Task

Your task consists of two different steps. First, you have to create a Flask application that serves basic **CRUD** (*Create, Read, Update, Delete*) operations via HTTP requests. You have to use MongoDB, a popular NoSQL database, to persist data. Second, you have to make a Docker image of the Flask application and start two services (i.e., the application and the database) with Docker Compose.

Tip

You should first get familiar with Flask and Docker Compose by following the related tutorials in MyCourses.

The application listens for HTTP requests, processes them, and provides the corresponding outcome in **JSON** format. Error handling can be done by following standard practices. This page summarizes the different types of HTTP requests and status codes.

Attention

Make sure the database name is set to **'flask-database'** and the database schema, collections and routes are defined according to the instructions below.

Schemas

`Photo`, to be used in the **'photo'** collection:

Attribute	Type	Conditions	Values
name	String	required:true	Any valid string
tags	List	–	[Any valid strings]
location	String	–	Any valid string
image_file	Image	required:true	Any valid image file
albums	List of `Album`	–	[albumIds]

`Album`, to be used in the **'album'** collection:

Attribute	Type	Conditions	Values
name	String	required:true unique: true	Any valid string
description	String	–	Any valid string

Operations

Note

All GET requests with a body should use the query string to pass the parameters.

`Photo`:

Request	Type	Route	Request Body	Response Body	Response Status Code
Create	POST	/listPhoto	Photo (**)	{ message: ‘Photo successfully created’, id: `photo_id` }	201
Read	GET	/listPhoto/{photo_id}	–	Single database object { name: `name`, tags: [`tags`], location: `location`, albums: [`albums`], file: `image_file` }	200
Read	GET	/listPhotos	tag (*)	Multiple database objects [{name: `name`, location: `location`, file: `image_file`}]	200
Read	GET	/listPhotos	albumName (*)	Multiple database objects [{name: `name`, location: `location`, file: `image_file`}]	200
Update	PUT	/listPhoto/{photo_id}	Photo (~)	{ message: ‘Photo successfully updated’, id: `photo_id`}	200
Delete	DELETE	/listPhoto/{photo_id}	–	{ message: ‘Photo successfully deleted’, id: `photo_id` }	200

(*) This is a query string as in the note above.

(**) The request body is a multi-part form, i.e., `file` (the image) as `request.files` and the associated json data as `request.form`

(~) `image_file` is not sent as part of request body. The remaining `Photo` is sent as json.

All the images by default are placed in an `Album` with the name `Default`. When any `Photo` entry is created the POST method should check if the default album exists and create it if it does not. Each image belongs to the default Album as well as to any additional albums to which it is added to. The albumName in GET /listPhotos is optional, thus, it could be empty. In the latter case, the default `Album` should be used.

The `image_file` returned in the three GET requests discussed in the above table should be encoded using `base64` encoding and then decoded using `utf-8`. This has been illustrated in the scaffolding code provided at the end of requirements section. No encoding/decoding is required while saving to the database.

Note

All `/listAlbum` requests use JSON in their request body.

`Album`:

Request	Type	Route	Request Body	Response Body	Response Status Code
Create	POST	/listAlbum	Album	{ message: ‘Album successfully created’, id: `album_id` }	201

Request	Type	Route	Request Body	Response Body	Response Status Code
Read	GET	/listAlbum/{album_id}	–	Single database object { id: `album_id`, name: `name` }	200
Update	PUT	/listAlbum/{album_id}	Album	{ message: ‘Album successfully updated’, id: `album_id` }	200
Delete	DELETE	/listAlbum/{album_id}	–	{ message: ‘Album successfully deleted’, id: `album_id` }	200

Note

You can run the following command on your machine to persist the requests on MongoDB:

```
docker run -d -p 27017:27017 -v ~/data:/data/db mongo
```

Requirements

To dockerize the Flask application, you have to write a Dockerfile in the project directory that does the following:

- Use `python:3.10.8-alpine3.16` as the base image
- Set the working directory to `/usr/app`
- Install the following alpine packages using alpine package manager:
    - gcc, libc-dev, linux-headers, zlib-dev, jpeg-dev, libjpeg
- Copy the `requirements.txt` files in the route `./`
- Install the dependencies (using `pip install`)
- Copy the app source code
- Expose the port 5000
- Set the command to run `flask run --host=0.0.0.0`

To start the two services, you must write a *docker-compose* file that creates:

- A service called `backend` with the following characteristics:
    - The name of the container is: `flaskbackend`
    - The image must be the dockerized flask app with the name `flaskbackend:v1`
    - Binds the port 5000:5000
    - Depends on the second service (mongo)
- A service called `mongo` with the following characteristics:
    - The name of the container is: `mongo`
    - The image must correspond with the version 6.0.2 of MongoDB (`mongo:6.0.2`)
    - Bind the port 1048:27017

Danger

This assignment does not use database authentication only for simplicity and easier integration with the grader. In a real scenario, authentication **must** be configured for the database, which should **not** be exposed to the Internet. See this resource on container network security and this other resource on passing secrets to containers for detailed security-related considerations.

Hint

To get started you can download a scaffolding Flask application **`here`**. The content of the archive provides the basic structure of the Flask application on top of which the remaining features can be added.

Grading

You must submit two files.

A **ZIP archive** containing the full implementation of the Flask application and the corresponding Dockerfile. Your ZIP file must have the following structure:

```
./exercise.zip
└── exercise
    ├── Dockerfile
    ├── requirements.txt
    ├── app.py
    ├── database
    ...
```

A `docker-compose.yaml` that starts both services: the backend application and the MongoDB database server.

Note

The assignment runs multiple unit tests which give fractional points based on how the requirements in task are fulfilled, according to the table below.

Test	Points
Required files and folders exist in the container	10
Port, work directory and commands are correct	10
The Flask container has the correct name	12
The database container has the correct name	12
The Flask container has the correct port	12
The database container has the correct port	12
Rest API tests (11 different tests, worth 8 points each)	88