The lab of the second week: service-oriented applications, testing, and containerization

(http://bit.ly/2OOaNxE)

Spyridon Koulouzis <S.Koulouzis@uva.nl>
Zhiming Zhao <z.zhao@uva.nl>

1. Introduction

1.1 About the lab assignment

In this assignment you will learn how to define a simple REST API with OpenAPI and Swagger, write REST API tests using Postman, develop the application logic, dockerise it and finally perform continuous integration (CI).

Reporting

At the end of this assignment you are expected to submit the following:

- Each student should write a short report (max 1 page) with the following structure:
    - **Introduction**: List which of the DevOps stages you practiced with this assignment and what are their primary objectives.
    - **Tasks**: Using the code templates you created (swagger and python sources) answer the exercises 2.1a,2.1b 2.5a . What was the issue and how you solved it (may include screens shoots). The exercises briefly require (see the tutorial bellow for more details):
        - Define the Student's object properties
        - Fix the 'DELETE' method
        - Dockerize and push your service to docker hub. In your report you are to deliver the name of your published service in https://hub.docker.com e.g. nginx/nginx-ingress.
    - **Service Granularity** : Analyze the granularity of the service given to you in the 'student_service.py' file. What are the advantages and disadvantages of separating the database from the service? Which database technologies (type of DB e.g., MySQL, MongoDB, etc. and platform e.g. cloud based, docker container, bear metal etc.) would you use and why?

- **Advanced question 1:** deploy a separate database from the web service based on your analysis in the report. The deployment must be accessible so the tests can be run.
- **Advanced question 2:** based on the use case discussion during the lecture, please discuss the advantages and risks of applying agile approach.

Assessment

If the dockerized service passed all the tests defined in python-flask-server-generated/python-flask-server/tests you will get 80%. A passing mark will be given for the first 10 tests. The rest of the 20% will be determined by your report. In order to be given a grade you **must** submit the following:

- Written report (see above for details)
- Name of the **published** docker(s) in https://hub.docker.com/. Must be able to perform (docker pull <REPO/NAME>)
- Implementation code (git link)

Technologies / Tools Overview

- Swagger: API definition and documentation, https://swagger.io/
- Postman: API testing, https://www.getpostman.com/
- Github: Code versioning, https://github.com/
- Travis: continuous integration (CI), https://travis-ci.org/
- Docker: deployment, https://www.docker.com/

1.2. Background

OpenAPI and Swagger

Swagger is an implementation of OpenAPI. Swagger contains a tool that helps developers design, build, document, and consume RESTful Web services. Applications implemented based on OpenAPI interface files can automatically generate documentation of methods, parameters and models. This helps keep the documentation, client libraries, and source code in sync.

Postman

Postman is a Chrome-based application for testing API calls. Postman supports variables, which can simplify API testing. API testing is done to determine if a service meets expectations for functionality, reliability, performance, and security.

GitHub

GitHub is a web-based hosting service for version control using Git. Version control helps keep track of changes in a project and allows for collaboration between many developers

Travis continuous integration (CI)

Travis CI is a hosted, distributed continuous integration service used to build and test software projects hosted at GitHub. The main aim of CI is to prevent integration problems. CI is often used in combination with automated tests written through the practices of test-driven development.

Docker

Docker performs operating-system-level virtualization, also known as "containerization". Docker uses the resource isolation features of the Linux kernel to allow independent "containers" to run within a Linux instance.

1.3. Preparation

Setup github/git

If you don't have an account already follow these instructions: https://github.com/join. Next create a repository and add your team members. To do that got to the newly created repository -> Settings -> Collaborators and add the collaborators username. To be able to commit code from your machine to the repository you'll need to install git. Follow these instructions to install on your machine: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

Setup Docker Hub

If you don't have an account already follow these instructions: https://hub.docker.com/signup . Install Postman For Ubuntu 18.04 (or other Debian-based distributions) follow these instructions: http://ubuntuhandbook.org/index.php/2018/09/install-postman-app-easily-via-snap-in-ubuntu-18-04/ https://www.getpostman.com/downloads/ For macOS, windows or other linux distributions : https://www.getpostman.com/downloads/ Install Docker For Ubuntu follow these instructions: https://docs.docker.com/install/linux/docker-ce/ubuntu/ For macOS see: https://docs.docker.com/docker-for-mac/install/ For windows: https://docs.docker.com/docker-for-windows/install/ Write the API definitions To get an understanding of Swagger and OpenAPI follow this tutorial till part 5: https://apihandyman.io/writing-openapi-swagger-specification-tutorial-part-1-introduction/.

