Practical: REST API with Multi-Container Service Architecture (Python)

In this practical, you will be hands on with a basic REST API implemented with Python and MariaDb

Your task is to review the code, containerise the application and deploy it on EC2.

Importantly, this application uses two containers: one for the Python process, another for the MariaDb database.

The application is a basic task manager. It does not implement any CPU-intensive task but a starting point for a REST API and a service architecture having multiple containers.

Table of Contents

Step 1: Understand the context

This is a task management app that has a REST API with five endpoints:

1. Get all tasks
2. Get a particular task
3. Create a task
4. Update a task
5. Delete a task

There is no separation of tasks by user, and no security, so anyone could be interacting with this API and be editing/deleting other peoples tasks. Obviously this does not make sense in the real-world but this prac is not about demonstrating separation of users.

We have provided a simple API to get started, just as a demonstration of how an Python FastAPI can be implemented and how to orchestrate "one application" that has multiple applications.

Database access with Python

Database methods to create, find, update, or delete records are asynchronous. This means that the database querying methods return immediately, and the code to handle the success or failure of them run at a later time when the query completes. Other code can execute while the server is

waiting for the database operation to complete, so the server can remain responsive to other requests.

Python has a number of mechanisms for supporting asynchronous behaviour. Python's standard behaviour is synchronous, meaning that code runs sequentially, one line after another. However, Python has support for asynchronous programming through the `asyncio` library and the `async` and `await` keywords. This is similar to how asynchronous programming is handled in JavaScript, where the `async` keyword is used to define an asynchronous function and the `await` keyword is used to pause execution until a Promise is resolved.

Step 2: Implement the Task Manager API

You should follow along with our instructions. We have not provided a ZIP of the code because you need to get hands-on with setting-up a project like this one. It is relatively small, to keep the efforts low. The important part is you understanding how the app is structured and what the code does. Remember, CAB432 is about cloud infrastructure, not software development (i.e., learning a programming language or building small apps). We need to focus on *how to design and deploy apps that are scalable*.

We are going to create a new python project from scratch. There are different ways to structure the python application. Focusing on the previous comment and examples to follow (like the JS version of this prac), we will use a layered architecture. This means that we will have a separation of concerns between the routes, controllers, and models.

```
mkdir -p {db,tasksapi}
touch tasksapi/routes.py
touch tasksapi/controllers.py
touch tasksapi/models.py
touch db/db.py
touch app.py
touch requirements.txt
```

The main entry-point of the app is `app.py`:

```
from fastapi import FastAPI
from tasksapi.routes import router as api_router

app = FastAPI(
    title="Task Management API",
    description="API for managing tasks",
    version="0.0.1",
)

# Prefix is used to group routes under a common path
app.include_router(api_router, prefix="/tasks")

if __name__ == "__main__":
```

```
  import uvicorn
  uvicorn.run(app, host="0.0.0.0", port=3000)
```

We also want to initialise the requirements file for the Python app. This file will list all the dependencies that the app needs to run. Inside the `requirements.txt` file, we will add the following dependencies:

```
fastapi
mariadb
uvicorn
```

There is also an dependency in ubuntu that is needed to run mariadb, so we will need to install that as well. Later we'll install this dependency on the Docker image via the Dockerfile.

Here is `db/db.py` which establishes the connection to the MariaDb database. Have a look at the code and try to understand how it works. Try to see what we are doing with `os.environ.get()` for example - what does the first field and the second field mean?

```
import os
import mariadb

DB_HOST = os.environ.get("DB_HOST", "mariadb")
DB_USER = os.environ.get("DB_USER", "user")
DB_PASSWORD = os.environ.get("DB_PASSWORD", "pass")
DB_NAME = os.environ.get("DB_NAME", "tasksdb")
DB_PORT = int(os.environ.get("DB_PORT", 3306))

def get_connection():
  try:
    conn = mariadb.connect(
        host=DB_HOST,
        user=DB_USER,
        password=DB_PASSWORD,
        database=DB_NAME,
        port=DB_PORT
    )
  except mariadb.Error as e:
    print(f"Error connecting to MariaDB Platform: {e}")
    sys.exit(1)
  return conn

# Initialize the database and create the tasks table if it doesn't ex
ist
def init_db():
  conn = get_connection()
  try:
    cursor = conn.cursor()
    cursor.execute("""
      CREATE TABLE IF NOT EXISTS tasks (
        id INT PRIMARY KEY AUTO_INCREMENT,
        title VARCHAR(255) NOT NULL,
```

```
        completed TINYINT DEFAULT 0
      );
    """)
    conn.commit()
    print("Database initialized and tasks table ensured.")
  except Exception as e:
    print(f"DB init failed: {e}")
  finally:
    conn.close()

init_db()
```

