Seminar Web Engineering WS 2018/2019

API Modeling and Description Languages

Alexander Senger and Shyam Agrawal

Professorship of Distributed and Self-organizing Systems
Technische Universität Chemnitz
Chemnitz, Germany

Number of words: 3278

1. Introduction

The purpose of this report is to investigate the practically usage and creation of APIs and the importance of description languages. The report also suggests four description languages which can be used in different use cases. The ongoing research area Web Engineering offers multiple construction approaches of different Web-based applications and distributed systems. One of those research areas is the modeling process of APIs and their description languages. With a big spectrum of API usage areas **[1]** and their design approaches the necessity of taking a closer look on the API landscape became more important. With more complex APIs the need of describing such cases became part of the modeling process as well.

2. API Modeling

API Modelling helps to identify the API's Requirements. The main goal of API Modelling is to identify and understand stakeholders and their activities. All the activities are explored in detail in order to understand a better overall picture which can be used while designing the API.

2.1 What is an API?

Nowadays connectivity is increasing day by day making life simpler. The world is connected like never before. This instinct connectivity is due to the API or Application Programming Interface. The definition in the textbooks as well as on internet is:

"In computer programming, an application programming interface (API) is a set of subroutine definitions, communication protocols, and tools for building software." **[2]**

In simple terms, an API is an interface of a system to which client can interact with. Almost every web service on the internet wants to expose their data. These data are exposed through apis called the endpoints. These endpoints are used to interact as well as exchange data with the web service. API acts an a messenger where your message is delivered to the recipient and the response is delivered to you back. API helps the user to use the functionality provided by the Web Service. Consider a Restaurant System, here the customer has a list of items that can be ordered from the kitchen. Now here there is a communication gap between the customer and the kitchen. Here waiter comes into the scenario. The waiter acts a messenger takes the order from customer to kitchen and deliver the dishes from kitchen back to the customer. There are various design factors and practices which should be considered before Modelling an API.

2.2 Important Factors in API Design

While designing an api the following factors should be considered.

2.2.1 Target Audience

Designing an api with target audience is very important. Target audience are the actual people who will be using the api. Here most important is to understand the API's purpose. API user can be anyone. A new developer or an experienced developer. A new first time developer can through the api documentation and use the api accordingly. An experienced developer can update oneself regarding the api usage. The consumers of the api are categorized into two types.

Direct Consumers

Consumer benefits from the direct usage of the specified core services. For Example, Uber has its own ecosystem. When an user uses an uber service, it is direct communicating with uber's servers using uber's api.

Indirect Consumers

Consumer benefits from addon services apart from the core services. For Example, Uber's ecosystem can integrate services from Yelp. So now as soon as the user ends the journey uber will suggest the user new restaurants nearby. Here uber will use yelp's api to fetch the desired information. Here service of a complete different segment is provided which makes the consumers an indirect user of the addon service.

2.2.2 Business Drivers

Business drivers aims to understand api's purpose. It focuses on to fulfil the business and technological goal. To fulfil the business needs apis are categorized into 3 types.

Private API

Private APIs are quickly set up for internal teams to communicate within the organization with low cost. For Example, Jeff Bezos told all the teams at Amazon that "all teams will expose their data and functionalities through interfaces and internal communication should be carried out by interfaces only." **[3]**

Public API

Public APIs can be considered as a commercial api. It is majorly developed with a goal of high adoption. This is the final endpoint which the developers will use to interact with the web service.

Partner API

Partner Apis are a combination of private and public apis. It also serves the organization's internal as well as external business and technological needs. Amazon is the perfect example of an organization with Partner APIs.

2.2.3 Resources

The technology stack is kept in mind while designing any API. The web service provider should be aware of what type of data is stored and should be transferred. The technological limitations sometimes plays an obstacle in long run. The technology used might have limitation to support the api functionality. Everything should be noted before designing an api.