Install Postman

For Ubuntu 18.04 (or other Debian-based distributions) follow these instructions: http://ubuntuhandbook.org/index.php/2018/09/install-postman-app-easily-via-snap-in-ubuntu-18-04/ https://www.getpostman.com/downloads/ For macOS, windows or other linux distributions : https://www.getpostman.com/downloads/

Install Docker

For Ubuntu follow these instructions: https://docs.docker.com/install/linux/docker-ce/ubuntu/ For macOS see: https://docs.docker.com/docker-for-mac/install/ For windows: https://docs.docker.com/docker-for-windows/install/

2. Lab assignments of week 2

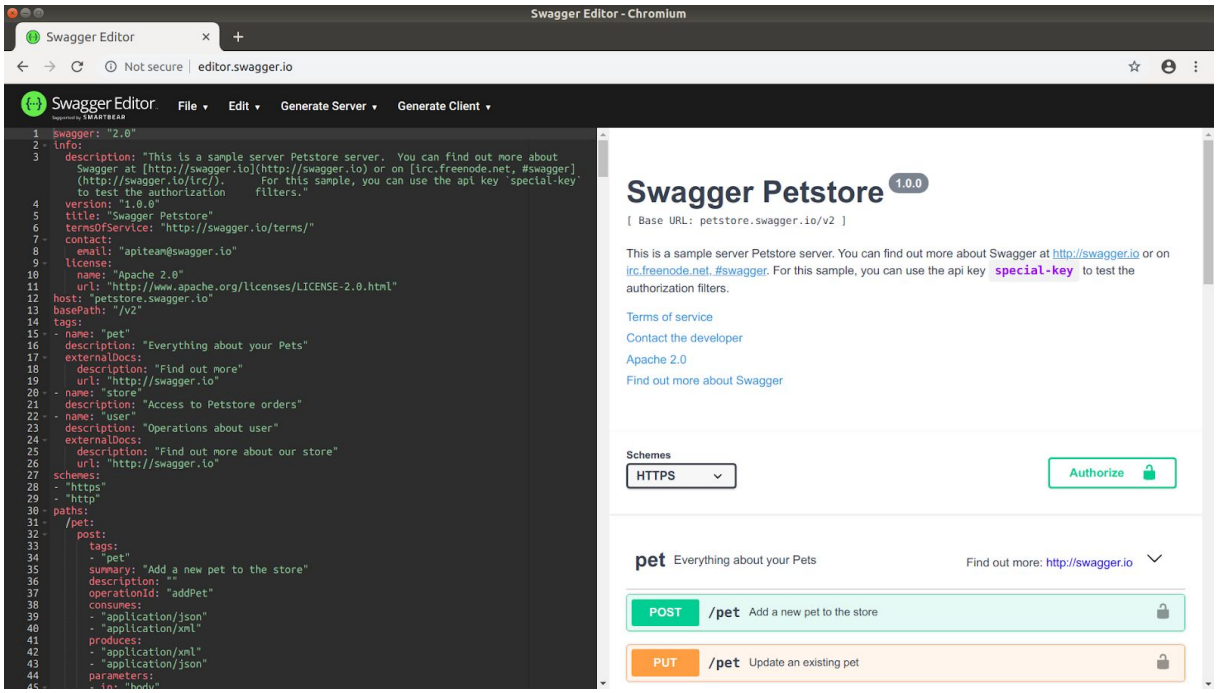
Define a simple REST API with OpenAPI and Swagger, write REST API tests using Postman, develop the application logic, dockerize it and finally perform continuous integration (CI).

2.1. Write the API definitions

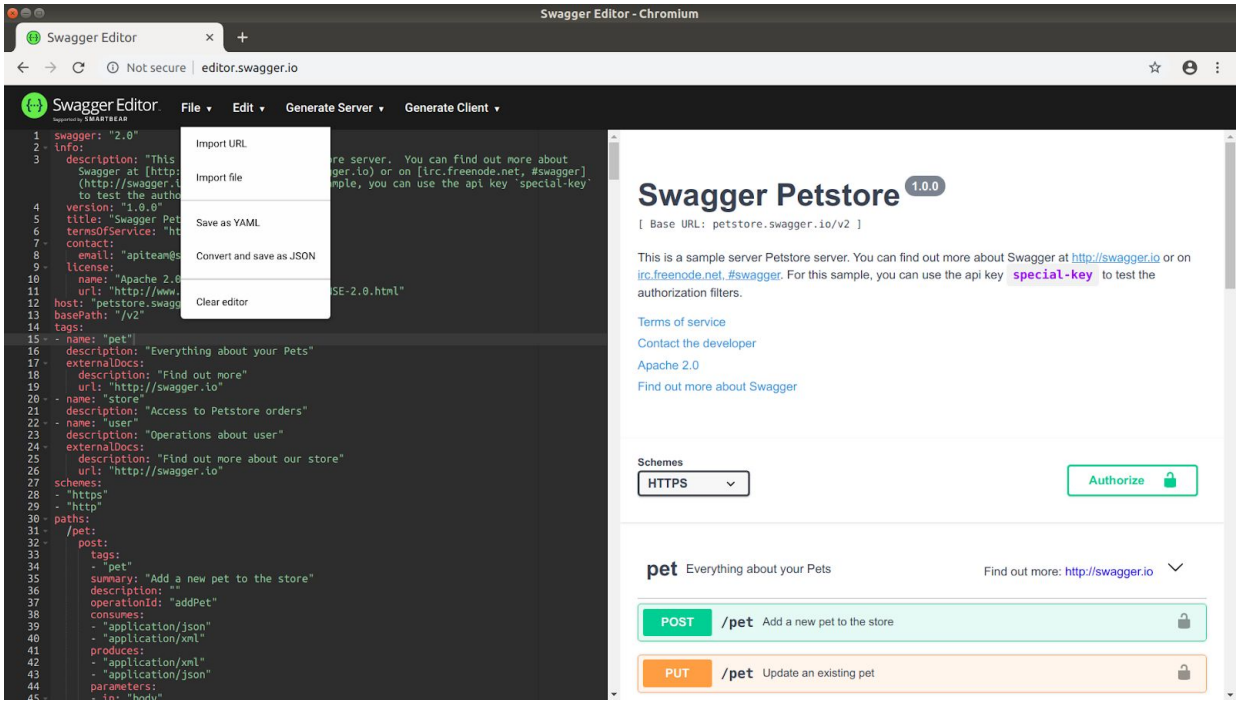
To get an understanding of Swagger and OpenAPI you may follow this tutorial till part 5: https://apihandyman.io/writing-openapi-swagger-specification-tutorial-part-1-introduction/.

