

LAB – REST interfaces

Hands on experience creating applications with rest interfaces

References:

- Flask library: <https://flask.palletsprojects.com/en/1.1.x/>
- REST API guideline: <https://hackernoon.com/restful-api-designing-guidelines-the-best-practices-60e1d954e7c9>

Design a REST interface

- Even though there are no official guidelines for the definition of REST APIs, there are some common rules
- Following those rules can help other developers to understand the structure of the REST interface and use it in the program
- Basic terms:
 - **Resource**, an object or representation of something. It has associated some data and a set of methods to operate (e.g. an Employee of a company, you can create, add or delete it)
 - **Collections**, a set of resources, e.g. Employees is the collection of Employee resources
 - **URL**, the Uniform Resource Locator and the path corresponding to the resource (through which the resource can be accessed)

Endpoint definition

- The name of an endpoint should reflect only the resource or the collection associated and not an action
- Actions should be defined *via the request method and not within the name of the resource*
- For instance, the APIs:
 - /getAllEmployees – to get the list of the employees of a company
 - /addNewEmployee – to add a new employee
 - /deleteEmployee – to remove an employee
 - /deleteAllEmployees – to remove all the employees
 - Includes not only the resource but also the action

Endpoint definition

- The proper definition is the following:
 - **Method GET /employees** – to get the list of the employees (a collection)
 - **Method DELETE /employees** – to remove all the employees
 - **Method DELETE /employees/ID** – to remove the employee corresponding to a certain ID
 - **Method POST /employees** – to create a new employee (the data of the employee is included in the payload of the request)
 - **Method GET /employees/ID** – to retrieve all the data associated to the employee with a certain ID

General rules

- The general rules to associate actions with HTTP methods are the following:
 - GET method is used to request data from the resource and should not produce any modification to the resource and its data
 - POST method is used to request the creation of a new resource, the data to be associated with the resource is included in the payload of the request
 - PUT method is used to request the update of the resource or the creation of a new one, the data for the update or the creation is included in the payload
 - DELETE method is used to request for the deletion of a resource
- A collection can be represented with its name (e.g. /employees), while resources within the collection can be represented with a unique ID or name (e.g. /employees/ID)

API design

- Several tools exist to help with the design of REST APIs
- Those tools can be used to specify the definition of the APIs, create the corresponding documentation and even create automatically a skeleton of the application
- One of those is SWAGGER (<https://swagger.io/>)
- An online version of the tool can be accessed via the following link:
 - <https://editor.swagger.io/>
- The editor adopt a specific language to define and describe the APIs, the language is defined within the framework of OpenAPI, an effort that aims at standardizing a language for REST API description
- The base path can include a string to identify the version of the API definition, e.g. /v2/employees

REST API example

General API definition

```
swagger: "2.0"
info:
  description: "My REST API"
  version: "1.0.0"
  title: "My REST API"
host: "myserver.domain.it"
basePath: "/v2"
tags:
- name: "employees"
  description: "Employee collection"
schemes:
- "http"
```


Paths definition

```
paths:
  /employees:
    post: ← POST Method
      tags:
      - "employees"
      summary: "Add a new employee"
      description: ""
      operationId: "addEmployee"
      consumes:
      - "application/json"
      - "application/xml"
      produces:
      - "application/xml" ← Input/Output type
      - "application/json"
```

File employees.yaml

```
parameters:
- in: "body"
  name: "body"
  description: "Employee data"
  required: true
  schema:
    $ref: "#/definitions/Employee"
responses:
  405:
    description: "Invalid input"
```

DATA: an
instance of
Employee

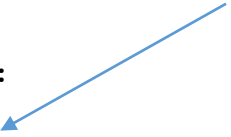


Input/Output type
definition



REST API example

definitions:

Employee: 

Employee object definition

```
Employee:
  type: "object"
  required:
    - "name"
    - "photoUrls"
  properties:
    id:
      type: "integer"
      format: "int64"
    name:
      type: "string"
      example: "Jim"
    photoUrls:
      type: "string"
      example: "http://mywebsite.org/mypic.jpg"
    xml:
      name: "Employee"
```

My REST API 1.0.0

[Base URL: 172.16.0.253:8080/v2]

My REST API

Schemes

HTTP

employees Employee collection

POST /employees Add a new employee

PUT /employees Update an existing employee

GET /employees/{employeeId} Find employee by ID

POST /employees/{employeeId} Updates an employee in the store with form data

DELETE /employees/{employeeId} Deletes an employee

Models

```
Employee {
  id integer($int64)
  name* string
  photoUrls* > [...]
}
```


