33.8 Rest Services with Swagger

Swagger https://swagger.io/ is a tool for developing API specifications based on the OpenAPI Specification (OAS). It allows not only the specification, but the generation of code based on the specification in a variety of languages.

Swagger itself has a number of tools which together build a framework for developing REST services for a variety of languages.

33.9 Swagger Tools

The major Swagger tools of interest are:

Swagger Core includes libraries for working with Swagger specifications https://github.com/swagger-api/swagger-core.

Swagger Codegen allows to generate code from the specifications to develop Client SDKs, servers, and documentation. https://github.com/swagger-api/swagger-codegen

Swagger UI is an HTML5 based UI for exploring and interacting with the specified APIs https://github.com/swagger-api/swagger-ui

Swagger Editor is a Web-browser based editor for composing specifications using YAML https://github.com/swagger-api/swagger-editor

The developed APIs can be hosted and further developed on an online repository named SwaggerHub https://app.swaggerhub.com/home The convenient online editor is available which also can be installed locally on a variety of operating systems including OSX, Linux, and Windows.

33.10 Swagger Specification

Swagger provides through its specification the definition of REST services through a YAML or JSON document.

When following the API-specification-first approach to define and develop a RESTful service, the first and foremost step is to define the API conforming to the OpenAPI specification, and then using codegen tools to conveniently generate server side stub code, client code, documentations, in the language you desire. In this Section 33.10.1 we use an example to show how to define an API following the OpenAPI 2.0 specification. The example is a simplified object to specify a virtual cluster. In Section 33.11 the we introduce the codegen tool and how to use that to generate server side and client side code.

The OpenAPI Specification is formerly known as Swagger RESTful API Documentation Specification. It defines a specification to describe and document a RESTful service API. It is also known under version 3.0 of swagger. However, as the tools for 3.0 are not yet completed, we will continue for now to use version swagger 2.0, till the transition has been completed. THis is especially of importance, as we need to use the swagger codegen tool, which currently support only up to specification v2. Hence we are at this time using OpenAPI/Swagger v2.0 in our example. There are some structure and syntax changes in v3, while the essence is very similar. For more details of the changes between v3 and v2, please refer to A document published on the Web titled Difference between OpenAPI 3.0 and Swagger 2.0.

You can write the API definition in json for yaml format. Let us discuss this format briefly and focus on yaml as it is easier to read and maintain.

On the root level of the yaml document we see fields like *swagger*, *info*, and so on. Among these fields, ***swagger***, ***info***, and ***path*** are **required**. Their meaning is as follows:

swagger specifies the version number. IN our case a string value '2.0' is used as we are writing the definition conforming to the v2.0 specification.

info defines metadata information related to the API. E.g., the API *version*, *title* and *description*, *termsOfService* if applicable, *contact* information and *license*, etc. Among these attributes, ***version*** and ***title*** are required while others are optional.

path defines the actual endpoints of the exposed RESTful API service. Each endpoint has a *field pattern* as the key, and a *Path Item Object* as the value. In this example we have defined */vc* and */vc/{id}* as the two service endpoints. They will be part of the final service URL, appended after the service *host* and *basePath*, which will be explained later.

Let us focus on the *Path Item Object*. It contains one or more supported *operations* on the service endpoint. An *operation* is keyed by a valid HTTP operation verb, e.g., one of **get**, **put**, **post**, **delete**, or **patch**. It has a value of *Operation Object* that describes the operations in more detail.

The *Operation Object* will always **require** a *Response Object*. A *Response Object* has a *HTTP status code* as the key, e.g., **200** as successful return; **40X** as authentication and authorization related errors; and **50x** as other server side servers. It can also has a default response keyed by **default** for undeclared http status return code. The *Response Object* value has a **required** *description* field, and if anything is returned, a *schema* indicating the object type to be returned, which could be a primitive type, e.g., *string*, or an *array* or customized *object*. In case of *object* or an *array* of *object*, use *$ref* to point to the definition of the object. In this example, we have

```
$ref: "#/definitions/VC"
```

to point to the *VC* definition in the *definitions* section in the same specification file, which will be explained later.