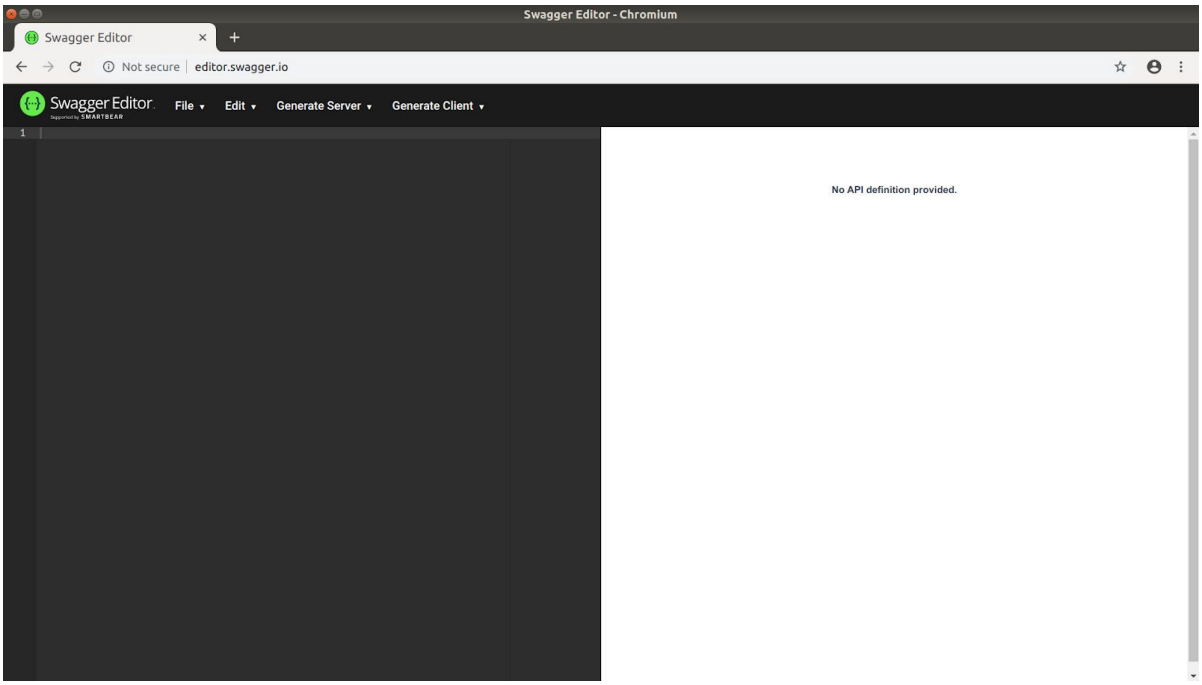
Set up example code

Open the swagger editor : http://editor.swagger.io/# . You should see the Swagger Petstore' example:



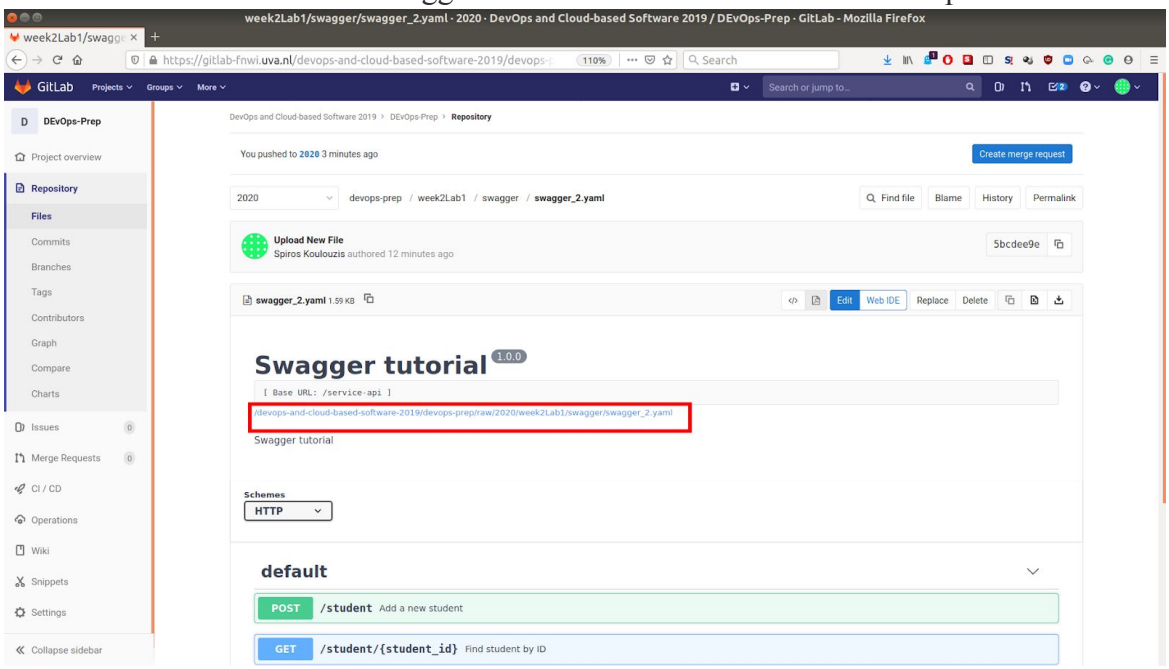
The existing code is quite complex for a first hands-on. Clear the code:





Paste the following code into the editor: swagger.yaml.