2.3 Practices in Good API Design

There are few common practices that should be implemented for a good API design.

2.3.1 Understanding Resources and Collections

"A resource is an object that's important enough to be referenced in itself. A group of resources is called a collection" **[4]**. There is an unique identifier which identifies these resources and collections. So these data should be addressed in simple format which should be understandable by anyone who uses it. Consider the following example:

Examples
1. /products - This api helps to retrieve all products from the web service.
2. /products/321 - This api helps to retrieve all the information for a specific product.

2.3.2 Use Nouns and NOT the Verbs

Developers comes across this problem. Sometimes verbs are used in the url which creates a confusion with use of http methods which performs the same function. Therefore, It is recommended to use Nouns in the urls.

Examples
1. /products - Recommended
2. /getAllProducts - Not Recommended

2.3.3 HTTP Methods

Resources and collections can be accessed with the help of several methods for the data being showcased by the API. There are various methods which can be used for different type of operation.

GET	To get a resource or collection of resources.
POST	To create a resource or collection of resources.
PUT	To update the existing resource or collection of resources.
DELETE	To delete the existing resource or the collection of resources.

Table 1: HTTP Methods **[5]**

2.3.4 HTTP Response Codes

There are plenty of HTTP Response codes but most of the time only 200 and 500 are used. A good approach is to use the http response codes specific with the response type.

200	OK

201	CREATED
202	ACCEPTED
301	MOVED PERMANENTLY
400	BAD REQUEST
401	UNAUTHORIZED
403	FORBIDDEN
404	NOT FOUND
500	INTERNAL SERVER ERROR
502	BAD GATEWAY

Table 2: Status Codes **[6]**

2.3.5 Versioning

There might be changes in the resources which will need updation in the apis. So versioning is introduced in order to track different changes made to the apis. These versioning can be also used as query parameters to access the data.

Examples
1. /v1/products
2. /v2/products

2.3.6 Pagination

Pagination is important where a large chunk of data is involved. If large chunk of data is not handled properly it may take the system down. In pagination, “offset” and “limit” attributes are used. “Limit” relates to the size of data and “offset” relates to the position in the chunk where the data should start.

Example
1. /products?limit=25&offset=50

Here the size of the data should be 25 and the offset position should be 50.

2.3.7 Complex Parameters

Resources comes with a lot of properties. Complex Parameters are used with the combination of these resource properties in order to extract data with a specific property.

Example
1. GET /products?name=’ABC’&price=20

In the above example a specific product is retrieved with a name “ABC” and price “20”.

2.3.8 Provide Examples

The best way to design and document an api is to provide examples of data request and response. The developer gets an idea what type of data response is expected from the server. Developers also learn what and how to call a function method in order to communicate with the web service. Below example shows type of response a developer can expect.

Example

1. { “products”: [{ “name”: “ABC”, "price": “20” }] }

3. API Description Languages (DL)

With evolving technologies of the web and the popularity of APIs, the need of having guidelines for constructing and manage these interfaces were getting stronger. Especially when the API wants to be understood and used by other developers, a document for describing those processes is necessary to get a overall overview. In the next sections the importance of description languages and how they differ will be demonstrated.

3.1 Definition

First of all in general API Description Languages (DLs) are specific languages for describing APIs **[7]**. They are written in a document and tell one which methods and data are used by the described API. There are also other information API description languages provide like endpoints, schemas and message-transfer-types. Second, they serve as a contract between service consumer and producer to make sure everyone is following the right rules. Especially when it comes to using an API as a external developer, they somehow need to understand how to call the methods. The last thing to mention about API description languages is that they are human and machine readable. Human readability means that documentations can be created out of a description language file to make it faster understandable and easy to read **[7]**. On the other hand machine readability means that the machines on the internet are also able to process it.

