

% RESTful services, Docker and Kubernetes

## Introduction

In this tutorial we will use OpenAPI to define a RESTful web service and Python to implement it.

The RESTful web service will be using 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 and Docker Image](#)
5. [Write Tests](#)
6. [Deploy Web Service on Kubernetes \(MicroK8s\)](#)

## Reporting and assessment

### Reporting

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

- Each student should write a short report (max 4 pages) with the following structure:
  - **Introduction:** List which of the DevOps stages you practiced with this assignment and what are their primary objectives.
  - **OpenAPI Exercises:** Report on the following exercises (for details see on the Section [OpenAPI Exercises](#)):
    - [Define Objects](#)
    - [Add Delete method](#)
  - **MongoDB Integration (Optional):** If you choose to integrate an external database report on how the modifications you had to make on your code and Dockerfile to connect to DB.
  - **Question 1:** In the Docker file we use the 'python:3.9-alpine' image as a base image. What are the advantage and disadvantages of using the Alpine Linux distribution in Docker.

### Assessment

---

#### IMPORTANT

In order to be given a grade you must submit the following:

- Written report (see [Reporting](#) for details)
- Name of the published docker(s) in <https://hub.docker.com/> . Must be able to perform (docker pull <REPO/NAME>)
- Git repository link
- If you choose the optional exercise task i.e. the MongoDB integration submit **only** the code for connecting to MongoDB.
- If you choose the optional exercise task i.e. the MongoDB integration the testing will be done using the '[docker-compose.yaml](#)'

**Do not add your code in Canvas or in your report.**

**All links such as DockerHub and Git must be accessible from the day of submission and onwards.**

---

- If the source code and dockerized service pass all the tests defined in newman you get 70%.
- A passing mark will be given if the source code and dockerized service pass tests 1-5 in newman (see Section [Build Test and Docker Image](#)).
- The tests will run on the docker created from source i.e. from building the docker file from source and the docker image from your DockerHub account.
- An extra point will be given for the [MongoDB Integration \(Optional\)](#).
- The rest will be determined by your report.

## 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 <https://www.youtube.com/watch?v=pRS9LRBgjYg>

### 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 <https://www.youtube.com/watch?v=wpISo9TNjIU>

### 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 with 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 <https://www.youtube.com/watch?v=0qotVMX-J5s>

### 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 <https://www.youtube.com/watch?v=kBF6Bvth0zw>

# Prepare your Development Environment

## Create GitHub Account

If you don't have an account already follow these instructions: <https://github.com/join>

## Setup Docker Hub

If you don't have an account already follow these instructions: <https://hub.docker.com/signup>

## SwaggerHub Account

If you have 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:

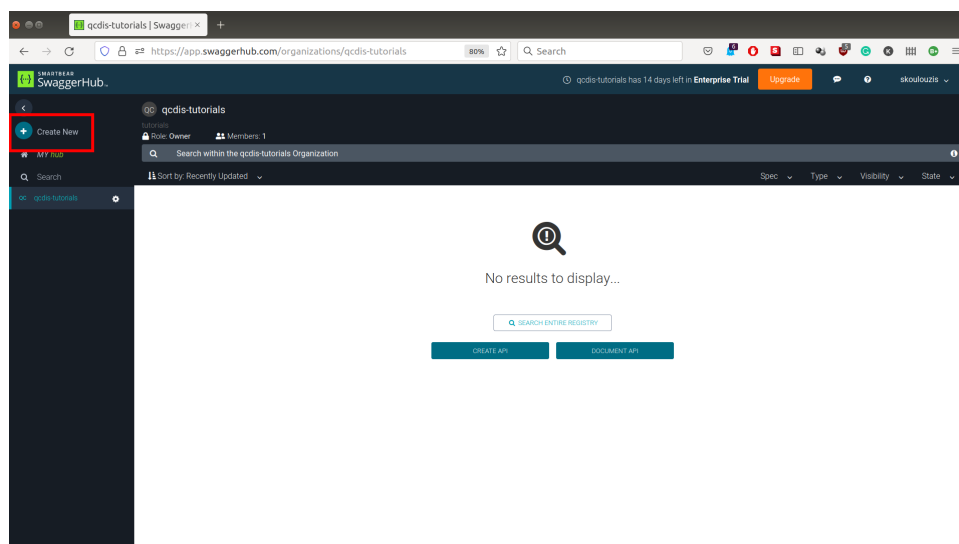
```
sudo snap install pycharm-community --classic
```