Note to see the source of the swagger definition click on the link on the top:



You will notice that the editor throws two errors:

Semantic error at paths./student.post.parameters.0.schema.$ref

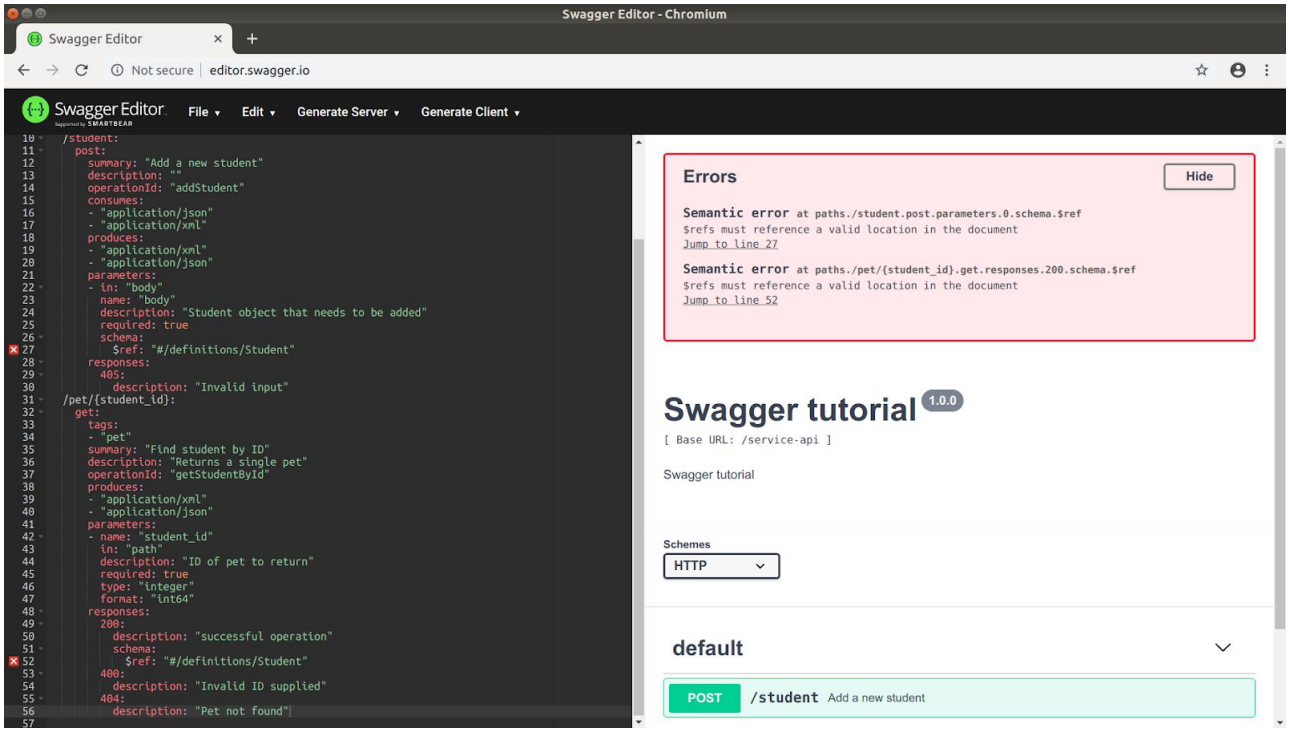
$refs must reference a valid location in the document

Jump to line 27

Semantic error at paths./student/{student_id}.get.responses.200.schema.$ref

$refs must reference a valid location in the document

Jump to line 57



Effectively what is said here is that the "#/definitions/Student" is not defined. You can find more about '$ref' here: https://swagger.io/docs/specification/using-ref/

Define Objects

Scroll down to the bottom of the page and create a new node called 'definitions' and a node 'Student' under that. The code should look like this:

```
definitions:
  Student:
    type: "object"
    properties:
```