The next file we will implement is `tasksapi/models.py`, which is a model to represent tasks. It is responsible for the main interactions with the database.

```
from db.db import get_connection

def get_all():
  conn = get_connection()
  cursor = conn.cursor()
  cursor.execute("SELECT * FROM tasks")
  rows = cursor.fetchall()
  columns = [desc[0] for desc in cursor.description]
  conn.close()
  return [dict(zip(columns, row)) for row in rows]

def get_by_id(task_id):
  conn = get_connection()
  cursor = conn.cursor()
  cursor.execute("SELECT * FROM tasks WHERE id = ?", (task_id,))
  row = cursor.fetchone()
  columns = [desc[0] for desc in cursor.description]
  conn.close()
  return dict(zip(columns, row)) if row else None

def create(title):
  conn = get_connection()
  cursor = conn.cursor()
  cursor.execute("INSERT INTO tasks (title, completed) VALUES (?,
?)", (title, 0))
  conn.commit()
  task_id = cursor.lastrowid
  conn.close()
  return {"id": task_id, "title": title, "completed": 0}

def update(task_id, title, completed):
  conn = get_connection()
  cursor = conn.cursor()
  cursor.execute(
    "UPDATE tasks SET title = ?, completed = ? WHERE id = ?",
    (title, 1 if completed else 0, task_id)
  )
  conn.commit()
```

```
  updated = cursor.rowcount > 0
  conn.close()
  return {"updated": updated}

def remove(task_id):
  conn = get_connection()
  cursor = conn.cursor()
  cursor.execute("DELETE FROM tasks WHERE id = ?", (task_id,))
  conn.commit()
  deleted = cursor.rowcount > 0
  conn.close()
  return {"deleted": deleted}
```

And a controller, `tasksapi/controllers.py` that implements the logic around the model. Notice that this controller handles the incoming request (`request`) and parses the outgoing response in json format (`response`), and has no direct interaction with the database -- the data model does that. This layered approach is useful for if/when the model is changed (we wouldn't need to update any controllers that interact with the Task model) or when the underlying database engine changes (we only update the models' interaction with the database, not any controllers). The controller also performs some input validation because it is capable of returning error messages to the requester.

```
from fastapi import APIRouter, HTTPException, status, Request
from fastapi.responses import JSONResponse
from tasksapi.models import *

router = APIRouter()


def get_all_tasks():
  try:
    return JSONResponse(content=get_all())
  except Exception as e:
    raise HTTPException(status_code=500, detail=str(e))


def get_task_by_id(task_id: int):
  try:
    result = get_by_id(task_id)
    if not result:
      raise HTTPException(status_code=404, detail="Task not found")
    return JSONResponse(content=result)
  except Exception as e:
    raise HTTPException(status_code=500, detail=str(e))


async def create_task(request: Request):
  data = await request.json()
  title = data.get("title")
  if not title:
    raise HTTPException(status_code=400, detail="Title is required")
  try:
```

```
    return create(title)
  except Exception as e:
    raise HTTPException(status_code=500, detail=str(e))


async def update_task(task_id: int, request: Request):
  data = await request.json()
  title = data.get("title")
  completed = data.get("completed", False)
  try:
    result = update(task_id, title, completed)
    if not result.get("updated"):
      raise HTTPException(status_code=404, detail="Task not found")
    return {"message": "Task updated"}
  except Exception as e:
    raise HTTPException(status_code=500, detail=str(e))

def delete_task(task_id: int):
  try:
    result = remove(task_id)
    if not result.get("deleted"):
      raise HTTPException(status_code=404, detail="Task not found")
    return {"message": "Task deleted"}
  except Exception as e:
    raise HTTPException(status_code=500, detail=str(e))
```

Now that we have a model and controller, we need to connect these to the interface of the Task Manager app. Our UI is going to be a REST HTTP API, therefore, we need to tell the Python app what are our endpoints/routes (i.e., HTTP URLs) for reaching the controllers.

