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 <u>not an action</u>
- Actions should be defined *via the request method and not within the name of the resource*
- For instance, the following APIs are <u>**not**</u> compliant:
  - /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 emplyees
  - <u>Includes not only the resource but also the action</u>

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 dada 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/
- Run a specific version on a container:
- docker run -d -p 9090:8080 swaggerapi/swagger-editor:v4.10.2
- Open the following page: http://10.1.X.Y:9090/

API design

- 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
- The definition is carried out by defining a YAML file, which a set of data definition and the list of the methods
- Example: Employees management system
- Data structures: Employee data structure
- Functionalities: get all the employees, add an employee, get info about a single employee, delete an employee

File openapi.yaml

REST API example

General API definition

```
openapi: 3.0.3
info:
  title: Swagger Employee
  license:
    name: Apache 2.0
version: 1.0.11
servers:
  - url: http://10.1.1.230:8080/v2
```

Paths definition

```
paths:
  /employees:
    put:
      tags:
        - employee
      operationId: updateEmployee
      requestBody:
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Employee'
          application/xml:
            schema:
              $ref: '#/components/schemas/Employee'
        required: true
```

POST Method

DATA: an instance of Employee

Response definition

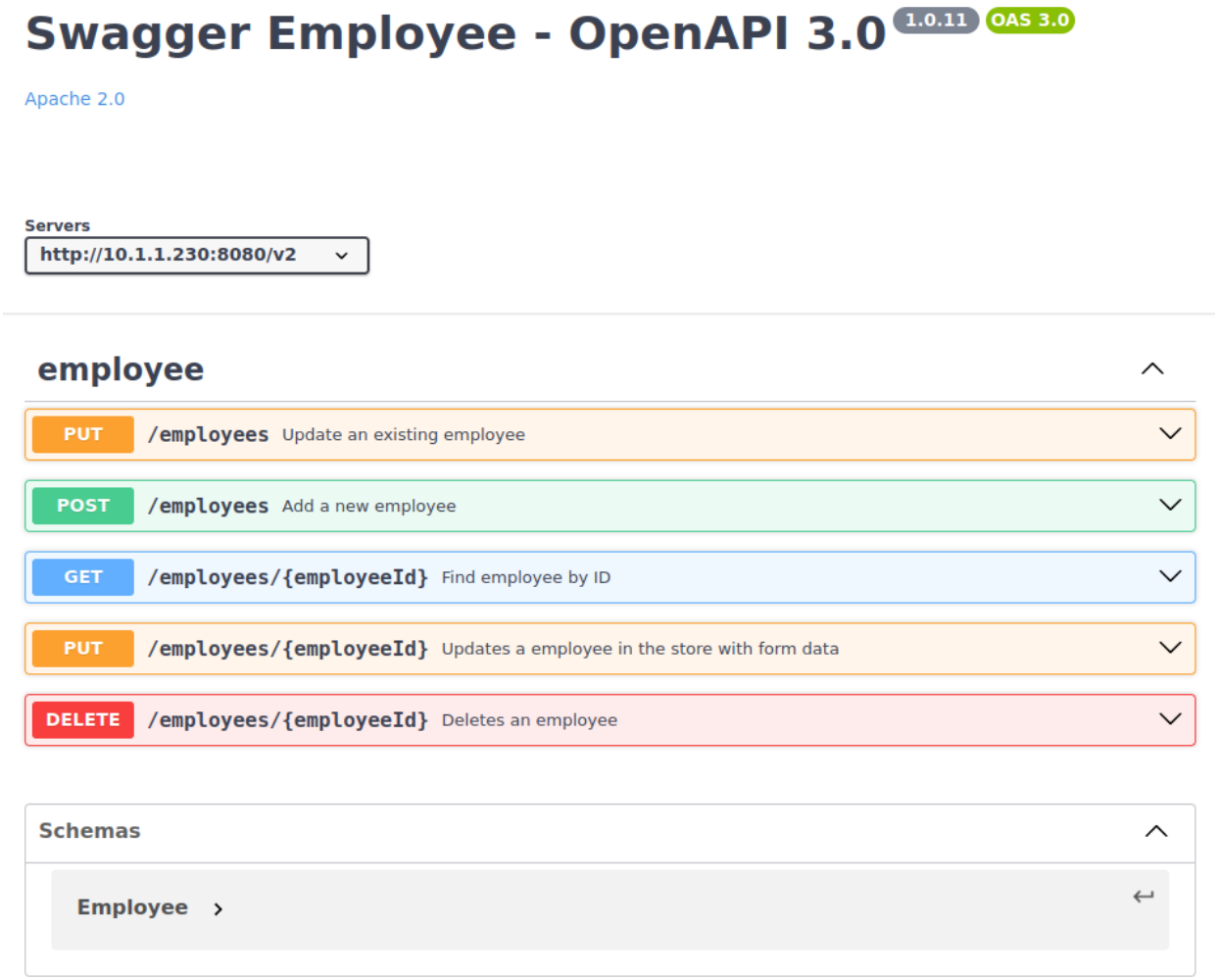
```
responses:
      '200':
        description: Successful operation
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Emplo
          application/xml:
            schema:
              $ref: '#/components/schemas/Emplo
```

