GIS – Tutorial 2 – Backend

University of Konstanz | Due Date: 18.11.2022 08:00

1 Submission Instructions

Please provide **one** well-formatted **PDF** file with all your submissions on schedule (i.e. before the deadline). Otherwise your submission will not be graded!

2 Background

Python is an interpreted, high-level and general-purpose programming language. Python is dynamically typed and garbage-collected. It supports multiple programming paradigms, including structured (particularly, procedural), object-oriented, and functional programming.
For more information visit: https://www.python.org/
For a list of tutorials visit: https://hackr.io/tutorials/learn-python/

Flask is a lightweight WSGI web application framework. It is designed to make getting started quick and easy, with the ability to scale up to complex applications.
For more information visit: https://flask.palletsprojects.com/

2.1 Installation

This tutorial assumes that PostgreSQL and the PostGIS extension will be installed as a Docker container. If you want to manually install both of these programs please visit the following URLs:

- Python: https://www.python.org/downloads/
- Flask: https://flask.palletsprojects.com/en/1.1.x/installation/

2.2 Useful commands for this assignment

During this tutorial we will often start & stop Docker containers, the following commands will be often used or might be helpful:

- *docker run [OPTIONS] IMAGE*
  https://docs.docker.com/engine/reference/run/
  Runs the *IMAGE* Docker container with the specified *[OPTIONS]*
- *docker ps*
  https://docs.docker.com/engine/reference/commandline/ps/
  Shows a list of currently running Docker containers
- *docker stop [OPTIONS] CONTAINER*
  https://docs.docker.com/engine/reference/run/
  Stops the *[CONTAINER]*. You can identify the container id or name using *docker ps*.
- *docker system prune -a*
  https://docs.docker.com/engine/reference/commandline/system_prune/ Remove all unused containers, networks, images, etc.

Later in the tutorial we will also use docker-compose, here the following commands are the most used:

- *docker-compose up*
  https://docs.docker.com/compose/reference/up/
  Builds, (re)creates, starts, and attaches to containers for a service.
- *docker-compose down*
  https://docs.docker.com/compose/reference/down/
  Stops containers and removes containers, networks, volumes, and images created by up.

3 Homework

Task 1: Python Setup **0 Points**

1. Download the following Docker container that runs Python

   docker pull python:3.9-slim

2. Ensure you can successfully run this Docker container by running the following command:

   docker run -it --name python python:3.9-slim bash

   This will run bash inside the downloaded Python Docker container, you can check that the correct version of Python installed is using the command:

   python --version

   which should return *Python 3.9.x*. You can then exit the Docker container using the *exit* command.

Task 2: Modifying the Docker container **4 Points**

1. Save the code below in a file named "Dockerfile":

```
# specify base image that we want to extend
FROM python:3.9-slim

# app target location
ENV APP_DIR=/var/opt/gis
RUN mkdir -p ${APP_DIR}
WORKDIR ${APP_DIR}

# install build-essentials
RUN apt-get update
RUN apt-get install -y build-essential

# install dependencies
RUN pip3 install --upgrade pip
RUN pip3 install Flask Flask-Cors
RUN pip3 install psycopg2-binary

# environment variables
ENV FLASK_APP=server.py
ENV FLASK_DEBUG=1

# run flask server
CMD [ "python", "-m", "flask", "run", "--host=0.0.0.0" ]
```

2. Check that you can successfully run this modified Docker container by running the two following commands:

   (a) Build the new Docker container and give it a name (tag):
   docker build - < Dockerfile -t my_own_python_container

   (b) Run the new Docker container:
   docker run -it my_own_python_container bash

   (c) Run the command *python* and check that you can execute the following code without any errors:
   from flask import Flask
   Hint: To exit Python either type exit() or use the shortcut Ctrl+D.
   Fun Fact: Check what the result of *1.2 - 1* is in Python.

Submission: State the tag, image id and size of your modified Python container. You can check this information with the command *docker images*.