3.2 Problems without Description Languages

To get a better understanding why description languages are so important it is needed to come across typical problems which might occur in some situations where an API is needed which was not selfmade. For the first example let’s take an API call from this imaginary website: *https://example.com/shoppingcart*. This example should deliver the current shopping cart with the common GET request but you can’t really tell if there are other valid HTTP methods for the website available like POST or DELETE. A simple created documentation from a description language would handle this problem because the documentation would list all methods which are used in the API as a overview. The next problem describes the accessibility of an API. There are different ways to structure APIs for example with paths or queries. In the next example the goal is to access exactly one item out of the warehouse store. A closer look at the two website GET requests: *https://example.com/warehouse/**1*** and *https://example.com/warehouse**?item=1***, already tells that both of them can be indeed valid GET requests. The outstanding problem here is that it is still unknown which of the requests is heard by the API. Therefore a description language can teach a proper usage of the current API. The last problem which needs to be tackled is the expectation of the outcoming results. Working with APIs mostly deliver structures which need to be processed further with a parser into other data in order to be able to work with it. It’s really hard to achieve that without knowing the responses of an API request. As an example the GET request *https://example.com/warehouse/1* works now and delivers one item of the warehouse as a XML structure. As a modern developer one expect JSON as a response but it is more likely to run into a issue here. To get a better understanding which method is delivering what kind of result it is recommended to use a description language.

3.3 Four API Description Languages in Detail

In the following section four API description languages will be presented. The focus will be to show the strengths and weaknesses of each language and to point out the need of them. Those languages were chosen because they are still relevant nowadays and popular in different cases as well. It is worth to mention that there are more description languages out there. In almost every description language there is the possibility to automatically generate the description language out of the API code and vice versa.

3.3.1 Web Services Description Language (WSDL)

WSDL, in version 1.1, is the first description language which is created to describe SOAP based web services. It is written in XML format therefore it is not very human readable because when the document is growing it will be harder to take a quick look on it if something needs to be changed. Even if XML is hard to read it still has its purpose. First XML is used by the SOAP protocol and second XML is still used and can be interpreted by many machines in the world. That makes XML platform independent **[8]**. When it comes to the question what a WSDL would look like, it firstly needs to be inspected what a WSDL is doing. “A client can load a WSDL file and know exactly which RPC-style methods it can call, what arguments those methods expect, and which data types they return.” **[9]** This also includes the different endpoints the API is offering and their transfer protocol types. Usually WSDL is using SOAP over HTTP as transportation protocol for messages but other protocols like SMTP or FTP are possible.