For instance, we may send a request to update Task #123 with the following HTTP request: `PUT https://cab432.com/tasks/123`. Its payload (i.e., HTTP body) may be something like `{ title: "Task Renamed", completed: 1 }`. Here is `tasksapi/routes.py`, which defines the routes for the Task Manager API:

```
from fastapi import APIRouter
from tasksapi.controllers import *

router = APIRouter()

router.get("/", response_model=list)(get_all_tasks)
router.get("/{task_id}")(get_task_by_id)
router.post("/")(create_task)
router.put("/{task_id}")(update_task)
router.delete("/{task_id}")(delete_task)
```

We have now provided code for all the necessary files. However, you won't yet be able to run this app because there is no database available. The next step is to create relevant Docker containers: one for the Node app and one that is running MariaDB.

Step 3: Deploy with Docker

This section is about setting up the app to run with Docker. At the bottom of this prac, in the appendix, are some extra Docker commands that you might find useful along the way (in case things go wrong or to help with debugging). The appendix also includes a command to launch a temporary container that is running a web-based database explorer, so you can manually create/update/delete records in the MariaDB.

Let's start by creating a container for the Node app.

Create a Dockerfile at the root of the project (i.e., create `Dockerfile` in the same directory as `app.py`):

```
FROM python:3.11-slim

RUN apt-get update \
    && apt-get -yy install libmariadb3 libmariadb-dev gcc

WORKDIR /usr/src/app

COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["python3", "-m", "uvicorn", "app:app", "--host=0.0.0.0", "--port
=3000"]
```

Build that container

```
docker build -t task-api .
```

Create a network (read docs **here (https://docs.docker.com/engine/network/)**) so the containers can communicate

```
docker network create tasknet
```

Run MariaDB as a container

```
docker run -d --name mariadb --network tasknet -e MARIADB_ROOT_PASSWO
RD=rootpass -e MARIADB_DATABASE=tasksdb -e MARIADB_USER=user -e MARIA
DB_PASSWORD=pass -p 3306:3306 -v mariadb-data:/var/lib/mysql mariadb:
11
```

- `docker run -d`: Runs the container in **detached mode** (background).
- `--name sqlite`: Names the container `sqlite`.
- `--network tasknet`: Connects the container to the `tasknet` Docker network.

- **Environment variable (`-e`)**:
  - `SQLITE_DATABASE=tasks.db`: Creates a new SQLite database file named `tasks.db`.
- `-v sqlite-data:/data`: Mounts a **named volume** (`sqlite-data`) to persist database data.
- `ghcr.io/linuxserver/sqlite:latest`: Specifies the latest SQLite image.

This container will host the SQLite database file. Your Python FastAPI app will connect to this database using the path `/data/tasks.db` (update your `DB_PATH` environment variable accordingly if needed).

Run the Task Manager container

```
docker run -d --name task-api --network tasknet -p 3000:3000 -e DB_HOST=mariadb -e DB_USER=user -e DB_PASSWORD=pass -e DB_NAME=tasksdb task-api
```

- `docker run -d`: Runs the container in **detached mode** (background).
- `--name task-api`: Names the container `task-api`.
- `--network tasknet`: Connects the container to the `tasknet` Docker network.
- `-p 3000:3000`: Maps port **3000** on the host to **3000** in the container (used by the API).
- **Environment variables (`-e`)**:
  - `DB_HOST=mariadb`: Specifies the database host (the MariaDB container).
  - `DB_USER=user`: Sets the database username.
  - `DB_PASSWORD=pass`: Sets the database password.
  - `DB_NAME=tasksdb`: Specifies the database name to connect to.
- `task-api`: Uses the `task-api` image to run the container.

The two containers should now be running. Check that by running

```
docker container ls -a
```

Step 4: Test the API

Try all endpoints

`curl http://localhost:3000/tasks`

Appendix

```
# View running containers
docker ps

# View logs for MariaDB (or any container)
docker logs mariadb

# Inspect container details (network, volumes, etc.)
```

```
docker inspect mariadb

# View container resource usage
docker stats

# Stop a container
docker stop mariadb

# Remove a container
docker rm mariadb

# Remove the Docker network
docker network rm tasknet

# List all Docker networks
docker network ls

# Run an interactive container on the same network for debugging
docker run -it --rm --network tasknet alpine sh

# Run a temporary container with Adminer, a web-based database explor
er.
# Access it at http://localhost:8080
# Connect using:
#   System:    MariaDB
#   Server:    mariadb
#   Username:  user
#   Password:  pass
#   Database:  tasksdb
docker run --rm -d --name adminer --network tasknet -p 8080:8080 admi
ner
```