Input type definition

REST API example

Employee object definition

```
components:
  schemas:
    Employee:
      required:
        - name
        - photoUrls
      type: object
      properties:
        id:
          type: integer
          format: int64
          example: 10
        name:
          type: string
          example: Carlo
        photoUrls:
          type: string
          example: "http://my.org/mypic.jpg"
```



Swagger interface

Can be used to generate curl commands for invocation

PUT /employees Update an existing employee

Update an existing employee by Id

Parameters Cancel

No parameters

Request body required application/json

Update an existent employee in the store

```
{
  "id": 10,
  "name": "Carlo",
  "photoUrls": "http://mywebsite.org/mypic.jpg"
}
```

Execute Clear

Responses

Curl

```
curl -X 'PUT' \
  'http://10.1.1.230:8080/v2/employees' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "id": 10,
  "name": "Carlo",
  "photoUrls": "http://mywebsite.org/mypic.jpg"
}'
```

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)

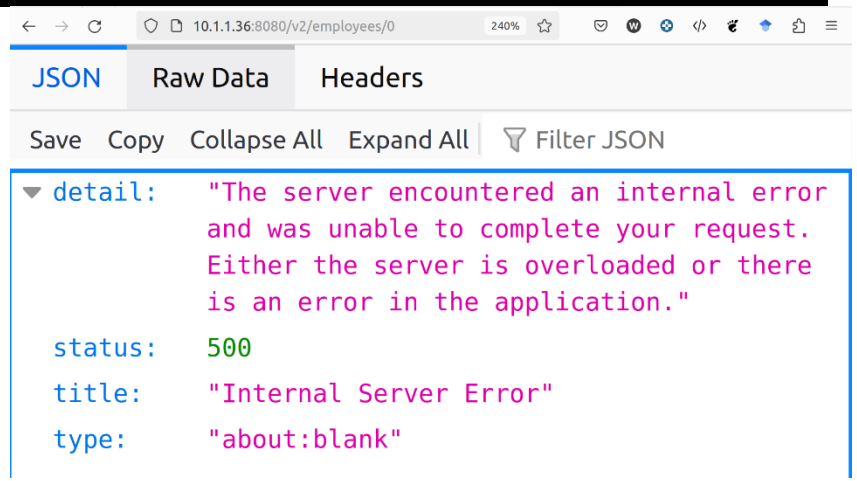
Pyhton flask

- 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

```
root@HAJJVX8OPD7M5QO:~/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)
```



The implementation is missing

- 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://10.1.Y.X:8080/v2/employees/0
- Run from the command line