Structure

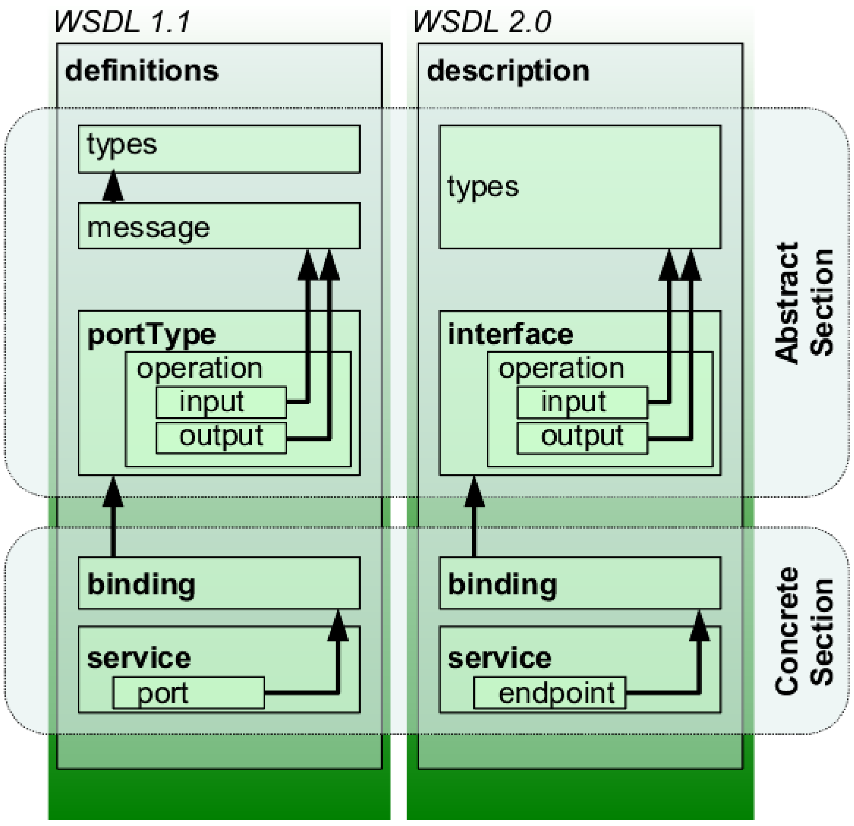


Figure 1: A WSDL Structure Overview **[10]**.

There are two versions of WSDL. The major change between both is the part where types and messages got combined to the type element in the version 2.0. In the following paragraph the WSDL version 1.1 will be briefly covered because the 2.0 version was a answer to the evolving REST community but another description language managed it better. Therefore version 1.1 was better adopted by the industry **[11]**. A WSDL document, which can be seen in **Figure 1**, is parted into two main sections: the abstraction section and the concrete section. While the concrete section describes how the web service communicates and where it can be reached for for instance via an URL, the abstraction section describes what the web service does for instance which operations are available.

The concrete part consists of the two elements binding and service with its including port element. The service element describes the endpoints of the API with their reachable URLs and the binding element describes how the web service is bound to a protocol. There are different network protocols available but the most common one is SOAP over HTTP. The abstraction section is the bigger part and consists of the elements portType, messages and types. PortType describes all operations with their input parameters and output result types **[12]**. To get a better understanding why the message element exists it needs to take a look at the types element. The types element simply defines all kinds of data types used by the web service. It can be either self defined data structures or already existing types like integer. The problem is that the WSDL 1.1 SOAP/RPC-centric communication is limited. “For example, it cannot describe a variable number of input parameters or a choice of responses **[13]**.” It means that it is not possible to have multiple data types as input or output declared if there is only the possibility to insert one data type. Therefore a element which combines multiple data types to one element is needed. It’s called the message element.

3.3.2 Web Application Description Language (WADL)

The answer to the growing REST popularity was WADL. Since SOAP was too complex and being inflexible with errors, it wasn't the ideal solution for the web **[14]**. There also became a need to describe contracts not only in a SOAP/RPC but also in a RESTful manner. WADL filled this gap with describing HTTP based web services and strictly targeting the requirements of RESTful services by serving web resources. WADL isn't only more lightweight and easier to understand, it als shows relationships between resources and is able to deliver HTTP response codes like 200 OK. Even if WADL supports REST, it is still written in XML.

Structure

```
<application>
  <grammars>
    <include href="Error.xsd"/>
  </grammars>
  <resources base="http://api.example.com/MyService/">
    <resource path="productCatalogue">
      <method name="GET" id="search">
        <request>
          <param name="results" style="query" type="xsd:int" default="10"/>
        </request>
        <response status="400">
          <representation mediaType="application/xml" element="Error"/>
        </response>
      </method>
    </resource>
  </resources>
</application>
```

Listing 1: A simple WADL document structure **[15]**.

Visually WADL looks more simple and easier to read than WSDL which can be seen in **Listing 1**. The application tag is parted into two main parts. Grammars where all schemas and formats are defined which describe the exchanged data and the resources tag with the base URI of the API which stores all resources with all operations provided by the API.

