DevOps and Cloud Based Software

Lab 1-1: RESTful services, Docker and Kubernetes

University of Amsterdam

Introduction

This tutorial will use OpenAPI to define a RESTful web service and Python to implement it.

The RESTful web service will use a database to store data.

More specifically, the steps of this tutorial are the following:

1. Write OpenAPI definition using SwaggerHub
2. Generate the service stubs in Python
3. Implement the logic
4. Build Test ands Docker Image
5. Write Tests
6. Deploy Web Service on Kubernetes (MicroK8s)

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.
You can find a short technical explanation here

Git

Git is an open-source distributed version control system. Version control helps keep track of changes in a project and allows for collaboration between many developers.
You can find a short technical explanation here

GitHub Actions

GitHub Actions automates your software development workflows from within GitHub. In GitHub Actions, a workflow is an automated process that you set up in your GitHub repository. You can build, test, package, release, or deploy any project on GitHub as a workflow.

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.
You can find a short technical explanation on containerization here

Kubernetes (MicroK8s)

Kubernetes is an open-source container orchestration system for automating software deployment, scaling, and management.
You can find a short technical explanation on container orchestration here

Prepare your Development Environment

Create GitHub Account

In case you don’t have a GitHub account, follow these instructions to create one: https://github.com/join

Setup Docker Hub

In case you don't have a Dock Hub account, follow these instructions to create one: https://hub.docker.com/signup

SwaggerHub Account

If you have a GitHub account, you may go to https://app.SwaggerHub.com/login
and select 'Log In with GitHub'. Alternatively, you can select to sign up.

Install Docker and Docker Compose on your Local machine

You can find instructions on how to install Docker here: https://docs.docker.com/get-docker/
You may also find a detailed tutorial on Docker here: https://docker-curriculum.com/

To test if your installation is running, you may test docker by typing:

```
docker run hello-world
```

You can find instructions on how to install Docker Compose here: https://docs.docker.com/compose/install/

Install Pycharm

In this tutorial, we will use the Pycharm Integrated Development Environment (IDE). If you have a preferred IDE you are free to use it.

You can find instructions on how to install Pycharm here: https://www.jetbrains.com/pycharm/download/

If you are using snap, you can type:

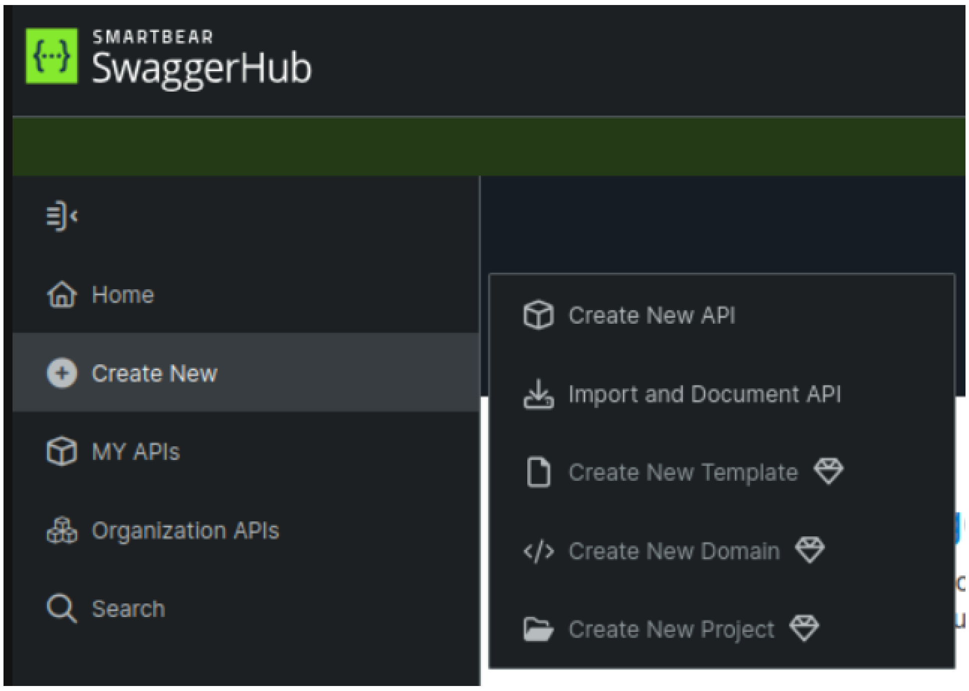
```
sudo snap install pycharm-community --classic
```

You may also find a detailed tutorial on Pycharm here:
https://www.jetbrains.com/help/pycharm/creating-and-running-your-first-python-project.html