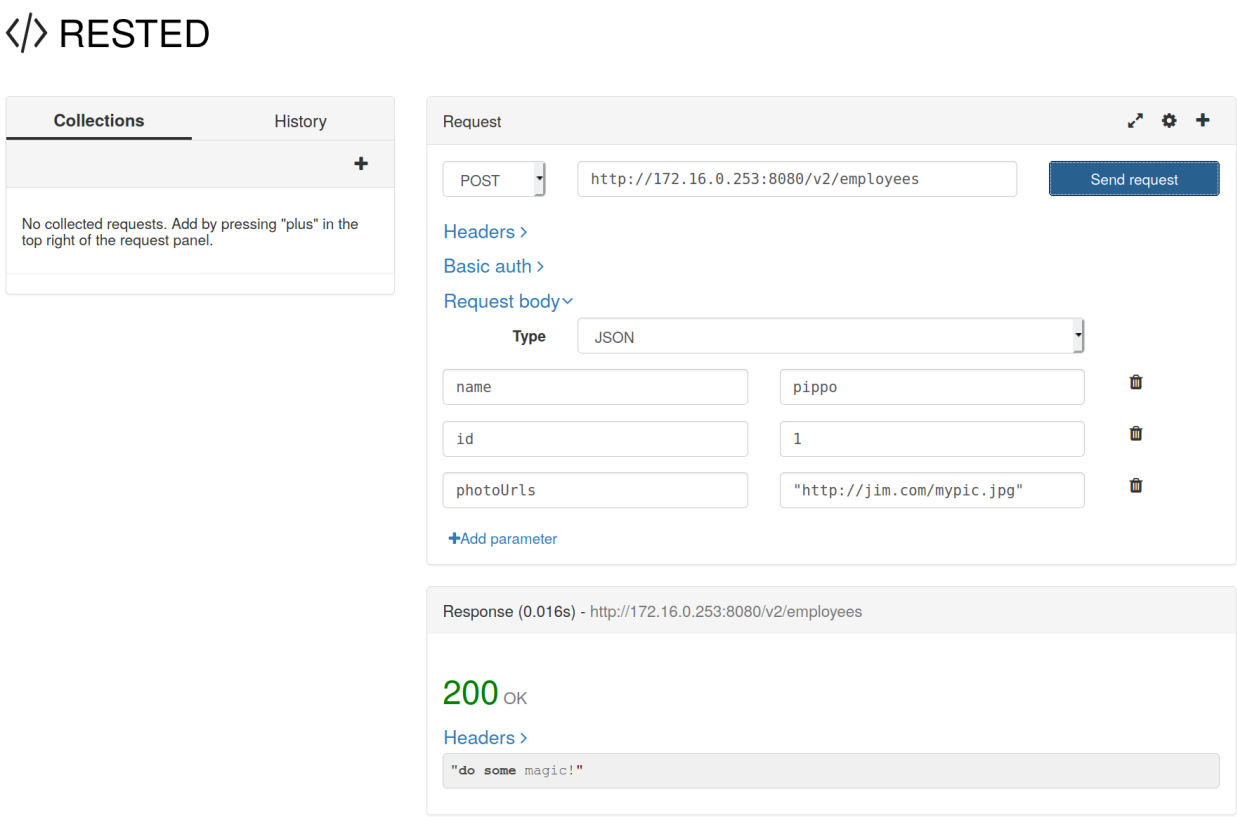
```
curl -X POST "http://10.1.Y.X:8080/v2/employees" -H
"accept: application/json" -H  "Content-Type:
application/json" -d "{  \"id\": 0,  \"name\": \"Jim\",
\"photoUrls\": \"http://mywebsite.org/mypic.jpg\"}"
```

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
- 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/employee_controller.py