Besides the required field, the *Operation Object* **can** have *summary* and *description* to indicate what the operation is about; and *operationalId* to uniquely identify the operation; and *consumes* and *produces* to indicate what MIME types it expects as input and for returns, e.g., *application/json* in most modern RESTful APIs. It can further specify what input parameter is expected using *parameters*, which requires a *name* and *in* fields. *name* specifies the name of the parameter, and *in* specifies from where to get the parameter, and its possible values are *query*, *header*, *path*, *formData* or *body*. In this example in the */vc/{id}* path we obtain the *id* parameter from the URL path wo it has the *path* value. When the *in* has *path* as its value, the *required* field is required and has to be set as *true*; when the *in* has value other than *body*, a *type* field is required to specify the type of the parameter.

While the three root level fields mentioned above are required, in most cases we will also use other optional fields.

host to indicate where the service is to be deployed, which could be *localhost* or a valid IP address or a DNS name of the host where the service is to be deployed. If other port number other than *80* is to be used, write the port number as well, e.g., *localhost:8080*.

schemas to specify the transfer protocol, e.g, *http* or *https*.

basePath to specify the common base URL to be append after the *host* to form the base path for all the endpoints, e.g., */api* or */api/1.0/*. In this example with the values specified we would have the final service endpoints *http://localhost:8080/api/vcs* and *http://localhost:8080/api/vc/{id}* by combining the *schemas*, *host*, *basePath* and *paths* values.

consumes and produces can also be specified on the top level to specify the default MIME types of the input and return if most *paths* and the defined operations have the same.

definitions as used in in the *paths* field, in order to point to a customized object definition with a *$ref* keyword.

The *definitions* field really contains the object definition of the customized objects involved in the API, similar to a class definition in any Object Oriented programming language. In this example, we defined a *VC* object, and hierarchically a *Node* object. Each object defined is a type of *Schema Object* in which many field could be used to specify the object (see details in the REF link at top of the document), but the most frequently used ones are:

type to specify the type, and in the customized definition case the value is mostly *object*.

required field to list the names of the required attributes of the object.

properties has the detailed information of each attribute/property of the object, e.g, *type*, *format*. It also supports hierarchical object definition so a property of one object could be another customized object defined elsewhere while using *schema* and *$ref* keyword to point to the definition. In this example we have defined a *VC*, or virtual cluster, object, while it contains another object definition of

Node as part of a cluster.

33.10.1 The Virtual Cluster example API Definition

```yaml
swagger: "2.0"
info:
  version: "1.0.0"
  title: "simple VC"
  description: "A simple service for VC as a test of using
   swagger-2.0 specification and codegen"
  termsOfService: "http://swagger.io/terms/"
```

section/rest/swagger-spec.tex

```
  contact:
    name: "IU ISE software and system team"
  license:
    name: "Apache"
host: "localhost:8080"
basePath: "/api"
schemes:
  - "http"
consumes:
  - "application/json"
produces:
  - "application/json"
paths:
  /vcs:
    get:
      description: "Returns all VCs from the system that the
      user has access to"
      produces:
        - "application/json"
      responses:
        "200":
          description: "A list of VCs."
          schema:
            type: "array"
            items:
              $ref: "#/definitions/VC"
  /vcs/{id}:
    get:
      description: "Returns all VCs from the system that the
      user has access to"
      operationId: getVCById
      parameters:
        - name: id
          in: path
          description: ID of VC to fetch
          required: true
          type: string
      produces:
        - "application/json"
      responses:
        "200":
          description: "The vc with the given id."
          schema:
            $ref: "#/definitions/VC"
        default:
          description: unexpected error
          schema:
            $ref: '#/definitions/Error'
definitions:
  VC:
    type: "object"
    required:
      - "id"
      - "name"
      - "nnodes"
      - "FE"
```