3.3.3 RESTful API Modeling Language (RAML)

"WADL itself isn't too great; it doesn't really capture enough of the semantics of the service to make it possible to tool things up." **[16]** With the need of more powerful REST description languages, two modern REST API description languages are presented. RAML 1.0 is a YAML-based language. This makes it more readable. RAML supports the API first approach which forces one to think about the structure of the API first. In some situations designing an API might be difficult, therefore prototypes can easily built with the API mocking technique. API mocking is the process of simulating components in a test environment without the need to write code. The design patterns can also be shown to customers to make flexible decisions without messing around with code. RAML is also used by many big companies like Spotify or Sky **[17]**. Behind RAML stands a large community with handy tools like RAML API-Designer which is a great online editor including code-completion and instant method testing support. Another great tool to mention is the API console which creates automatically a full documentation out of the API in HTML format.

Structure

```
#%RAML 1.0
title: Product Catalogue API
version: v1
protocols: [http, https]
baseUri: https://api.Product-Catalogue.com
mediaType: [application/json, application/xml]

/products:
  /{productID}:
    get:
      queryParameters:
        productID:
          description: The ID of the product
          type: integer
          required: true
      responses:
        200:
          body:
            application/json:
              example: {"productID": 1, "productType": "chair", "price": "7.9
```

Listing 2: A simple RAML document structure **[18].**

As it is shown in **Listing 2** the RAML file splits into the metadata part and the resources part. The metadata usually describe a quick overview of the use protocols, the base URI and the supported mediaTypes. There are also more options available like schemas (type in RAML 0.8), documentations and security options. The resources have a detailed structurering now with many sub elements. The idea of input parameters and responses remain the same. It is also possible to include example responses for API mocking.

3.3.4 OpenAPI

As the second modern API description language OpenAPI was chosen, which was originally known as Swagger 2.0. “It became a separate project in 2016 [...]” with a consortium of industry experts like IBM or Microsoft **[19]**. There are two differences between RAML and OpenAPI. While RAML supports YAML as language format, OpenAPI also supports JSON as a language format. OpenAPI doesn’t only support the API first approach, but also the code first approach. Like RAML, OpenAPI also has a big community with a lot of tools like the Swagger Editor, a online OpenAPI editor with code-completion and live editing. If there is a plan to create a documentation, Swagger UI can be used for creating a overview of all API methods.

Structure

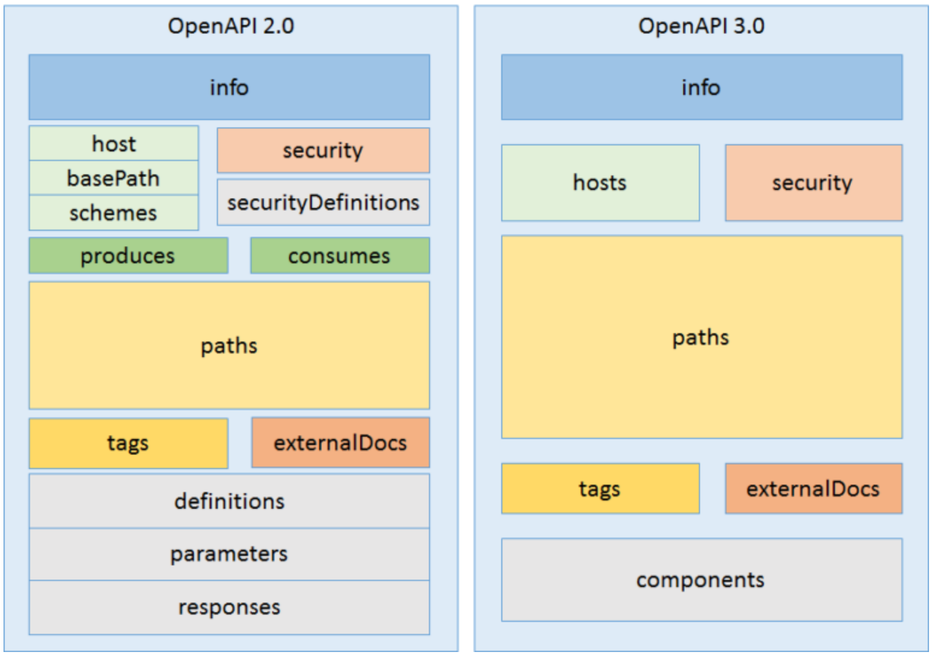


Figure 2: A OpenAPI Structure Overview **[20]**.