Auto generated code

- The tool auto generate a skeleton of the code with a set of REST server for tomcat ready to execute a set of skeleton functions when the corresponding REST method is invoked
- The tool support different framework to create REST APIs, among them the *Apache CXF* and *Python Flask*
- To download the code:
 - “Generate Server” -> jaxrs-cxf or python-flask
- Both the packages contain a skeleton in which the functions associated with the REST calls are implemented, so developers can focus on the logic
- jaxrs-cxf is a Java framework to create web services, the package is a WAR package to be executed in Tomcat

Apache Tomcat

- Apache tomcat is an open-source implementation of the Java Servlet, a software component that extends the functionalities of a web server allowing to run Java code in response to an HTTP request
- Java Servlets are the Java counterpart to other dynamic web content technologies such as PHP and ASP.NET
- Tomcat instantiates an HTTP server to receive requests from clients
- The server can have one or more servlets instantiated
- A Servlet is an object that receives a request and generates a response based on that request
- Servlets are packaged in a WAR file (Web Application)

Pyhton flask

- Flask is a lightweight web service application framework designed to produce REST applications as simple as possible
 - <https://www.palletsprojects.com/p/flask/>
 - <https://flask.palletsprojects.com/en/1.1.x/>
- The package that swagger produces is ready to be executed into a container
- Download the package and copy it to the VM
- Unzip the package (install unzip 'apt install unzip')
`unzip python-flask-server-generated.zip`

Deploy the server

- Generate the container

```
docker build -t rest-server .
```

- Run it

```
docker run -p 8080:8080 -it rest-server
```

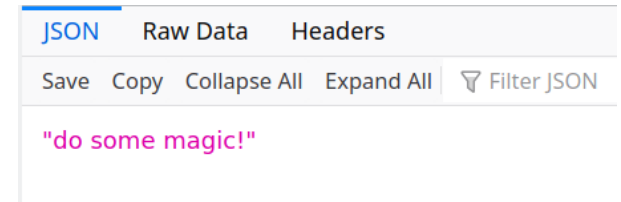
- Test it

- Open a browser and go to <http://172.16.0.X:8080/v2/employees/0>

- Run from the command line

```
curl -X POST "http://172.16.0.253:8080/v2/employees" -H  
"accept: application/json" -H "Content-Type:  
application/json" -d "{  \"id\": 0,  \"name\": \"Jim\",  
  \"photoUrls\": \"http://mywebsite.org/mypic.jpg\"}"
```

```
root@HAJJVX80PD7M5Q0:~/python-flask-server# docker run -p 8080:8080 -it rest-server  
* Serving Flask app "__main__" (lazy loading)  
* Environment: production  
  WARNING: This is a development server. Do not use it in a production deployment.  
  Use a production WSGI server instead.  
* Debug mode: off  
* Running on http://0.0.0.0:8080/ (Press CTRL+C to quit)
```



Fix a bug

- The current version of the Swagger editor has a bug
- One requirement is missing in the file requirements.txt
- Add this line in the second line:

Werkzeug==0.16.1

Test the server

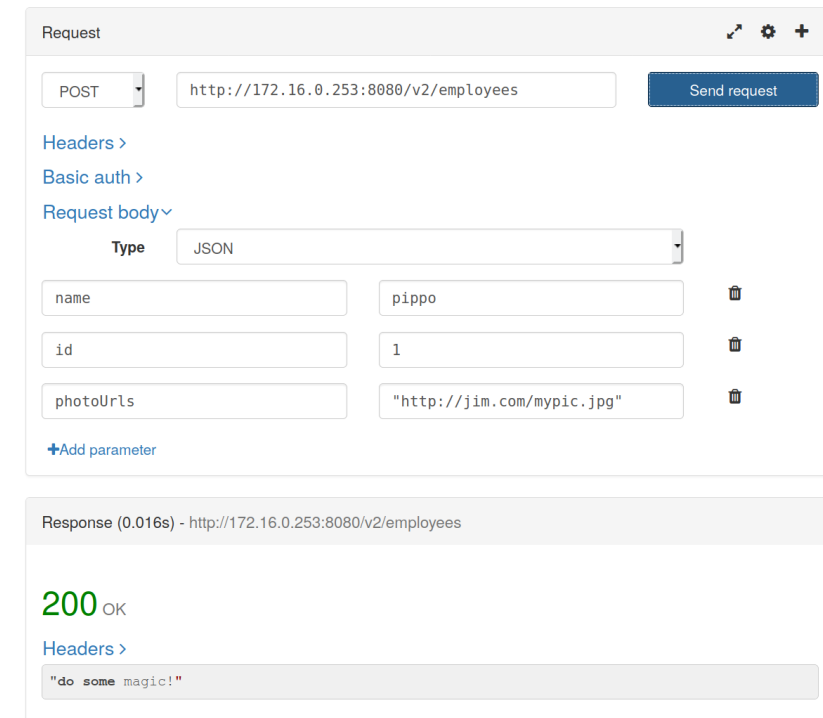
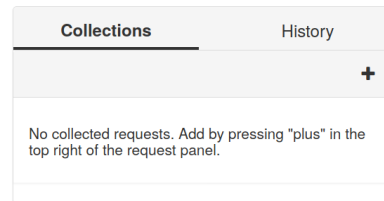
- To test REST interfaces install on your browser a REST extension (e.g. RESTED in Firefox or Chrome)

- Test a POST to add a new employee </> RESTED

- The payload must contain the data associated with the employee, e.g.:

```
{  
  "id": 0,  
  "name": "Jim",  
  "photoUrls":  
    "http://jim.com/mypic.jpg",  
}
```

- The response is “do some magic!” as the implementation is missing



Implement some function

- The implementation of the functions associated with the REST request must be included in the file:

`swagger_server/controllers/employees_controller.py`

- For instance the following function is called when a POST is received:

```
def add_employee(body):  
    if connexion.request.is_json:  
        body = Employee.from_dict(connexion.request.get_json())  
    return 'do some magic!'
```

Implement some function

- Add some print:

```
def add_employee(body):  
    if connexion.request.is_json:  
        body = Employee.from_dict(connexion.request.get_json())  
        print(body.name)  
        print(str(body.id))  
    return 'do some magic!'
```


Multi-tier cloud application



To retrieve IP addresses of the containers in the virtual network:

```
docker network inspect bridge
```

Deploy a database (Backend)

- Instantiate and additional container with MySQL

- Fetch a docker image with MySQL preinstalled

```
docker pull mysql/mysql-server
```

- Instantiate the container

```
docker run --name=mysql -d mysql/mysql-server
```

- Recover root DB password

```
docker logs mysql 2>&1 | grep GENERATED
```

```
root@HAJJVX80PD7M5Q0:~/python-flask-server# docker logs mysql 2>&1 | grep GENERATED
[Entrypoint] GENERATED ROOT PASSWORD: 4DuDUPuSIs#AHuk0bUcopfuLujUS
root@HAJJVX80PD7M5Q0:~/python-flask-server#
```

Configure the database

- Change the password

```
docker exec -it mysql bash
mysql -uroot -p
```

- Type the new password

```
ALTER USER 'root'@'localhost' IDENTIFIED BY 'password';
```

- Allow root connections from other hosts

```
use mysql;
update user set host='%' where host='localhost' AND
user='root';
FLUSH PRIVILEGES;
```

Create DB and table

- Create a new database

```
CREATE DATABASE company;
```

- Create a new table employees

```
CREATE TABLE IF NOT EXISTS `company`.`employees` (  
  `id` INT AUTO_INCREMENT ,  
  `name` VARCHAR(150) NOT NULL ,  
  `picUrl` VARCHAR(150) NOT NULL ,  
  PRIMARY KEY (`id`) )  
ENGINE = InnoDB;
```

Configure the container (Frontend)

- Install at container bootstrap the libraries to implement MySQL client functionalities (add before RUN pip3 install ...)

```
RUN apk add --update --no-cache mariadb-connector-c-dev \  
    && apk add --no-cache --virtual .build-deps \  
        mariadb-dev \  
        gcc \  
        musl-dev \  
    && pip install mysqlclient==1.4.6 \  
    && apk del .build-deps
```

- Add among requirements the python MySQL libraries, add the following line in the file requirements.txt:

```
mysqlclient
```

New Employee (PUSH)

- Retrieve the IP of the DB container and connect to the DB on the python program

```
import MySQLdb
```

```
...
```

```
mydb = MySQLdb.connect(host="172.17.0.4",  
user="root",passwd="password", db="company")
```