Write OpenAPI Definition

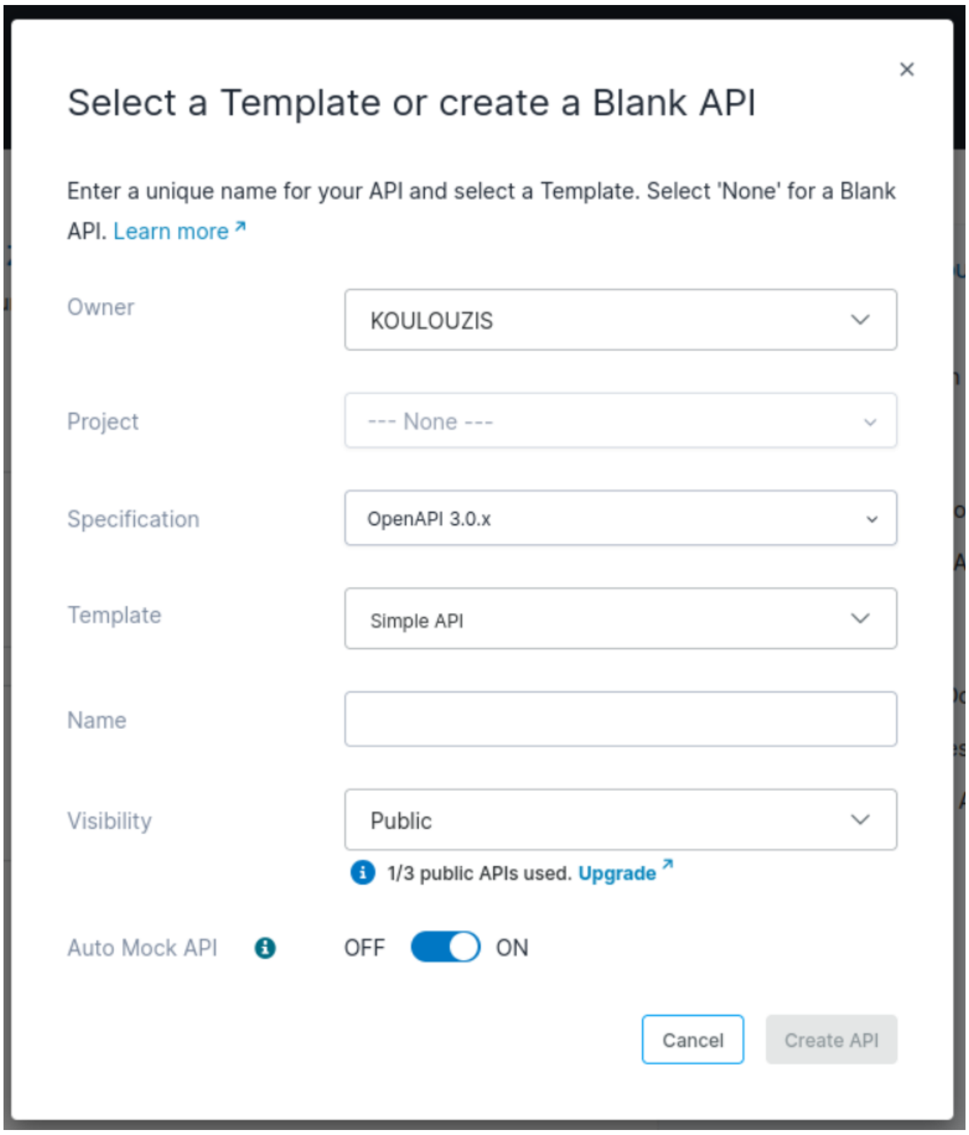
In this section, we will define a web service interface that will support the Create, Read, Update, Delete (CRUD) pattern for manipulating resources using OpenAPI. To get a more in-depth understanding of Swagger and OpenAPI you may follow this tutorial https://idratherbewriting.com/learnapidoc/openapi_tutorial.html

Log in to your SwaggerHub account at https://app.SwaggerHub.com/login and select 'Create New' -> 'Create New API'



Name your API 'tutorial'.

Then select version 3.0.x and 'Template' 'Simple API' and press 'CREATE API'.

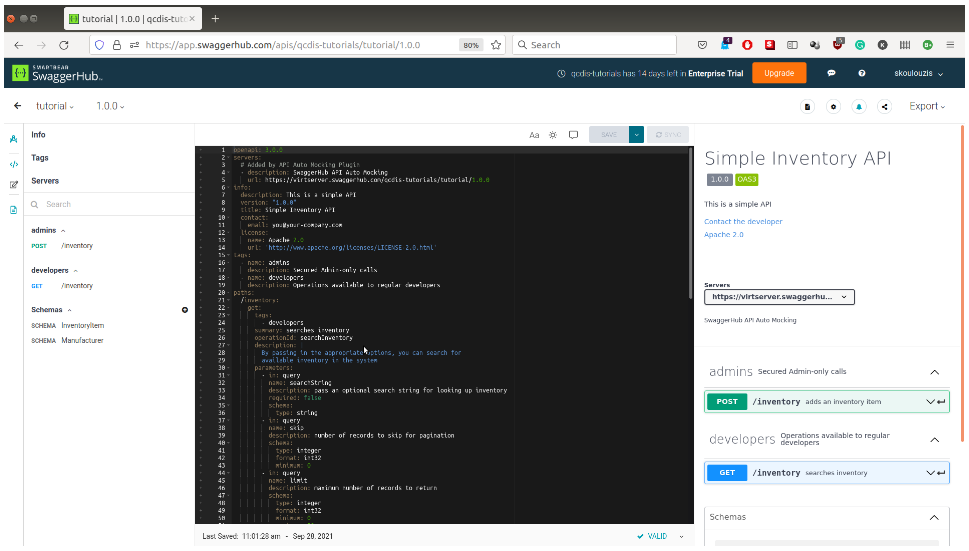


Select a Template or create a Blank API

Enter a unique name for your API and select a Template. Select 'None' for a Blank API. Learn more