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

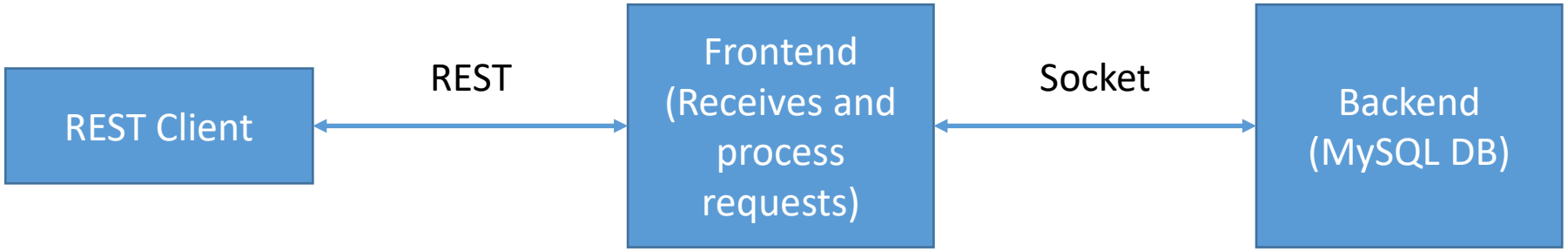
```
def add_employee(body):
  if connexion.request.is_json:
    body = Employee.from_dict(connexion.request.get_json())
  return 'do some magic!'
```

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@HAJJVX8OPD7M5QO:~/python-flask-server# docker logs mysql 2>&1 | grep GENERATED
[Entrypoint] GENERATED ROOT PASSWORD: 4DuDUPuSIs#AHukObUcopfuLujUS
```

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")
```

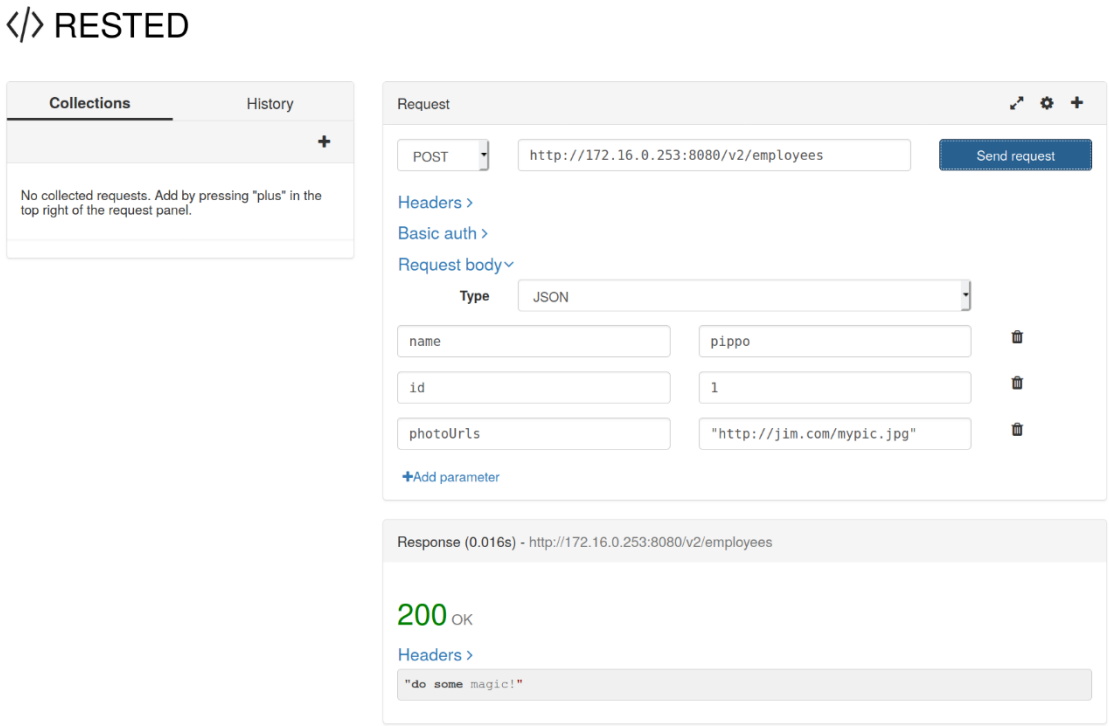
- Store the data in the database

```
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://10.1.1.X:8080/v2/employees" -H "accept: application/json" -H "Content-Type: application/json" -d "{ \"id\": 0, \"name\": \"Jim\", \"photoUrls\": \"http://mywebsite.org/mypic.jpg\"}"
```