section/rest/swagger-spec.tex

```
      - "computes"
    properties:
      id:
        type: "string"
      name:
        type: "string"
      nnodes:
        type: "integer"
        format: "int64"
      FE:
        type: "object"
        schema:
          $ref: "#/definitions/Node"
      computes:
        type: "array"
        items:
          $ref: "#/definitions/Node"
      tag:
        type: "string"
  Node:
    type: "object"
    required:
      - "ncores"
      - "ram"
      - "localdisk"
    properties:
      ncores:
        type: "integer"
        format: "int64"
      ram:
        type: "integer"
        format: "int64"
      localdisk:
        type: "integer"
        format: "int64"
  Error:
    required:
    - code
    - message
    properties:
      code:
        type: integer
        format: int32
      message:
        type: string
```

33.10.2 Refernces

The official OpenAPI 2.0 Documentation

33.11 REST Service Generation with Swagger

Swagger (36:02)

In this subsection we are discussing how to use OpenAPI 2.0 and Swagger Codegen to define and develop a REST Service.

We assume you have been familiar with the concept of REST service, OpenAPI and Swagger as discussed in Sections 33.1 and 33.10.

We will use a simple example to demonstrate the process of developing a REST service with Swagger/OpenAPI 2.0 specification and the tools related to is. The general steps are:

- Step 1 (Section 33.11.1). Define the REST service conforming to Swagger/OpenAPI 2.0 specification. It is a YAML document file with the basics of the REST service defined, e.g., what resources it has and what actions are supported.
- Step 2 (Section 33.11.2). Use Swagger Codegen to generate the server side stub code. Fill in the actual implementation of the business logic portion in the code.
- Step 3 (Section 33.11.3). Install the server side code and run it. The service will then be available.
- Step 4 (Section 33.11.4). Generate client side code. Develop code to call the REST service. Install and run to verify.

33.11.1 Step 1: Define Your REST Service

In this example we define a simple REST service that returns the hosting server's basic CPU info. The example specification in yaml is as follows:

```yaml
swagger: "2.0"
info:
  version: "0.0.1"
  title: "cpuinfo"
  description: "A simple service to get cpuinfo as an example of
    using swagger-2.0 specification and codegen"
  termsOfService: "http://swagger.io/terms/"
  contact:
    name: "Cloudmesh REST Service Example"
  license:
    name: "Apache"
host: "localhost:8080"
basePath: "/api"
schemes:
  - "http"
consumes:
  - "application/json"
produces:
  - "application/json"
paths:
  /cpu:
    get:
      description: "Returns cpu information of the hosting
   server"
      produces:
        - "application/json"
      responses:
        "200":
          description: "CPU info"
          schema:
            $ref: "#/definitions/CPU"
definitions:
  CPU:
    type: "object"
    required:
      - "model"
    properties:
      model:
        type: "string"
```

33.11.2 Step 2: Server Side Stub Code Generation and Implementation

With the REST service having been defined, we can now generate the server side stub code easily.

Setup the Codegen Environment

You will need to install the Swagger Codegen tool if not yet done so. For OSX we recommend that you use the homebrew install via

```
brew install swagger-codegen
```

On Ubuntu you can install swagger as follows (update the version as needed):

```
 mkdir ~/swagger
cd ~/swagger
wget https://oss.sonatype.org/content/repositories/releases/io/
   swagger/swagger-codegen-cli/2.3.1/swagger-codegen-cli-2.3.1.
   jar
alias swagger-codegen="java -jar ~/swagger/swagger-codegen-cli
   -2.3.1.jar"
```

Add the alias to your `.bashrc` or `.bash_profile` file. After you start a new terminal you can use in that terminal now the command

```
swagger-codegen
```

For other platforms you can just use the `.jar` file, which can be downloaded from this link.

Also make sure Java 7 or 8 is installed in your system. To have a well defined location we recommend that you place the code in the directory `~/cloudmesh`. In this directory you will also place the `cpu.yaml` file.