Owner	KOULOUZIS
Project	--- None ---
Specification	OpenAPI 3.0.x
Template	Simple API
Name	
Visibility	Public
	1/3 public APIs used. Upgrade
Auto Mock API	OFF [✓ ON]

You will get an OpenAPI template



```
openapi: 3.0.0
servers:
  # Added by API Auto Mocking Plugin
  - description: SwaggerHub API Auto Mocking
    url: https://virtserver.swaggerhub.com/qcdis-tutorials/tutorial/1.0.0
info:
  description: This is a simple API
  version: "1.0.0"
  title: Simple Inventory API
  contact:
    email: you@your-company.com
  license:
    name: Apache 2.0
    url: 'http://www.apache.org/licenses/LICENSE-2.0.html'
tags:
  - name: admins
    description: Secured Admin-only calls
  - name: developers
    description: Operations available to regular developers
paths:
  /inventory:
    get:
      tags:
        - developers
      summary: searches inventory
      operationId: searchInventory
      description: |
        By passing in the appropriate options, you can search for
        available inventory in the system
      parameters:
        - in: query
          name: searchString
          description: pass an optional search string for looking up inventory
          required: false
          schema:
            type: string
        - in: query
          name: skip
          description: number of records to skip for pagination
          schema:
            type: integer
            format: int32
            minimum: 0
        - in: query
          name: limit
          description: maximum number of records to return
          schema:
            type: integer
            format: int32
            minimum: 0
```

Last Saved: 11:01:28 am - Sep 28, 2021 ✓ VALID

Simple Inventory API
1.0.0 OAS3

This is a simple API

Contact the developer
Apache 2.0

Servers
https://virtserver.swaggerhu...
SwaggerHub API Auto Mocking

admins Secured Admin-only calls
POST /inventory adds an inventory item

developers Operations available to regular developers
GET /inventory searches inventory

Schemas

Replace the definition with the following: openAPI_1.yaml

You will notice that the editor at the bottom throws some errors:

```
Errors (2)
$refs must reference a valid location in the document
$refs must reference a valid location in the document
```

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

OpenAPI Exercises

Define Objects

Scroll down to the bottom of the page and create the following nodes:

- components
  - schemas
    - Student
    - GradeRecord

The code should look like this:

```
components:
  schemas:
    Student:
      type: object
      required:
        - first_name
        - last_name
      properties:
        ...
    GradeRecord:
      type: object
      required:
        - subject_name
        - grade
      properties:
        ...
```

Define the Student's and GradeRecord object properties.

The Student properties to define are:

Property Name	Type
student_id	number (integer format)
first_name	string
last_name	string
grade_records	array

The GradeRecord properties to define are:

Property Name	Type
subject_name	string
grade	number (float format, minimum: 0, maximum: 10)

NOTE

Note which properties are required and which are not. This will affect the services' validation process.
To get more information on the 'required' see: https://swagger.io/docs/specification/data-models/data-types/ in the 'Required Properties' Section.

It is helpful to add 'example' fields in the properties. That way, your API is easier to consume.

You can find details about the 'example' field here: https://swagger.io/docs/specification/adding-examples/
You can find details about data models here: https://swagger.io/docs/specification/data-models/

