. Newman, S.: Learning Building Microservices. O'Reilly Media, Inc., Sebastopol, California, USA (2015).
4. Kim, T., Park, C., & Wu, C.: Mock object models for test driven development. In: Software Engineering Research, Management and Applications, 2006. Fourth International Conference on, IEEE, pp. 221-228 (2003).
5. Soltesz, S., Potzl, H., Fiuczynski, M., Bavier, A., and Peterson, L.: Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In: ACM SIGOPS Operating Systems Review., vol. 41, no. 3, pp. 275–287 (2007).
6. Fielding, R.: Representational state transfer. In: Architectural Styles and the Design of Netowork-based Software Architecture, pp. 76-85 (2000):
7. Haupt, F.: A Framework for the Structural Analysis of REST APIs. In: Software Architecture (ICSA), IEEE International Conference on, pp.55-58 (2017).
8. Cloves, C., Schmelmer, T.: Defining APIs. In: Microservices From Day One, pp.59-74. Apress, NY, USA (2016).
9. Surwase, I.: REST API Modeling Languages-A Developer's Perspective. In: IJSTE-International Journal of Science Technology & Engineering 2, vol. 10, pp. 634-637 (2016).
10. API Blueprint, http://apiblueprint.com, last accessed 2017/04/15.
11. Voegler, J., Bornschein, J., Weber, G.: Markdown – A Simple Syntax for Transcription of Accessible Study Materials. In: Miesenberger K., Fels D., Archambault D., Peňáz P., Zagler W, (eds.) Computers Helping People with Special Needs, ICCHP 2014, Lecture Notes in Computer Science, vol 8547, pp. 545-548. Springer, Cham (2014).
12. Mountebank - over the wire test doubles, http://www.mbtest.org, last accessed 2017/04/15.
13. Sana, A., Naji, M., A., Alsmadi, I.: Web services testing challenges and approaches. In: Proceedings of the 1st Taibah University International Conference on Computing and Information Technology, pp. 291-296 (2012).
14. Tsouroplis, R., Petychakis, M., Alvertis, I., Biliri, E., Askounis, D.: Community-based API Builder to manage APIs and their connections with Cloud-based Services. In: CAiSE Forum, pp. 17-23 (2015).
15. Merkel, D.: Docker: lightweight Linux containers for consistent development and deployment, In: Linux Journal, v.2014 n.239, pp. 2 (2014).
16. JSON Schema: syntax and semantics, http://cswr.github.io/JsonSchema/, last accessed 2017/04/15.
17. Grinberg, M.: Flask web development: developing web applications with Python. In: O'Reilly Media, Inc (2014).
18. Elizabeth, http://elizabeth.readthedocs.io/en/latest/, last accessed 2017/04/15.
19. Ramlification | Python parser for RAML, https://github.com/spotify/ramlfications, last accessed 2017/04/15.
20. Naumen Service Desk, http://www.naumen.ru/products/service_desk/, last accessed 2017/04/15.
21. Raml-mock-service, https://github.com/veor12/raml-mock-service, last accessed 2017/04/15.