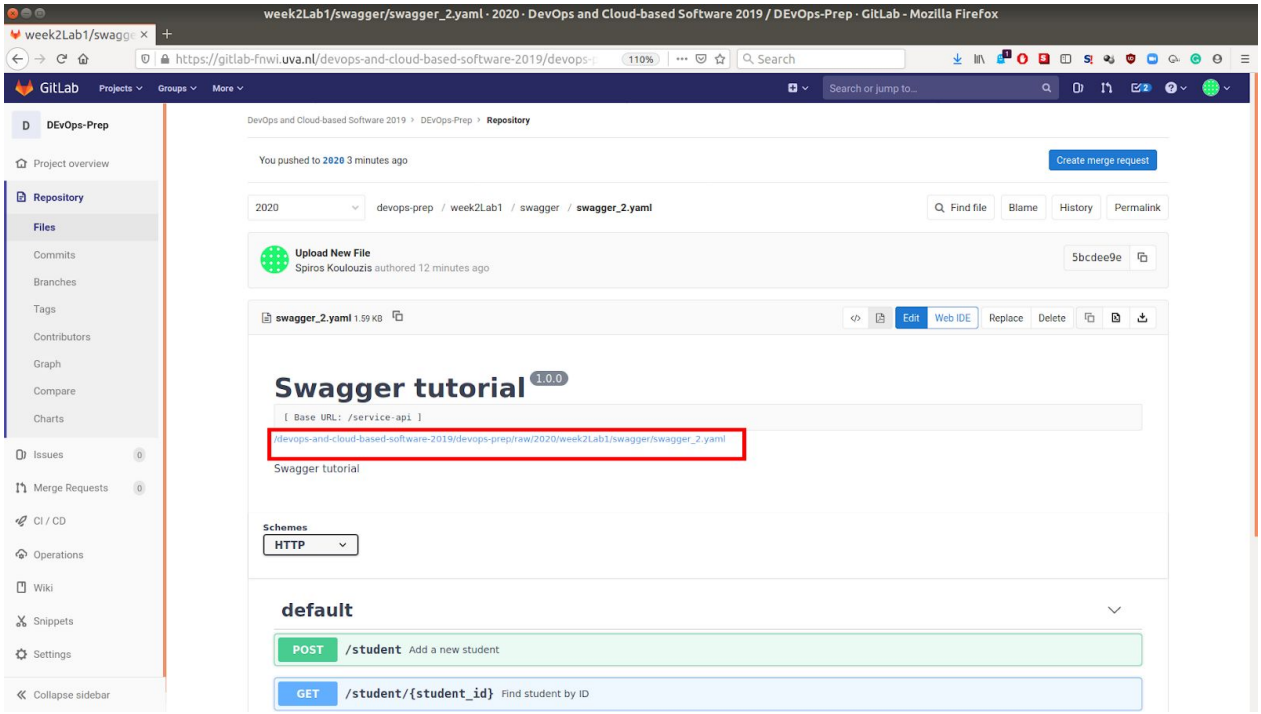
Exercise 2.1a

Define the Student's object properties. The properties to set are:

Property Name	Type
student_id	integer (int64 format)
first_name	string
last_name	string
grades	map

You can find details about data models here: https://swagger.io/docs/specification/data-models/

Note to see the source of the swagger definition click on the link on the top:



Add Delete method

The API definition at the moment only has 'GET' and 'POST' methods. We will add a 'DELETE' method Before the 'definitions' node add the following:

```
  delete:
    description: ""
    operationId: "deleteStudent"
    produces:
    - "application/xml"
    - "application/json"
    parameters:
    - name: "student_id"
      in: "path"
      description: "ID of student to return"
      required: true
      type: "integer"
      format: "int64"
    responses:
      200:
        description: "successful operation"
        schema:
```

```
          $ref: "#/definitions/Student"
      400:
        description: "Invalid ID supplied"
      404:
        description: "student not found"
```

You will notice that the editor is throwing an error:

```
Errors
Hide
Semantic error at paths./pet/{student_id}
Declared path parameter "student_id" needs to be defined within every operation in the path (missing in "delete"), or moved to the path-level parameters object
Jump to line 31
```

Exercise 2.1b

Fix the 'DELETE' method to remove this error

Add Query Parameters to Get

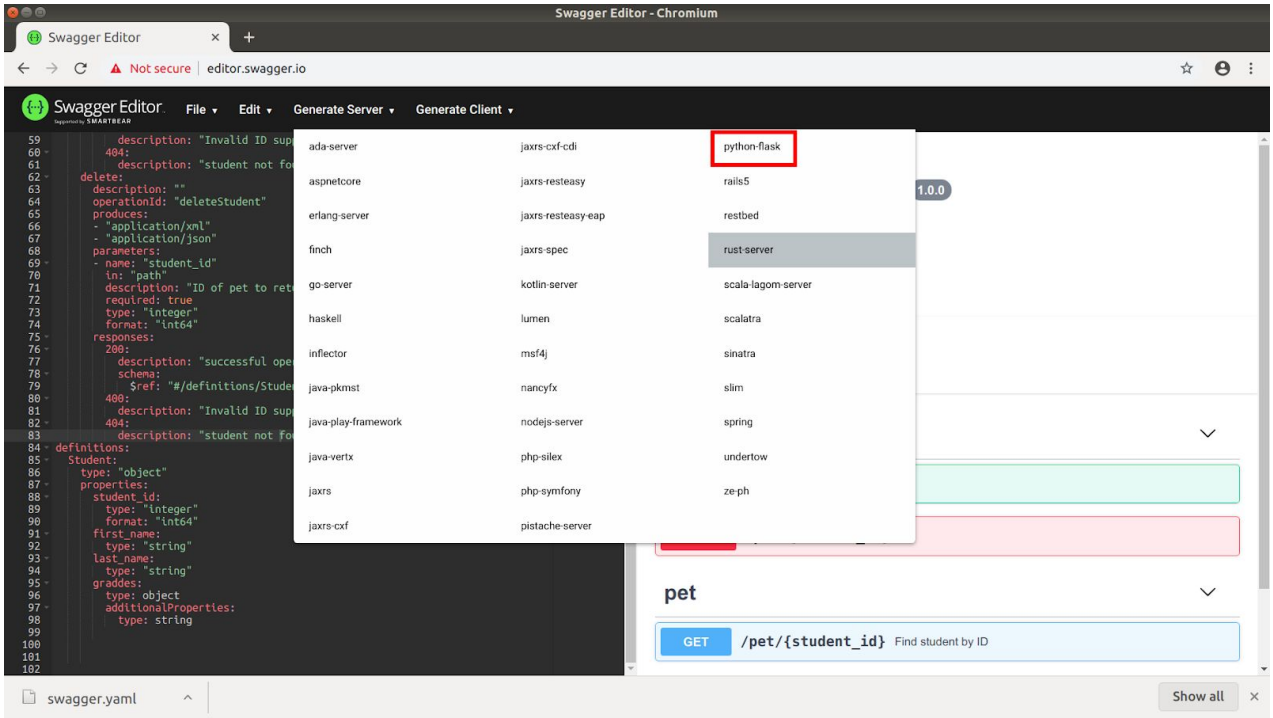
We will update the get method to include query parameters In the 'parameters' node add the following:

```
    - name: "subject"
      in: "query"
      description: "The subject name"
      required: false
      type: "string"
```