Add Delete method

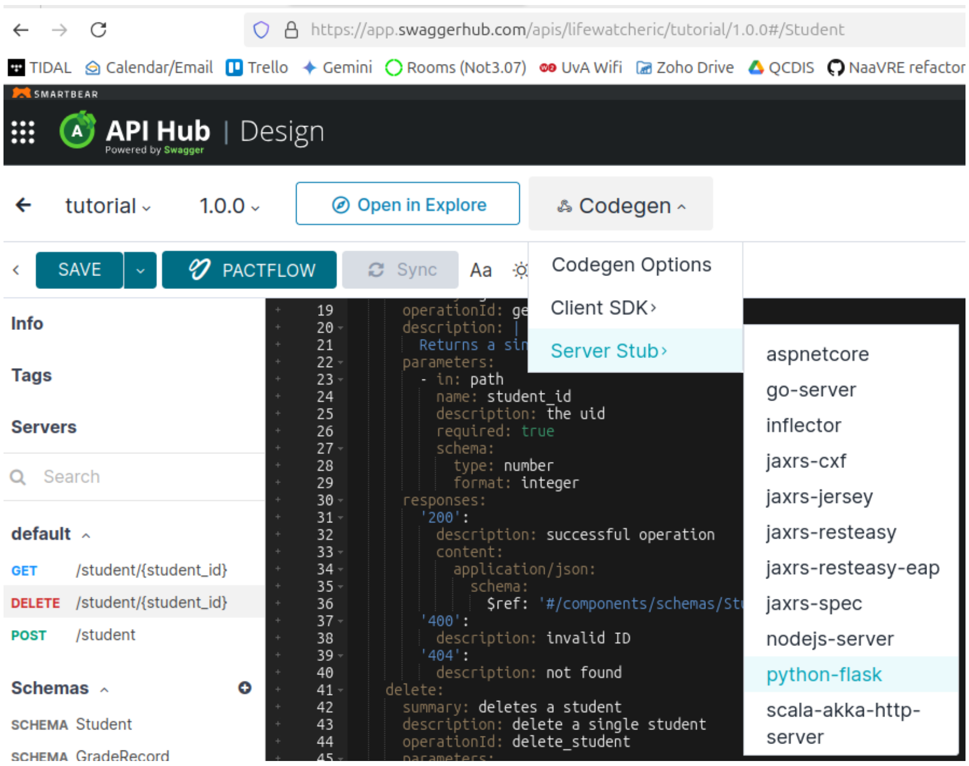
At the moment, the API definition only has 'GET' and 'POST' methods. We will add a 'DELETE' method. Under the '/student/{student_id}' path add the following:

```
    delete:
      summary: deletes a student
      description: |
        delete a single student
      operationId: delete_student
      parameters:
      - name: student_id
        in: path
        description: the uid
        required: true
        schema:
          type: number
          format: integer
      responses:
        "200":

        "400":

        "404":
```

You will need to fill in the proper responses for 200, 400, and 404. More information about responses can be found here: https://swagger.io/docs/specification/describing-responses/

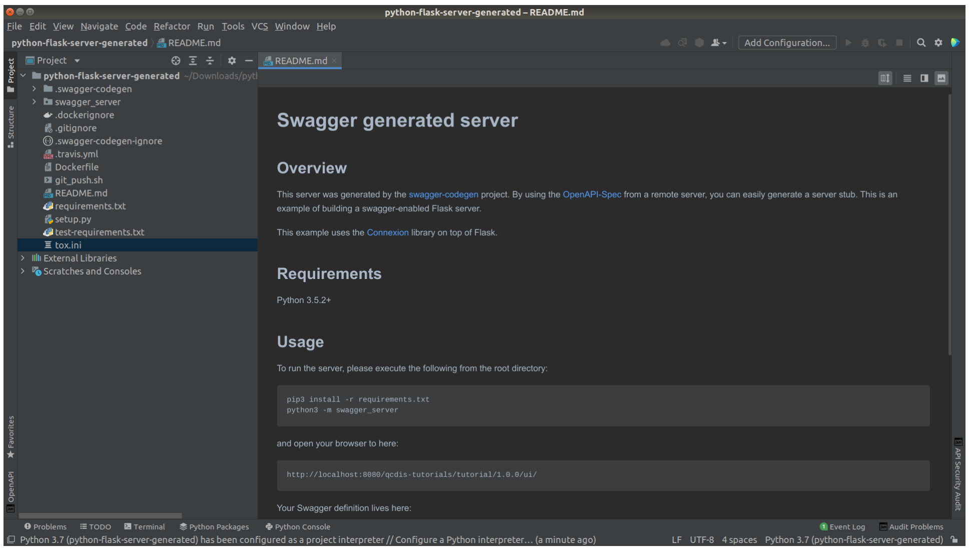
Generate Python Code

Now that we have the OpenAPI definitions, we can create the server stub on Python. Select 'Codegen'->'Server Stub'->'python-flask'



```
      operationId: ge
      description: |
        Returns a sin
      parameters:
        - in: path
          name: student_id
          description: the uid
          required: true
          schema:
            type: number
            format: integer
      responses:
        '200':
          description: successful operation
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/St
        '400':
          description: invalid ID
        '404':
          description: not found
    delete:
      summary: deletes a student
      description: delete a single student
      operationId: delete_student
      parameters:
```

Save the 'python-flask-server-generated.zip' and unzip the archive. Open Pycharm and open the project.



Swagger generated server

Overview

This server was generated by the swagger-codegen project. By using the OpenAPI-Spec from a remote server, you can easily generate a server stub. This is an example of building a swagger-enabled Flask server.

This example uses the Connexion library on top of Flask.