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

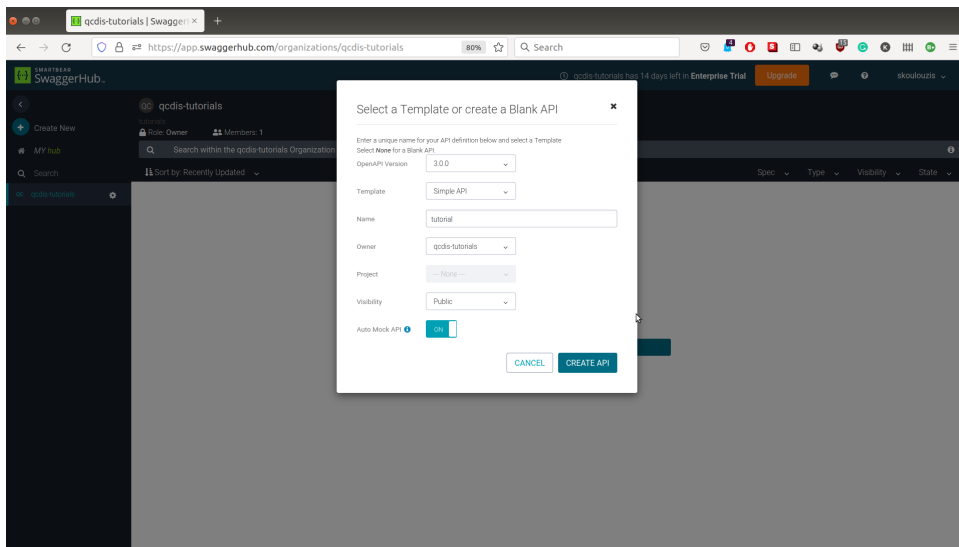
# Write OpenAPI Definition

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](https://idratherbewriting.com/learnapidoc/openapi_tutorial.html)

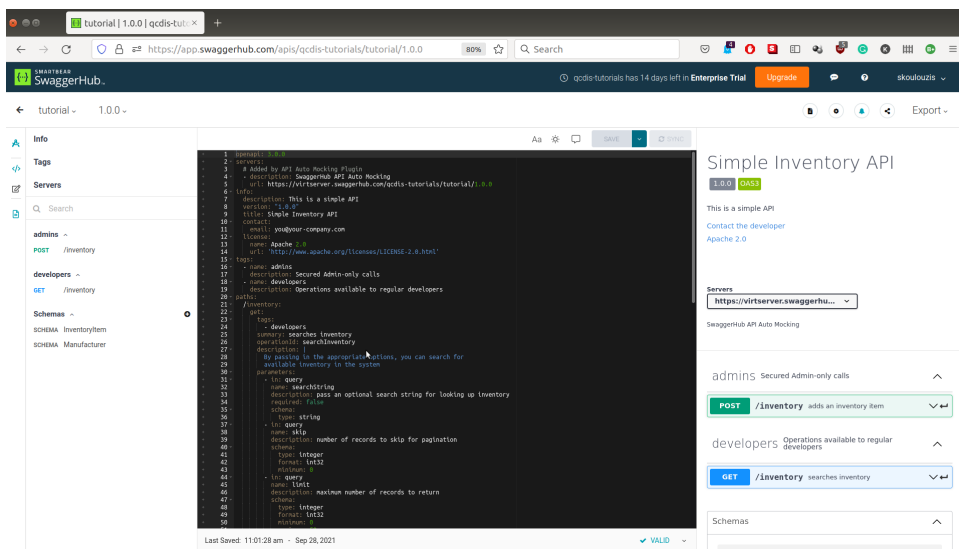
Log in to your SwaggerHub account at <https://app.SwaggerHub.com/login> and select 'Create New' -> 'Create New API'



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



You will get a OpenAPI template



Name your API 'tutorial'.

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
gradeRecords	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 useful to add 'example' fields in the properties. That way you 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