Generate Server Stub Code

After you have the codegen tool ready, and with Java 7 or 8 installed in your system, you can run the following to generate the server side stub code:

```
swagger-codegen generate \
    -i ~/cloudmesh/cpu.yaml \
    -l python-flask \
    -o ~/cloudmesh/swagger_example/server/cpu/flaskConnexion \
    -D supportPython2=true
```

or if you have not created an alias

```
java -jar swagger-codegen-cli.jar generate \
    -i ~/cloudmesh/cpu.yaml \
    -l python-flask \
    -o ~/cloudmesh/swagger_example/server/cpu/flaskConnexion \
    -D supportPython2=true
```

In the specified directory under *flaskConnexion* you will find the generated python flask code, with python 2 compatibility. It is best to place the swagger code under the directory `~/cloudmesh` to have a location where you can easily find it. If you want to use python 3 make sure to chose the appropriate option. To switch between python 2 and python 3 we recommend that you use pyenv as discussed in our python section.

Fill in the actual implementation

Under the *flaskConnecxion* directory, you will find a *swagger_server* directory, under which you will find directories with *models* defined and *controllers* code stub resides. The models code are generated from the definition in Step 1. On the controller code though, we will need to fill in the actual implementation. You may see a `default_controller.py` file under the *controllers* directory in which the resource and action is defined but yet to be implemented. In our example, you will find such a function definition which we list next:

```
def cpu_get():  # noqa: E501
    """cpu_get

    Returns cpu info of the hosting server # noqa: E501


    :rtype: CPU
    """
    return 'do some magic!'
```

Please note the `do some magic!` string at the return of the function. This ought to be replaced with actual implementation what you would like your REST call to be really doing. In reality this could be some call to a backend database or datastore; a call to another API; or even calling another REST service from another location. In this example we simply retrieve the cpu model information from the hosting server through a simple python call to illustrate this principle. Thus you can define the following code:

```
import os, platform, subprocess, re

def get_processor_name():
    if platform.system() == "Windows":
        return platform.processor()
    elif platform.system() == "Darwin":
        command = "/usr/sbin/sysctl -n machdep.cpu.brand_string"
        return subprocess.check_output(command, shell=True).
  strip()
    elif platform.system() == "Linux":
        command = "cat /proc/cpuinfo"
        all_info = subprocess.check_output(command, shell=True).
  strip()
        for line in all_info.split("\n"):
            if "model name" in line:
                return re.sub(".*model name.*:", "", line, 1)
    return "cannot find cpuinfo"
```

section/rest/swagger-codegen.tex

And then change the **cpu_get()** function to the following:

```
def cpu_get():  # noqa: E501
    """cpu_get

    Returns cpu info of the hosting server # noqa: E501


    :rtype: CPU
    """
    return CPU(get_processor_name())
```

Plese note we are returning a CPU object as defined in the API and later generated by the codegen tool in the *models* directory.

It is best *not* to include the definition of `get_processor_name()` in the same file as you see the definition of `cpu_get()`. The reason for this is that in case you need to regenerate the code, your modified code will naturally be overwritten. Thus, to minimize the changes, we do recommend to maintain that portion in a different filename and import the function as to keep the modifications small.

At this step we have completed the server side code development.

33.11.3 Step 3: Install and Run the REST Service:

Now we can install and run the REST service. It is strongly recommended that you run this in a pyenv or a virtualenv environment.

Start a virtualenv:

In case you are not using pyenv, please use virtual env as follows:

```
virtualenv RESTServer
source RESTServer/bin/activate
```

Make sure you have the latest pip:

```
pip install -U pip
```

Install the requirements of the server side code:

```
cd ~/cloudmesh/swagger_example/server/cpu/flaskConnexion
pip install -r requirements.txt
```

Install the server side code package:

Under the same directory, run:

```
python setup.py install
```

Run the service

Under the same directory:

```
python -m swagger_server
```

You should see a message like this:

```
* Running on http://0.0.0.0:8080/ (Press CTRL+C to quit)
```

Verify the service using a web browser:

Open a web browser and visit:

- http://localhost:8080/api/cpu

to see if it returns a json object with cpu model info in it.