Requirements

Python 3.5.2+

Usage

To run the server, please execute the following from the root directory:

```
pip3 install -r requirements.txt
python3 -m swagger_server
```

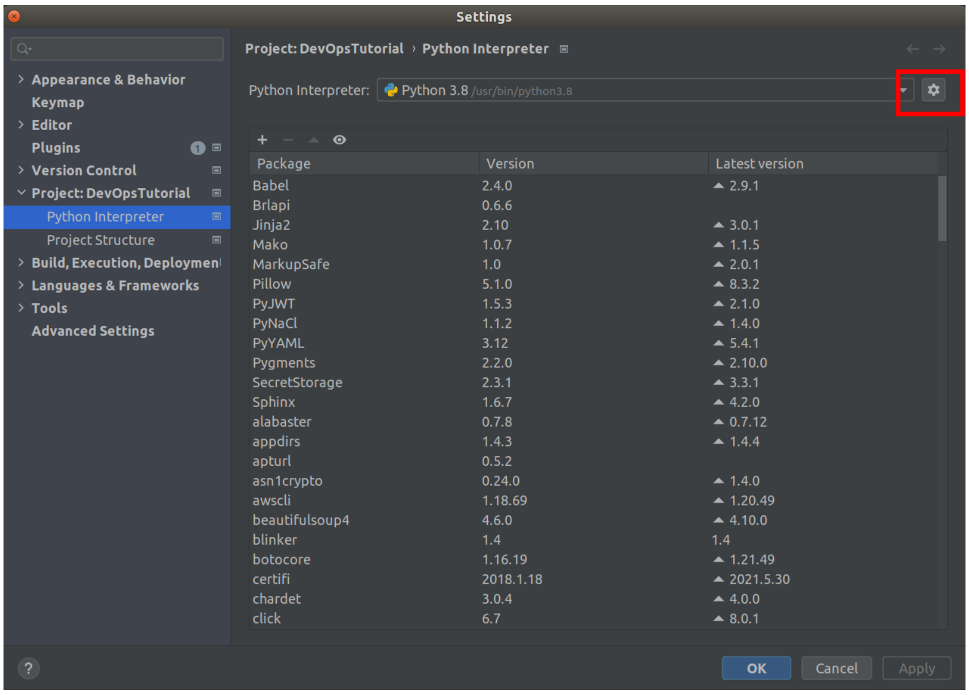
and open your browser to here:

```
http://localhost:8080/qcdis-tutorials/tutorial/1.0.0/ui/
```

Your Swagger definition lives here:

Python 3.7 (python-flask-server-generated) has been configured as a project interpreter // Configure a Python interpreter... (a minute ago)   LF   UTF-8   4 spaces   Python 3.7 (python-flask-server-generated)

To create the virtual environment for the project, go to 'File'->'Settings'->'Project'->'Python Interpreter' or ‘Pycharm'->'Preferences'->Project'->'Python Interpreter'. Select Python version 3.8 or later. Then select the gear icon to add a new environment:



Project: DevOpsTutorial › Python Interpreter

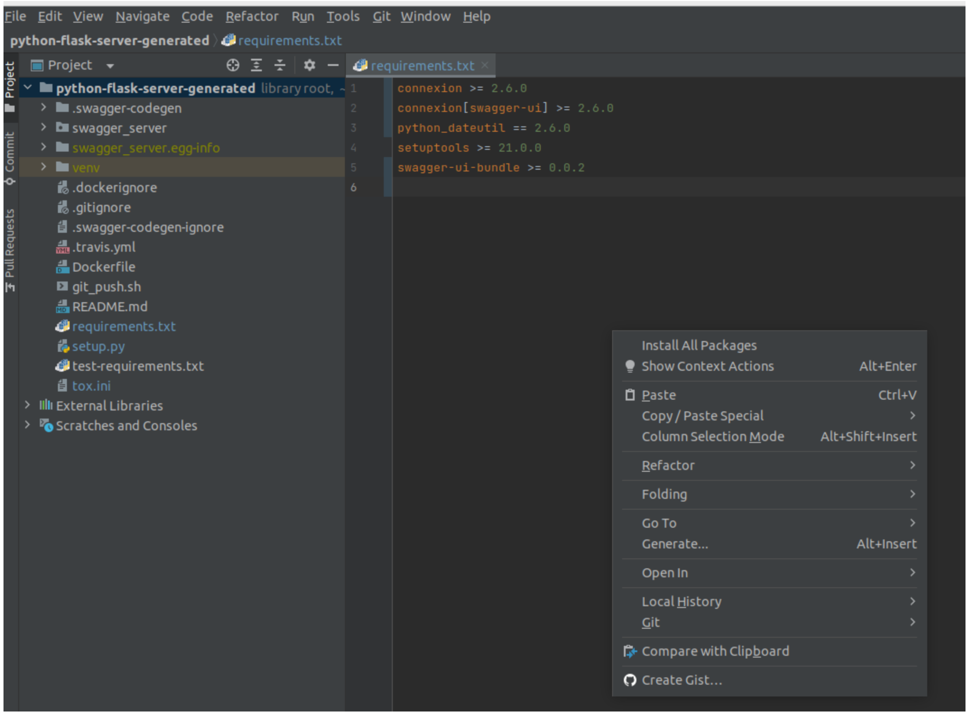
Python Interpreter: Python 3.8 /usr/bin/python3.8