The API definition at the moment 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":
```

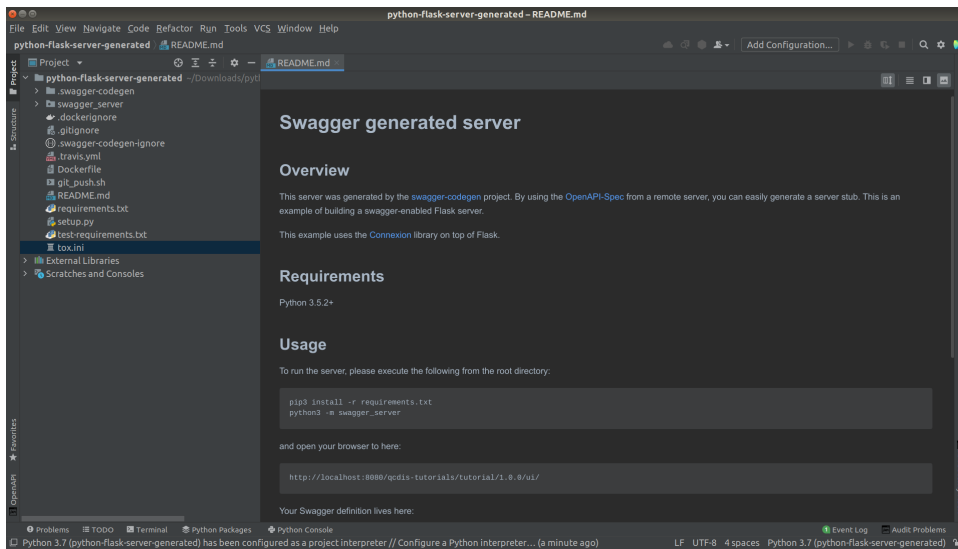
You will need to fill in the proper responses for 200, 400, and 404. More information about responses can be found here: <https://swagger.io/docs/specification/describing-responses/>

## Generate Python Code

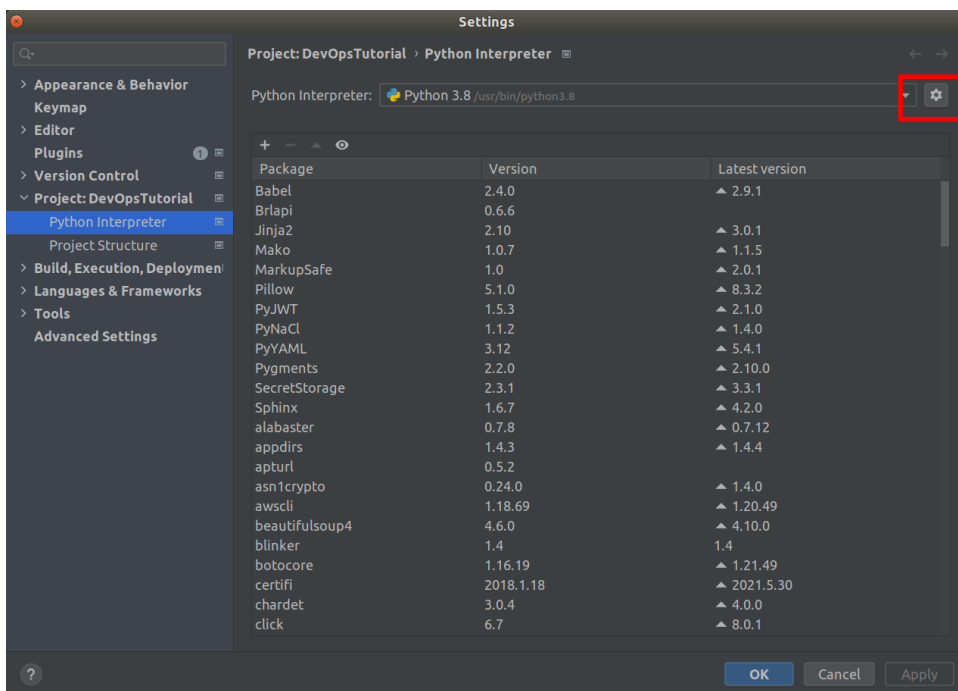
Now that we have the OpenAPI definitions we can create the server stub on python. Select 'Export'-'>'Server Stub'->'python-flask'

The screenshot shows the SwaggerHub web interface. On the left, the 'Info' tab is active, displaying the API definition for 'tutorial' (version 1.0.0). The definition includes a 'delete' method for the '/student/{student\_id}' endpoint. The 'Responses' section shows three responses: 200 (search results), 400 (bad input parameter), and 404 (no links). On the right, the 'Export' menu is open, and the 'Server Stub' option is selected. Below the menu, the 'python-flask' option is also visible. The bottom of the interface shows a 'Valid Definition' status and a 'Schemas' section with a 'MeasurementItem' schema.

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



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:



Select 'New environment' and press 'OK'

Replace the 'requirements.txt' file with this [requirements.txt](#)

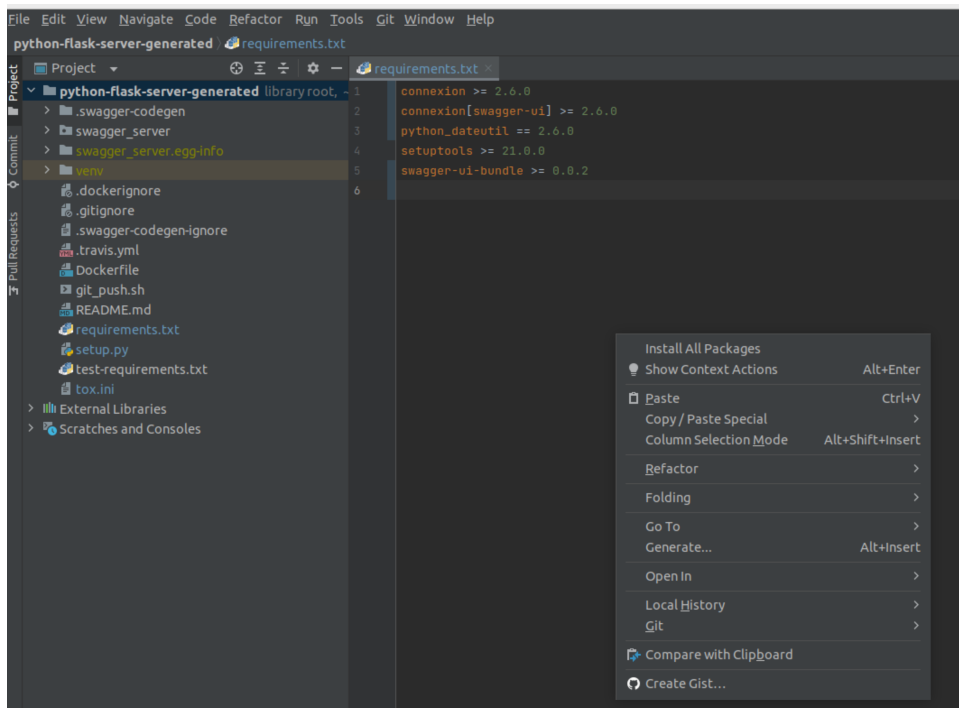
Open the 'requirements.txt' file and right click and select install all packages.

## NOTE

If the installation of the requirements fails use from command line:

```
pip install -r requirements.txt
```

This may happen because Pycharm sometimes uses conda channel to search for packages.



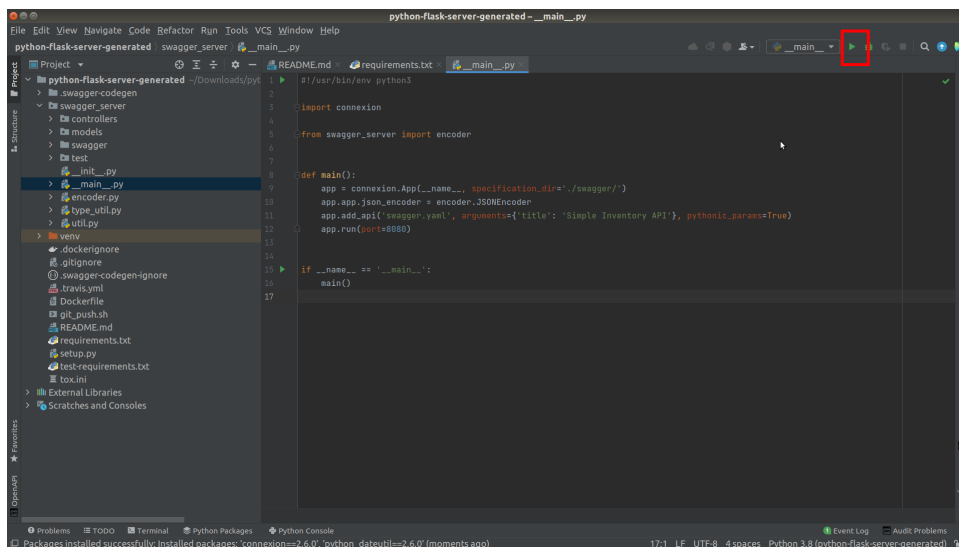
Open the file 'swagger\_server/swagger/swagger.yaml' and in the section 'servers' you should have only one url and decryption. The servers section should look like this:

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:



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':



In Pycharm if you open the 'default\_controller.py' file, you'll see that the method 'add\_student' returns the string "do some magic!".

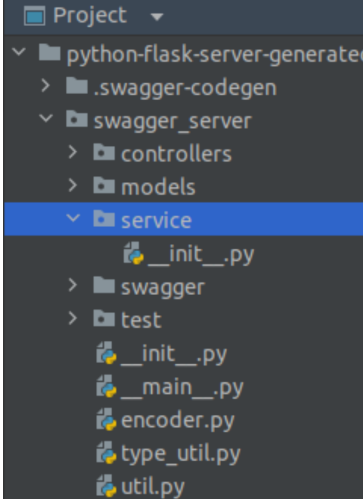
If you make any changes to the code you'll need to restart the flask server.

## IMPORTANT

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 <REPOSITORY_URL>
git push -u origin main
```

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'. In the student service add the following code: [student\\_service.py](#)

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

```
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 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 (Optional)

The code provided above uses an internal database called TinyDB. Change the code so that your service saves data in an 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 during grading.

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. First time the name of the secret will be REGISTRY\_USERNAME and 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 and page 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 build and pushed in DockerHub.



To check the tests go to your Github repository and click on 'Actions'. After the action is successfully 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 chose 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/`