Assignment: How would you verify that your service works with a `curl` call?

33.11.4 Step 4: Generate Client Side Code and Verify

In addition to the server side code swagger can also create a client side code.

Client side code generation:

Generate the client side code in a similar fashion as we did for the server side code:

```
java -jar swagger-codegen-cli.jar generate \
    -i ~/cloudmesh/cpu.yaml \
    -l python \
    -o ~/cloudmesh/swagger_example/client/cpu \
    -D supportPython2=true
```

Install the client side code package:

Although we could have installed the client in the same python pyenv or virtualenv, we showcase here that it can be installed in a completely different environment. That would make it even posible to use a python 3 based client and a python 2 based server showcasing interoperability between python versions (although we just use python 2 here). Thus we create ane new python virtual environment and conduct our install.

```
virtualenv RESTClient
source RESTClient/bin/activate
pip install -U pip
cd swagger_example/client/cpu
pip install -r requirements.txt
python setup.py install
```

Using the client API to interact with the REST service

Under the directory *swagger_example/client/cpu* you will find a README.md file which serves as an API documentation with example client code in it. E.g., if we save the following code into a .py file:

```
from __future__ import print_function
import time
import swagger_client
from swagger_client.rest import ApiException
from pprint import pprint
# create an instance of the API class
api_instance = swagger_client.DefaultApi()

try:
    api_response = api_instance.cpu_get()
    pprint(api_response)
except ApiException as e:
    print("Exception when calling DefaultApi->cpu_get: %s\n" % e
  )
```

We can then run this code to verify the calling to the REST service actually works. We are expecting to see a return similar to this:

```
{'model': 'Intel(R) Core(TM)2 Quad CPU    Q9550  @ 2.83GHz'}
```

Obviosly, we could have applied additional cleanup of the information returned by the python code, such as removing duplicated spaces.

33.11.5 Towards a Distributed Client Server

Although we develop and run the example on one localhost machine, you can separate the process into two separate machines. E.g., on a server with external IP or even DNS name to deploy the server side code, and on a local laptop or workstation to deploy the client side code. In this case please make changes on the API definition accordingly, e.g., the **host** value.

33.11.6 Exercises

Exercise 33.11 In Section 33.11.1, we introduced a schema. The question relates to termsOf-Service: Investigate what the termOfService attribut is and suggest a better value. Discuss on piazza. ■

Exercise 33.12 In Section 33.11.1, we introduced a schema. The question relates to model: What is the meaning of model under the definitions ■

Exercise 33.13 In Section 33.11.1, we introduced a schema. The question relates to $ref: what is the meaning of the $ref. Discuss on piazza, come up with a student answer in class. ■

Exercise 33.14 In Section 33.11.1, we introduced a schema. What does the response 200 mean. Do you need other responses? ■

Exercise 33.15 After you have gone through the entire section and verified it works for you add create a more sophisticated schema and add more attributes exposing more information from your system. ■

Exercise 33.16 How can you for example develop a rest service that exposes portions of your file system serving large files, e.g. their filenames and their size? How would you download these files? Would you use a rest service, or would you register an alternative service such as ftp, DAV, or others? Please discuss in piazza. Note this will be a helping you to prepare a larger assignment. Think abou this first before you implement. ■

Exercise 33.17 You can try expand the API definition with more resources and actions included. E.g., to include more detailed attributes in the CPU object and to have those information provided in the actual implementation as well. Or you could try defining totally different resources. ■

section/rest/swagger-codegen.tex

Exercise 33.18 The codegen tool provides a convenient way to have the code stubs ready, which frees the developers to focus more on the API definition and the real implementation of the business logics. Try with complex implementation on the back end server side code to interact with a database/datastore or a 3rd party REST service. ■

Exercise 33.19 For advanced python users, you can naturally use function assignments to replace the `cpu_get()` entirely even after loading the instantiation of the server. However, this is not needed. If you are an advanced python developer, please feel free to experiment and let us know how you suggest to integrate things easily. ■