The **Figure 2** shows that there are a couple of changes happened between Swagger 2.0 and OpenAPI 3.0. There are two major changes which will be covered. The first change was the support of multiple hosts in version 3.0 because 2.0 only supported one host with its basePath and schemas. The second major change was to get rid of the two elements produces and consumes which moved to the components element. In 2.0 it was only possible to choose one media type for the whole API. Now in 3.0 this changed and every path can have its own media type. As a minor change the three bottom elements in 2.0 got a rename to the components element with some additional subelements **[21]**.

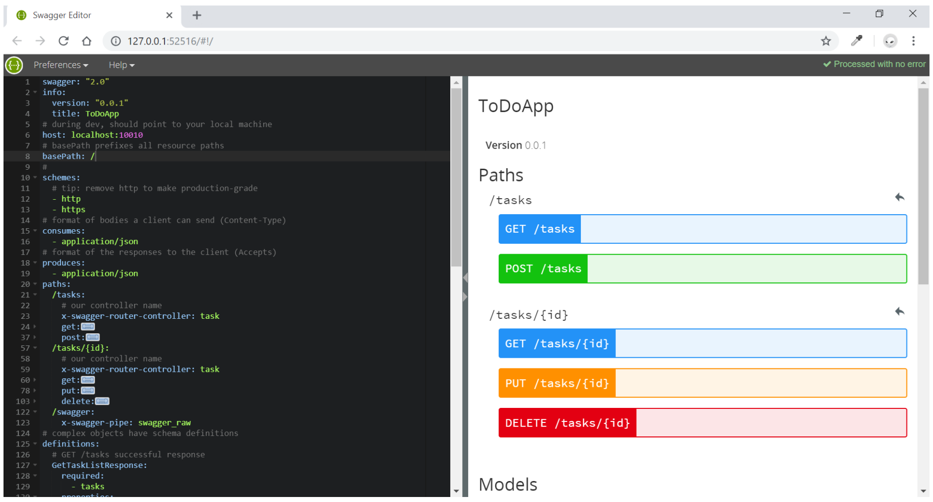
3.4 Choosing the right description language

When it comes to choosing a description language for the next API project there are several conditions to think about. First of all there is the decision between a SOAP and a REST API. When it gets into the SOAP direction the best choice would be WSDL because its made for complex enterprise systems with SOAP based web services **[14]**. On the other hand there are more options for a REST API available. WADL is by far the oldest option to think about but its not really used that often anymore. “WADL appeals to people coming from the SOAP world […].” **[22]** RAML and OpenAPI are more attractive nowadays with a lot of community tools but how to decide between both? In the end it comes down to minor differences like the language structure and the design approach but in general both languages will do a great job. It is recommended using OpenAPI because its more up to date with its 3.0 version and has a consortium who tries to standardize APIs.

4. Demo

The idea of the demo is to create a todo list application. The application will help to add, edit, delete, find and display daily tasks as shown in the Fig 3. These tasks will accessed through an API. The API is designed using Swagger 2.0. Swagger framework is based on

OpenAPI specifications. The API is created using the Design-First approach. The business logic is written in JavaScript. An unique ID is generated automatically and assigned to the tasks.



```yaml
swagger: "2.0"
info:
  version: "0.0.1"
  title: ToDoApp
# during dev, should point to your local machine
host: localhost:10010
# basePath prefixes all resource paths
basePath: /
#
schemes:
  # tip: remove http to make production-grade
  - http
  - https
# format of bodies a client can send (Content-Type)
consumes:
  - application/json
# format of the responses to the client (Accepts)
produces:
  - application/json
paths:
  /tasks:
    # our controller name
    x-swagger-router-controller: task
    get:
    post:
  /tasks/{id}:
    # our controller name
    x-swagger-router-controller: task
    get:
    put:
    delete:
  /swagger:
    x-swagger-pipe: swagger_raw
# complex objects have schema definitions
definitions:
  # GET /tasks successful response
  GetTaskListResponse:
    required:
      - tasks
```