Get Employee By ID (GET)

- Use the function **`get_employee_by_id(employee_id)`**
- Retrieve the data from the DB

```
from flask import jsonify
…
sql = "SELECT * FROM employees WHERE id='{}'".format(employee_id)
mycursor.execute(sql)
myresult = mycursor.fetchall()
print(myresult)
return jsonify(name=myresult[0][1],
               picUrl=myresult[0][2],
               id=myresult[0][0])
```

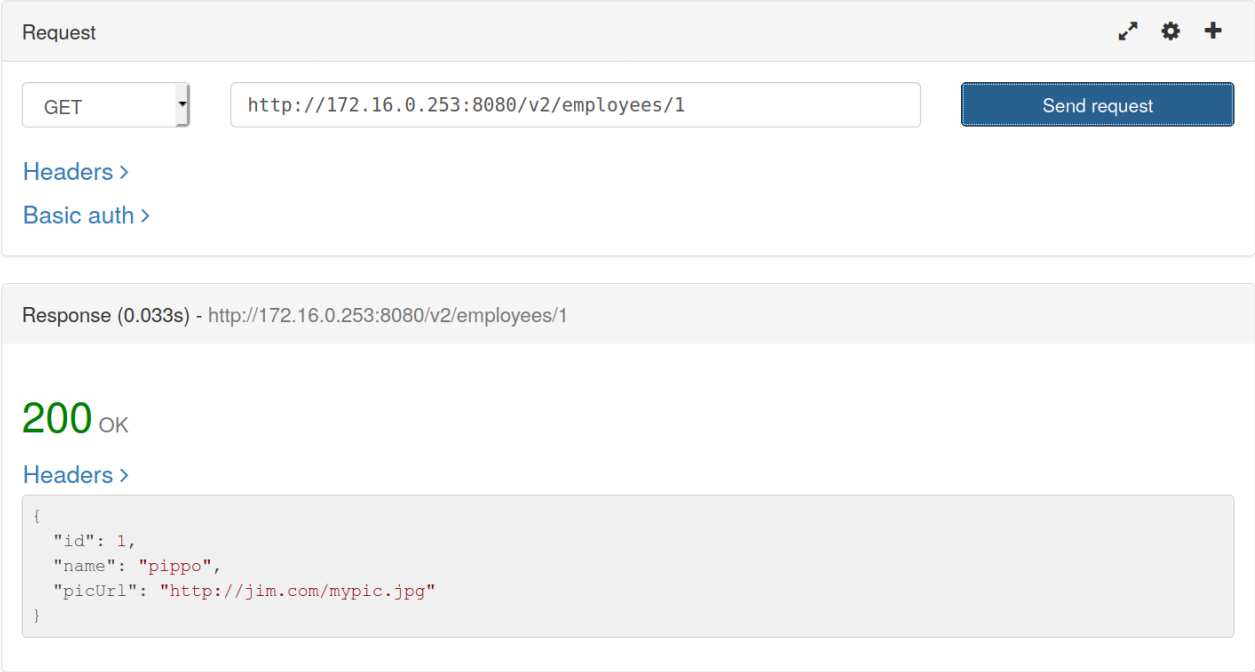
Connect to the DB

Issue the request

Create a response in JSON format

Retrieve info on employee #1

- **`curl -X GET "http://10.1.1.X:8080/v2/employees/1" -H "accept: application/json"`**



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