2.2. Generate the Server Code

Python-flask

In the swagger editor go to 'Generate Server' and select 'python-flask'



Update the requirements files to look like this:

- requirements.txt
- test-requirements.txt

Also the generated code comes with a bug in 'python-flask-server/swagger_server/util.py' in the '_deserialize' method. Replace the code for the '_deserialize' method with this:

```
def _deserialize(data, klass):
    """Deserializes dict, list, str into an object.
    :param data: dict, list or str.
    :param klass: class literal, or string of class name.
    :return: object.
    """
    if data is None:
        return None

    if klass in six.integer_types or klass in (float, str, bool):
        return _deserialize_primitive(data, klass)
    elif klass == object:
        return _deserialize_object(data)
    elif klass == datetime.date:
        return deserialize_date(data)
```

```
    elif klass == datetime.datetime:
        return deserialize_datetime(data)
    elif hasattr(klass, '__origin__'):
        if klass.__origin__ == list:
            return _deserialize_list(data, klass.__args__[0])
        if klass.__origin__ == dict:
            return _deserialize_dict(data, klass.__args__[1])
    else:
        return deserialize_model(data, klass)
```

Create Git Repository and Commit the Code

Create **one** git repository. The owner of the repository should invite the other team members as collaborators. You can find how to do this here:
https://help.github.com/en/github/setting-up-and-managing-your-github-user-account/inviting-collaborators-to-a-personal-repository
Go to the directory of the code and initialize the git repository and push the code:

```
git init
git add .
git commit -m "first commit"
git remote add origin <REPOSETORY_URL>
git push -u origin master
```

More information on git can be found here: https://www.tutorialspoint.com/git/index.htm

Create Python Virtual Environment

Go to your local folder in 'python-flask-server-generated/python-flask-server' and create a new Python Virtual Environment:

```
python3 -m venv venv
```

More information on Python Virtual Environment can be found here:
https://docs.python.org/3/tutorial/venv.html

Add Edit .gitignore file

Because you don't want to push the entire venv folder in git add/edit the '.gitignore' file to look like this:
.gitignore

Install Requirements and Run

Go to 'python-flask-server-generated/python-flask-server' and install the project requirements :

```
./venv/bin/pip3 install --no-cache-dir -r requirements.txt
```

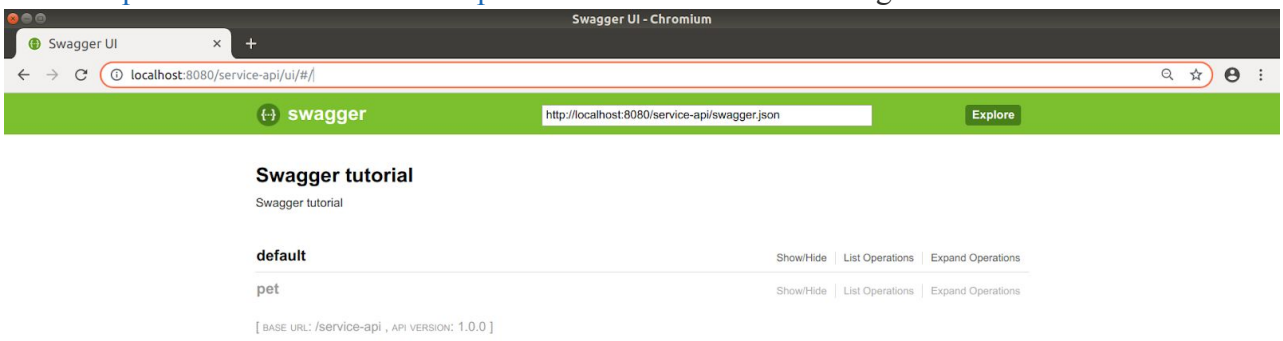
and the test requirements:

```
./venv/bin/pip3 install --no-cache-dir -r python-flask-server/test-requirements.txt
```

Go to python-flask-server-generated/python-flask-server Run the service:

```
./venv/bin/python3 -m swagger_server
```

Go to: http://localhost:8080/service-api/ui/ You should see something like this:



You can now shutdown the server

2.3. Write Unit Tests

Python

The code generated has also created a TestCase. Go to 'python-flask-server-generated/python-flask-server/swagger_server/test' and open the 'test_default_controller.py' file. There you can add the tests 'test_add_student' and test_delete_student. Here is the test_default_controller.py
After you are done commit the code.