ToDoApp

Version 0.0.1

Paths

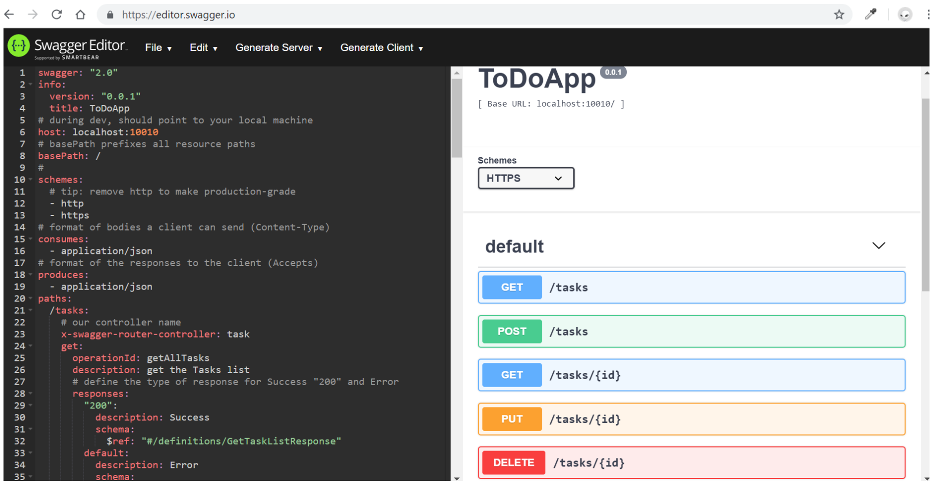
/tasks
- GET /tasks
- POST /tasks

/tasks/{id}
- GET /tasks/{id}
- PUT /tasks/{id}
- DELETE /tasks/{id}

Models

Figure 3: Swagger Demo Overview.

The online Swagger editor lets you design the API and then automatically generate client and server stubs as shown in Fig 4.



```yaml
swagger: "2.0"
info:
  version: "0.0.1"
  title: ToDoApp
# during dev, should point to your local machine
host: localhost:10010
# basePath prefixes all resource paths
basePath: /
#
schemes:
  # tip: remove http to make production-grade
  - http
  - https
# format of bodies a client can send (Content-Type)
consumes:
  - application/json
# format of the responses to the client (Accepts)
produces:
  - application/json
paths:
  /tasks:
    # our controller name
    x-swagger-router-controller: task
    get:
      operationId: getAllTasks
      description: get the Tasks list
      # define the type of response for Success "200" and Error
      responses:
        "200":
          description: Success
          schema:
            $ref: "#/definitions/GetTaskListResponse"
        default:
          description: Error
          schema:
```

ToDoApp 0.0.1

[Base URL: localhost:10010/]

Schemes: HTTPS

default
- GET /tasks
- POST /tasks
- GET /tasks/{id}
- PUT /tasks/{id}
- DELETE /tasks/{id}

Figure 4: Swagger Online Editor.