Package	Version	Latest version
Babel	2.4.0	▲ 2.9.1
Brlapi	0.6.6	
Jinja2	2.10	▲ 3.0.1
Mako	1.0.7	▲ 1.1.5
MarkupSafe	1.0	▲ 2.0.1
Pillow	5.1.0	▲ 8.3.2
PyJWT	1.5.3	▲ 2.1.0
PyNaCl	1.1.2	▲ 1.4.0
PyYAML	3.12	▲ 5.4.1
Pygments	2.2.0	▲ 2.10.0
SecretStorage	2.3.1	▲ 3.3.1
Sphinx	1.6.7	▲ 4.2.0
alabaster	0.7.8	▲ 0.7.12
appdirs	1.4.3	▲ 1.4.4
apturl	0.5.2	
asn1crypto	0.24.0	▲ 1.4.0
awscli	1.18.69	▲ 1.20.49
beautifulsoup4	4.6.0	▲ 4.10.0
blinker	1.4	1.4
botocore	1.16.19	▲ 1.21.49
certifi	2018.1.18	▲ 2021.5.30
chardet	3.0.4	▲ 4.0.0
click	6.7	▲ 8.0.1

Select 'New environment' and press 'OK'

Replace the 'requirements.txt' file with this requirements.txt

Open the 'requirements.txt' file and right click and select install all packages.



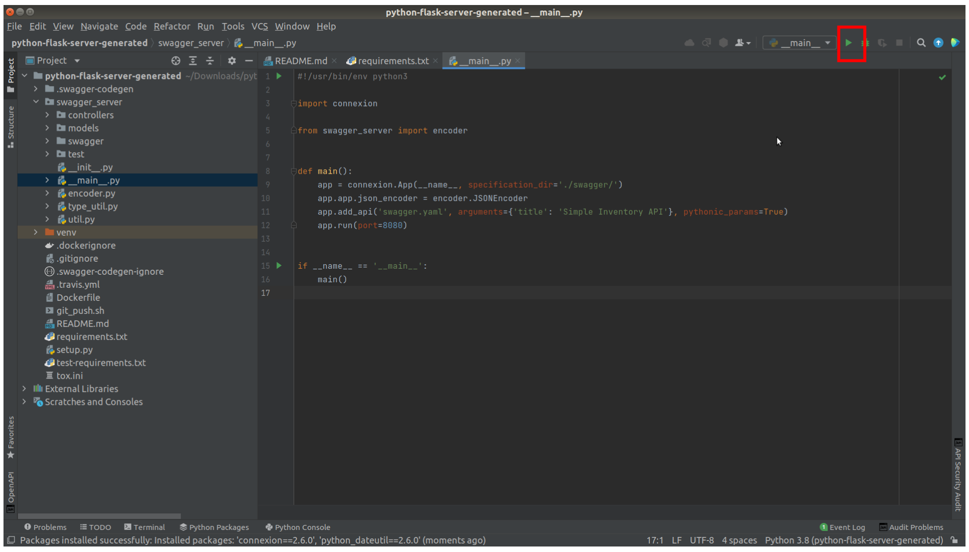
```
connexion >= 2.6.0
connexion[swagger-ui] >= 2.6.0
python_dateutil == 2.6.0
setuptools >= 21.0.0
swagger-ui-bundle >= 0.0.2
```

Open the file 'swagger_server/swagger/swagger.yaml' and in the section 'servers' you should have only one url and description. The servers section should look like this:

```
servers:
- url: https://virtserver.swaggerhub.com/tutorial/1.0.0
  description: SwaggerHub API Auto Mocking
```

We need only one line, so the service will always start http://localhost:8080/tutorial/1.0.0/ui/.

Open the '__main__.py' file and press Run to start the flask server:



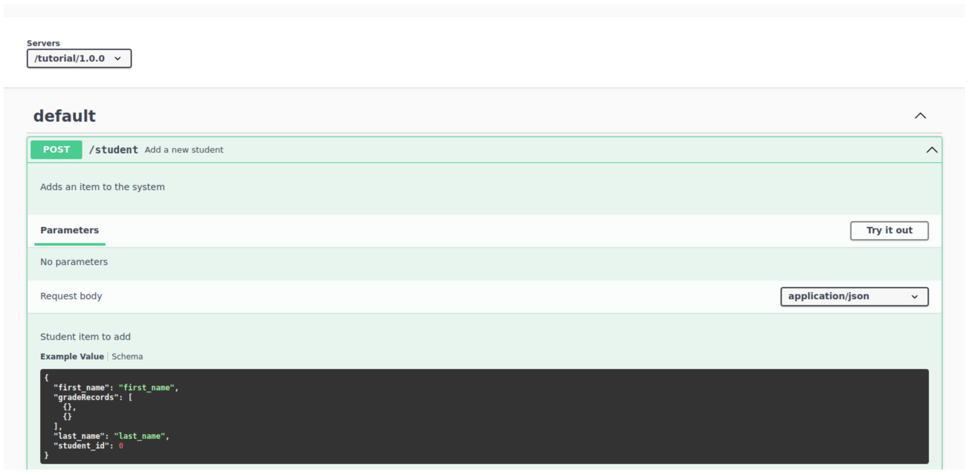
python-flask-server-generated – __main__.py

```
#!/usr/bin/env python3

import connexion

from swagger_server import encoder


def main():
    app = connexion.App(__name__, specification_dir='./swagger/')
    app.app.json_encoder = encoder.JSONEncoder
    app.add_api('swagger.yaml', arguments={'title': 'Simple Inventory API'}, pythonic_params=True)
    app.run(port=8080)


if __name__ == '__main__':
    main()
```

```
Packages installed successfully: Installed packages: 'connexion==2.6.0', 'python_dateutil==2.6.0' (moments ago)
```

The UI API of your service will be in http://localhost:8080/tutorial/1.0.0/ui/ .