Develop The Application Logic

Write the code for the get, delete and put methods.

In general it is a good idea to write application using layered architecture. By segregating an application into tiers, a developer can modify or add a layer, instead of reworking the entire application.
This is why we should create a new package in the code called 'service' and a python file named 'student_service.py'. Here is a template of such a file: student_service.py. In this code template we use a simple file-based database to store and query data called TinyDB. More information on TinyDB can be found here: https://tinydb.readthedocs.io/en/latest/getting-started.html
Now the controller just needs to call the service's methods: default_controller.py
After you are done commit the code.

Create a Board

All code repositories have some kind of board to manage projects. Create a board and add three issues: 'develop add_student', 'develop delete_student', 'develop get_student_by_id'. Assign each issue to one person and go back to the project you just created and add the issues to the board in the 'To do' column.

Create a branch and Merge

Each member of the team should check out the master branch.

```
git checkout master
```

Each member of the team create the issue branch assigned to you

```
git checkout -b <issue_num>
```

Where you see <issue_num> you should add the issue number assigned to you
Develop the code and commit to your branch

```
git add .
git commit -a -m 'added a get_student_by_id [<issue_num>]'
git push
```

As soon as you have fixed the issue assigned to you ask for a pull request

2.4. Continues Testing and Integration

Travis CI

Travis is hosted at GitHub and it allows to build test and integrate your code.To enable CI with travis you should add in the root project directory a file named .travis.yml. This file will define the steps to build and test your code every time you commit new changes. Your .travis.yml will look like this: .travis.yml
To be able to perform CD we need to push the Docker image to dockerhub so it may be pulled to the production environment. To do this we need to provide to Travis our dockerhub credentials without making them openly available. To do this you need to install travis gem. This is possible by typing:

```
gem install travis
```

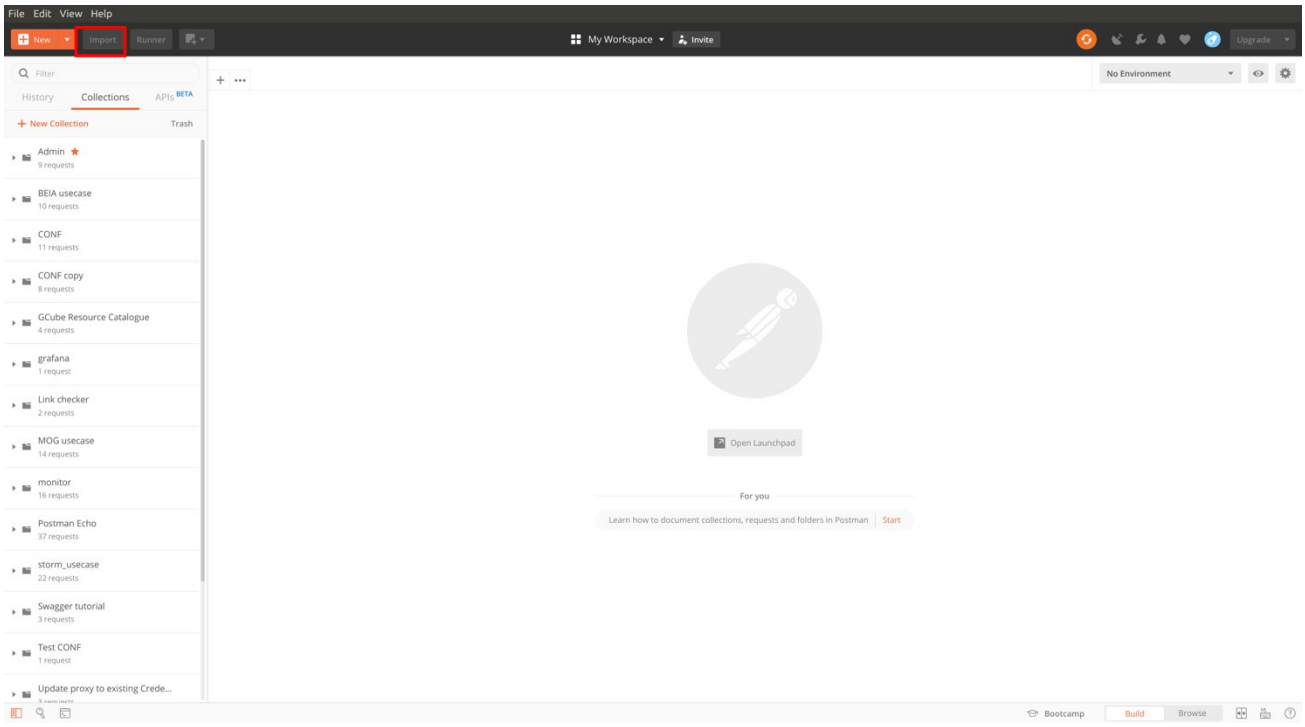
To encrypt your variables type root project directory where .travis.yml is:

```
travis encrypt DOCKER_USER=username --add env.global --com
travis encrypt DOCKER_PASS=password --add env.global --com
```