5. Bibliography

[1] Medjaoui, Mehdi (2018, November). The API Landscape [Online]. Available: **https://www.apidayssf.com/api-landscape** (Accessed: Jan. 02, 2019)

[2] Wikipedia [Online]. Available: **https://en.wikipedia.org/wiki/Application_programming_interface** (Accessed: Dec. 28, 2018)

[3] Ross Mason (2017 August). ACCELERATING INNOVATION IN THE DIGITAL AGE [Online]. Available: **https://www.cio.com/article/3218667/digital-transformation/have-you-had-your-bezos-moment-what-you-can-learn-from-amazon.html** (Accessed: Jan. 07, 2019)

[4] Best Practices in API Design [Online]. Available: **https://swagger.io/resources/articles/best-practices-in-api-design/** (Accessed: Jan. 07, 2019)

[5] HTTP Method Definitions [Online]. Available: **https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html** (Accessed: Jan. 07, 2019)

[6] HTTP Status Code Definitions [Online]. Available: **https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html** (Accessed: Jan. 07, 2019)

[7] mattbiehl (2015, June 17). What are API Description Languages? [Online]. Available: **https://api-university.com/blog/what-are-api-description-languages** (Accessed: Dec. 27, 2018)

[8] Knowledge Base (2015, December 02). SOAP vs REST [Online]. Available: **http://blog.empeccableweb.com/wp/2015/12/02/soap-vs-rest** (Accessed: Dec. 28, 2018)

[9] Bekkar, Sanae (2016, February 23). WSDL VS WADL [Online]. Available: **https://sanaebekkar.wordpress.com/2016/02/23/wsdl-vs-wadl** (Accessed: Dec. 28, 2018)

[10] Wikipedia [Online]. Available: **https://upload.wikimedia.org/wikipedia/commons/c/c2/WSDL_11vs20.png?1544693032133** (Accessed: Dec. 28, 2018)

[11] Murgante, Beniamino et al. "Computational Science and Its Applications – ICCSA 2013". 13th International Conference. Ho Chi Minh City, Vietnam. Proceedings, Part II. June 24-27 2013

[12] Christensen, Erik and Curbera, Francisco and Meredith, Greg and Weerawarana, Sanjiva (2001, March 15). Web Services Description Language (WSDL) 1.1 [Online]. Available: **https://www.w3.org/TR/2001/NOTE-wsdl-20010315** (Accessed: Dec. 29, 2018)

[13] Kalali, Masoud (2009, April 23). A Look at WSDL 2.0 [Online]. Available: **https://dzone.com/articles/look-wsdl-20** (Accessed: Dec. 29, 2018)

[14] Hemel, Zef (2008, January 8). Why WADL is Awesome [Online]. Available: **https://zef.me/why-wadl-is-awesome-5b811e32c74c** (Accessed: Dec. 29, 2018)

[15] A simpified and modified version of the W3C WADL example [Online]. Available: **https://www.w3.org/Submission/wadl** (Accessed: Dec. 29, 2018)

[16] Fellows, Donal (2012, February 7). Should I use WADL to describe my RESTful API? [Online]. Available: **https://softwareengineering.stackexchange.com/a/133713** (Accessed: Dec. 29, 2018)

[17] RAML Specification [Online]. Available: **https://raml.org/enterprises** (Accessed: Dec. 29, 2018)

[18] A simpified and modified version of the RAML Specification example [Online]. Available: **https://raml.org/developers/raml-100-tutorial** (Accessed: Dec. 29, 2018)

[19] Wikipedia: OpenAPI Specification [Online]. Available: **https://en.wikipedia.org/wiki/OpenAPI_Specification?oldformat=true** (Accessed: Dec. 30, 2018)

[20] Open API Initiative (2016, October 3). OpenAPI Specification [Online]. Available: **https://www.openapis.org/news/blogs/2016/10/tdc-structural-improvements-explaining-30-spec-part-2** (Accessed: Dec. 30, 2018)

[21] ReadmeBlog (2017, March 20). A Visual Guide to What's New in Swagger 3.0 [Online]. Available: **https://blog.readme.io/an-example-filled-guide-to-swagger-3-2** (Accessed: Dec. 30, 2018)

[22] Miller, Darrel (2009, August 21). What is the reason for using WADL? [Online]. Available: **https://stackoverflow.com/a/1314326/10192487** (Accessed: Jan. 01, 2019)