Task 3: Coding inside your Docker container? **4 Points**

After we've successfully created our modified Python Docker container we of course want to run some Python code in it. To do this we could, for instance, create a Python script, use the Docker command *COPY* in our Dockerfile to add it to the container and modify the *CMD* instruction to run this Python script. However, this would require us to rebuild the container after every change to the script, which would, of course, be quite tedious. It would be easier to map a file that we're currently working on our host system into the Docker container.

1. Create a folder GIS containing a subfolder "backend" and add the previous Dockerfile to it.
2. Create a folder "code" in your "backend" folder.
3. Create a file called server.py in the directory "backend/code" and add code block to it.

```
from flask import Flask, escape, request

app = Flask(__name__)

@app.route('/')
def hello():
    name = request.args.get("name", "World")
    return f'Hello, {escape(name)}!'
```

4. Create a file named "docker-compose.yml" in the GIS folder and add the following code-block to the "docker-compose.yml" file.
   Indentation is important here!

```
version: "3"
services:
    backend:
        build: ./backend
        ports:
            - "5000:5000"
        volumes:
            - ./backend/code:/var/opt/gis
```

Afterwards your directory structure should look like this:

```
GIS
├── backend
│   ├── code
│   │   └── server.py
│   └── Dockerfile
└── docker-compose.yml
```

5. Run the following command: *docker-compose up*
6. Visit *localhost:5000*, you should see the text “Hello, World!”
7. Visit *localhost:5000?name=<something>*, you should see the text “Hello, <something>”
8. Add a new function “add” in the server.py file using the template below that takes two numbers from the url and returns their sum. The default value if the number is not specified should be 0.

```
@app.route('/add')
def add():
    # do something here to retrieve both numbers from the url
    # check above on how to get default values with get()
    number1 = ...
    number2 = ...

    return f"{number1 + number2}"
```

Hint: You can specify multiple parameters with & in between them

Submission: Your add() function.

Task 4: Querying the database, again! **8 Points**

Finally we will combine our database from the last assignment with our backend from this assignment. You should have a separate Dockerfile for each.

1. Create subfolder "database" in the GIS folder and add the database Dockerfile from the last tutorial. Afterwards your directory structure should look like this:

```
GIS
├── backend
│   ├── code
│   │   └── server.py
│   └── Dockerfile
├── database
│   └── Dockerfile
└── docker-compose.yml
```

2. Update the "docker-compose.yml" file in the GIS folder. **Identation is important here!**

```
version: "3"

services:
    database:
        build: ./database
        ports:
            - "25432:5432"

    backend:
    build: ./backend
    ports:
        - "5000:5000"
    volumes:
        - ./backend/code:/var/opt/gis
```

3. Run the command *docker-compose up* in your GIS folder. This will start both the backend, as well as the database simultaneously and also enables containers to communicate with each other!

4. Finally, we again want to query the database. This time, however, we want to write a function, which either returns the number of all objects in the table *planet_osm_polygon* were the amenity is not null or can take an amenity type and returns the number of objects with this amenity type in the database. For this, use the code block below as a template.

```
@app.route('/how_many')
def howmany():
    # Don't change the parameters database and port.
    # You just have to change the dbname, user and password!
    connection = psycopg2.connect(host="database", port=5432,
                                  dbname=<name>, user=<user>,
                                  password=<password>)

    # TODO: if present, get amenity type from URL

    # TODO: write your query to get the amenities count here
    query = "YOUR QUERY HERE!"

    # Your query should return only one row
    # Thus fetchone() should be enough
    with connection.cursor() as cursor:
        cursor.execute(query)
        result = cursor.fetchone()

    #TODO: parse the data in the result variable and return it

    return f"{howmany}"
```

Note: Your query should work directly on `localhost:5000/how_many` and also parameterized, e.g. `localhost:5000/how_many?type=cafe`

Hint: Check `https://www.psycopg.org/docs/usage.html` to see how to use the psycopg2 library, how to pass parameters to the queries and how to parse the results from fetchone(), fetchmany(), etc.

Submission: The code for task 4.