On the UI select 'POST' and 'Try it out':



Servers /tutorial/1.0.0

default

POST /student Add a new student

Adds an item to the system

Parameters

No parameters

Request body: application/json

Student item to add

Example Value | Schema

```
{
  "first_name": "first_name",
  "gradeRecords": [
    {},
    {}
  ],
  "last_name": "last_name",
  "student_id": 0
}
```

The response body should be: "do some magic!"
In Pycharm if you open the 'default_controller.py' file, you'll see that the method 'add_student' returns the string "do some magic!".

NOTE

If you make any changes to the code you'll need to restart the flask server.

Create Git Repository and Commit the Code

Create a private git repository.

IMPORTANT

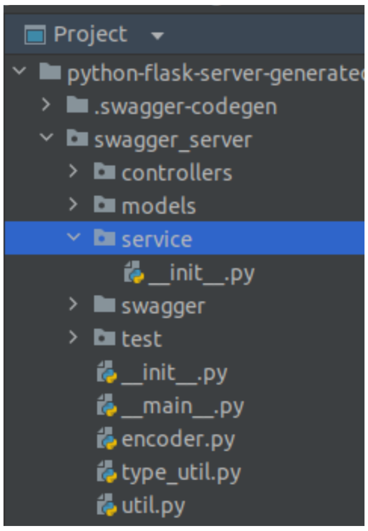
Don't forget to make your repository public from the day of submission and onwards.

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 main
```

Implement the logic

In Pycharm create a package named 'service'. To do that right click on the 'swagger_server' package select 'New'->
'Python Package' and enter the name 'service'



Inside the service package create a new python file named 'student_service'

Inside the service package create a new python file named 'student_service'
In the student_service add the following code: student_service.py

Now you can add the corresponding methods in the 'default_controller.py'. To do that on the top of the 'default_controller.py' add:

```
from swagger_server.service.student_service import *
```

In the 'add_student' method we add the 'add(body)' in the rerun statement, so now the method becomes :

```
def add_student(body=None):  # noqa: E501
    """Add a new student

    Adds an item to the system # noqa: E501

    :param body: Student item to add
    :type body: dict | bytes

    :rtype: float
    """
    if connexion.request.is_json:
        body = Student.from_dict(connexion.request.get_json())  # noqa: E501
        return add(body)
    return 500,'error'
```

Do the same for the rest of the methods. For example, in the ‘delete_student’ method in the 'default_controller.py' file, you need to add ‘delete(student_id)’ in the return statement.

In general, it is a good idea to write an 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'. 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 'default_controller.py' just needs to call the service's methods.

NOTE

Don't forget to regularly commit and push your code.

Build Test and Docker Image

You can now build your web service as a Docker image DockerHub. To do that, open the Dockerfile in the Pycharm project.

and update the python version from:

```
FROM python:3.9-alpine
```

To:

```
FROM python:3.8-alpine
```

So the Dockerfile will look like this: Dockerfile

Open a new terminal in the location of the Dockerfile and type:

```
docker build --tag <REPO_NAME>/student_service .
```

If the above command is not working you may need to use sudo.
Now test the image:

```
docker run -it -p 8080:8080 <REPO_NAME>/student_service
```

NOTE

Don't forget to stop the server from PyCharm otherwise, you'll get an error:

```
shell docker: Error response from daemon: driver failed programming external connectivity on endpoint trusting_joliot (8cbc8523e15eb68f343d048ab
```

MongoDB Integration

The code provided above uses an internal database called TinyDB. Change the code so that your service saves data in a mongoDB.
This includes configuration files for the database endpoint, database names, the Dockerfile itself etc.
For testing your code locally use this file: docker-compose.yaml. Make sure you replace the image with your own.

NOTE

The docker-compose.yaml file above will be also used to run the postman tests.
If you need to install Docker Compose you can follow the instructions here: https://docs.docker.com/compose/install/ .

Write Tests

Before writing the tests in Github you need to create a token in Docker hub. To do that follow these instructions: https://docs.docker.com/docker-hub/access-tokens/
Next you need to add your Docker hub and token to your Github project secrets.
To create secrets follow these instructions https://docs.github.com/en/actions/security-guides/encrypted-secrets#creating-encrypted-secrets-for-a-repository

You need to create two secrets, one named REGISTRY_USERNAME and one REGISTRY_PASSWORD.
Therefore, you need to run the above instructions twice. The first time the name of the secret will be REGISTRY_USERNAME, and the second will be REGISTRY_PASSWORD.

In your code directory create a new folder named 'postman'. In the new 'postman' folder add these files:

- collection.json
- environment.json

Make sure your code in the Git repository is up-to-date. Go to the repository page and create a new file with
'Add file'->'Create new file'. On the top define the path of your file.