- Store the data in the database

```
mycursor = mydb.cursor()
```

```
sql = "INSERT INTO employees (id, name, picUrl) VALUES (%s, %s,  
%s) "
```

```
val = (body.id, body.name, body.photo_urls)
```

```
mycursor.execute(sql, val)
```

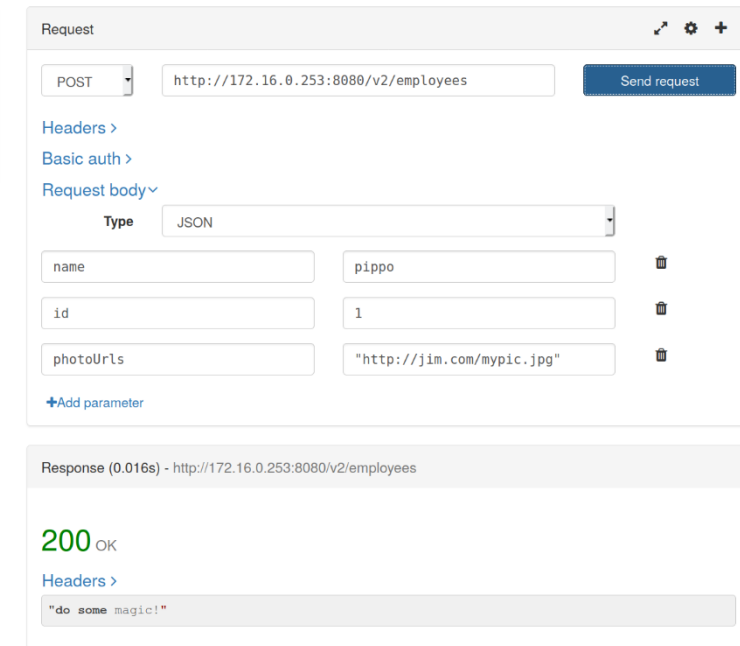
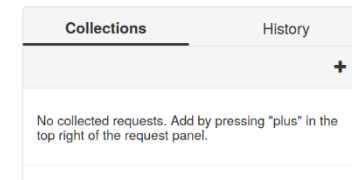
```
mydb.commit()
```

Add a new employee

- Test the program manually or via RESTED

```
curl -X POST
"http://172.16.0.253:8080/v2/employees" -H "accept:
application/json" -H "Content-
Type: application/json" -d "{
  \"id\": 0,  \"name\": \"Jim\",
  \"photoUrls\":
  \"http://mywebsite.org/mypic.jpg
  }"
```

</> RESTED



Get Employee By ID (GET)

- Use the function `get_employee_by_id`
- Retrieve the data from the DB

```
from flask import jsonify
```

Connect to the DB



```
...
```

```
sql = "SELECT * FROM employees WHERE id='{ }' ".format(employeeId)
```

```
mycursor.execute(sql)
```

```
myresult = mycursor.fetchall()
```


Issue the request



```
print(myresult)
```

```
return jsonify(name=myresult[0][1],
```

Create a response
in JSON format



```
    picUrl=myresult[0][2],
```

```
    id=myresult[0][0])
```


Retrieve info on employee #1

- `curl -X GET`
`"http://172.16.0.253:8080/v2/employees/1" -H`
`"accept: application/json"`

The screenshot shows a REST client interface with a 'Request' section and a 'Response' section. In the 'Request' section, the method is 'GET' and the URL is 'http://172.16.0.253:8080/v2/employees/1'. A 'Send request' button is visible. Below the URL, there are links for 'Headers >' and 'Basic auth >'. The 'Response' section shows a status of '200 OK' and a response time of '0.033s'. Below the status, there is a link for 'Headers >'. The response body is a JSON object:

```
{  "id": 1,  "name": "pippo",  "picUrl": "http://jim.com/mypic.jpg"}
```


Solution: python-flask-server.zip

Without SWAGGER

- Create a REST application with Flask is simple

```
#!/flask/bin/python
from flask import Flask
from flask import jsonify
app = Flask(__name__)
employee = [
    {
        'id': 1,
        'name': u'Jhon',
        'picUrl': u'http...'
    },
    ... ]
```

Define handlers for each method



```
@app.route('/v1/employees',
methods=['GET'])

def get_employees():
    return jsonify(employee)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8080)
```

Solution: custom-python-flask.zip