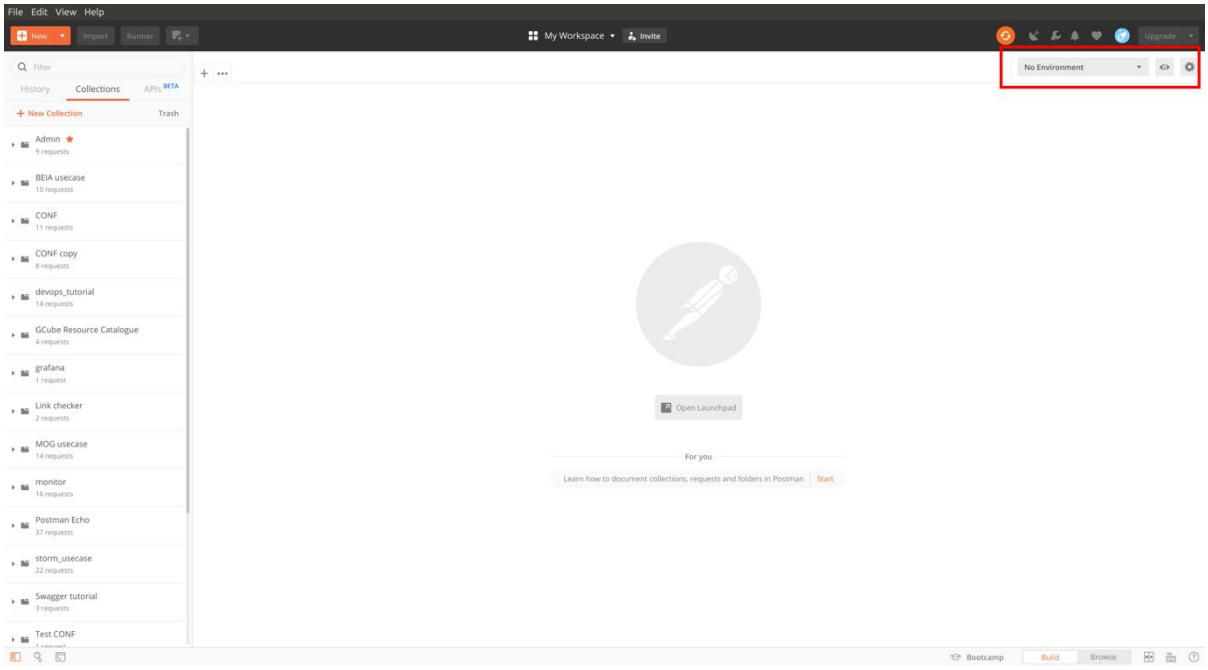
If not work, maybe login travis at first, use:

```
travis login --pro
```

This will append these variables encrypted in the .travis.yml file. More details can be found here: https://docs.travis-ci.com/user/environment-variables#defining-encrypted-variables-in-travisyml
More details can be found here: https://learning.getpostman.com/docs/postman/collection_runs/integration_with_travis/
Also in the env: section of the file you will need to add the output from the encryption step. Moreover do not forget to replace: 'my_docker_repository' with your actual dockerhub repository. More details on travis can be found here: https://docs.travis-ci.com/user/tutorial/
As you notice in the last line of the .travis.yml file we are using two files: tests/postman_collection.json and tests/postman_envirmoent.json These files are tests and environment variables defined in Postman. if you want to run the tests on Postman with the service running on your local machine import postman_collection.json and postman_envirmoent.json to postman by selecting 'Import':



Next, select the environment from the top right of the window:



2.5. Docker

Follow the tutorial here to build and publish your docker container:
https://docker-curriculum.com/

Exercise 2.5a

Using the Dockerfile build and push your docker container

3. Reporting

3.1. Requirements

At the end of this assignment you are expected to submit the following:

- Each student should write a short report (max 1 page) with the following structure:
  - **Introduction**: List which of the DevOps stages you practiced with this assignment and what are their primary objectives.
  - **Tasks**: Using the code templates you created (swagger and python sources) answer the exercises 2.1a,2.1b 2.5a . What was the issue and how you solved it (may include screens shoots). The exercises briefly require (see the tutorial bellow for more details):
    - Define the Student's object properties
    - Fix the 'DELETE' method

        - Dockerize and push your service to docker hub. In your report you are to deliver the name of your published service in https://hub.docker.com e.g. nginx/nginx-ingress.
    - **Service Granularity** : Analyze the granularity of the service given to you in the 'student_service.py' file. What are the advantages and disadvantages of separating the database from the service? Which database technologies (type of DB e.g., MySQL, MongoDB, etc. and platform e.g. cloud based, docker container, bear metal etc.) would you use and why?

- **Advanced question 1:** deploy a separate database from the web service based on your analysis in the report. The deployment must be accessible so the tests can be run.
- **Advanced question 2:** based on the use case discussion during the lecture, please discuss the advantages and risks of applying agile approach.

3.1. Assessment

If the dockerized service passed all the tests defined in python-flask-server-generated/python-flask-server/tests you will get 80%. A passing mark will be given for the first 10 tests. The rest of the 20% will be determined by your report. In order to be given a grade you **must** submit the following:

- Written report (see above for details)
- Name of the **published** docker(s) in https://hub.docker.com/. Must be able to perform (docker pull <REPO/NAME>)
- Implementation code (git link)