```
.github/workflows/main.yml
```

Set the contents of your file as: main.yml

Commit and push your changes to GitHub. After that, any time you commit new code to your repository, your code will be automatically tested, and the Docker container will be built and pushed in DockerHub.

To check the tests, you can go to your Github repository and click on 'Actions'. After the action is completed, the build container should be in your Dockerhub registry.

NOTE

Remember to fill in the REPO_NAME in main.yml, which should be your docker hub **username**, NOT your docker hub
repository name.

The REGISTRY_USERNAME is your **username** for docker hub, NOT your docker hub repository name.

Deploy Web Service on Kubernetes (MicroK8s)

Install MicroK8s

You can find MicroK8s installation instructions: https://MicroK8s.io/

If you have access to a cloud VM you may install MicroK8s there. Alternatively, you may also use
VirtualBox: https://www.virtualbox.org/wiki/Downloads

After you complete the installation, make sure you start MicroK8s and enable DNS

```
microk8s start
microk8s enable dns
```

NOTE

In Linux, you may need to use these commands with sudo

If you get an error:

```
sudo: microk8s: command not found
```

/snap/bin is probably not in your path. Either use: \

```
sudo /snap/bin/microk8s start
```

or add /snap/bin to your PATH.

Test K8s Cluster

This is a basic Kubernetes deployment of Nginx. On the master node, create a Nginx deployment:

```
microk8s kubectl create deployment nginx --image=nginx
```

You may check your Nginx deployment by typing:

```
microk8s kubectl get all
```

The output should look like this:

```
NAME                         READY   STATUS              RESTARTS   AGE
pod/nginx-748c667d99-zdxh6   0/1     ContainerCreating   0          13s
```

```
NAME                 TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.152.183.1   <none>        443/TCP   2m54s

NAME                    READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nginx   0/1     1            0           35s

NAME                               DESIRED   CURRENT   READY   AGE
replicaset.apps/nginx-748c667d99   1         1         0       13s
```

You will notice in the first line 'ContainerCreating'. This means that the K8s cluster is downloading and starting the Nginx container. After some minutes, if you run again:

```
microk8s kubectl get all
```

The output should look like this:

```
NAME                         READY   STATUS    RESTARTS   AGE
pod/nginx-748c667d99-zdxh6   1/1     Running   0          43s

NAME                 TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.152.183.1   <none>        443/TCP   3m24s

NAME                    READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nginx   1/1     1            1           65s

NAME                               DESIRED   CURRENT   READY   AGE
replicaset.apps/nginx-748c667d99   1         1         1       43s
```

At this point, Nginx is running on the K8s cluster, however it is only accessible from within the cluster. To expose Nginx to the outside world, we should type:

```
microk8s kubectl create service nodeport nginx --tcp=80:80
```

To check your Nginx deployment type:

```
microk8s kubectl get all
```

You should see the among others the line:

```
service/nginx        NodePort    10.152.183.82   <none>        80:31119/TCP   9s
```

This means that port 80 is mapped on port 31119 of each node in the K8s cluster.

NOTE

The mapped port will be different on your deployment. Now we can access Nginx from http://IP:NODE_PORT .

You may now delete the Nginx service by using:

```
microk8s kubectl delete service/nginx
```

Deploy Web Service on K8s Cluster

To deploy a RESTful Web Service on the K8s Cluster create a folder named 'service' and add this file in the folder:
student_service.yaml

Open 'student_service.yaml' and replace the line:

```
image: IMAGE_NAME
```

with the name of your image as typed in the docker push command.

If you choose to integrate with an extremal database you will need to add the Deployment and service for MongoDB:
mongodb.yaml

To create all the deployments and services, type in the K8s folder:

```
microk8s kubectl apply -f .
```

This should create the my-temp-service deployments and services. To see what is running on the cluster type:

```
microk8s kubectl get all
```

You should see something like this:

```
NAME                           READY   STATUS    RESTARTS   AGE
pod/nginx-6799fc88d8-q7hzv     1/1     Running   0          21m
pod/service-6c75dff7db-57sw5   1/1     Running   0          53s

NAME                 TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
service/kubernetes   ClusterIP   10.152.183.1     <none>        443/TCP          26m
service/service      NodePort    10.152.183.176   <none>        8080:30726/TCP   53s

NAME                      READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nginx     1/1     1            1           21m
deployment.apps/service   1/1     1            1           53s

NAME                                 DESIRED   CURRENT   READY   AGE
replicaset.apps/nginx-6799fc88d8     1         1         1       21m
replicaset.apps/service-6c75dff7db   1         1         1       53s
```

Note that in this output, 'service/service' is mapped to 30726. In your case, it may be a different number.

Now your service should be available on http://IP:NODE_PORT/