Docker and Docker Compose

Alessandro Bocci

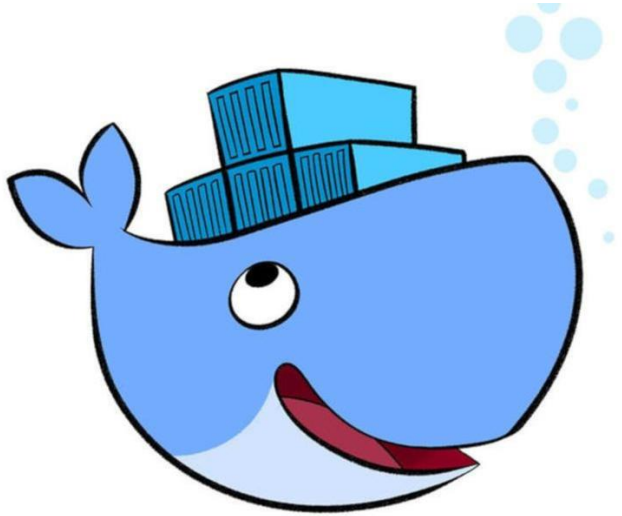
name.surname@phd.unipi.it

Advanced Software Engineering (Lab)

25/10/2023

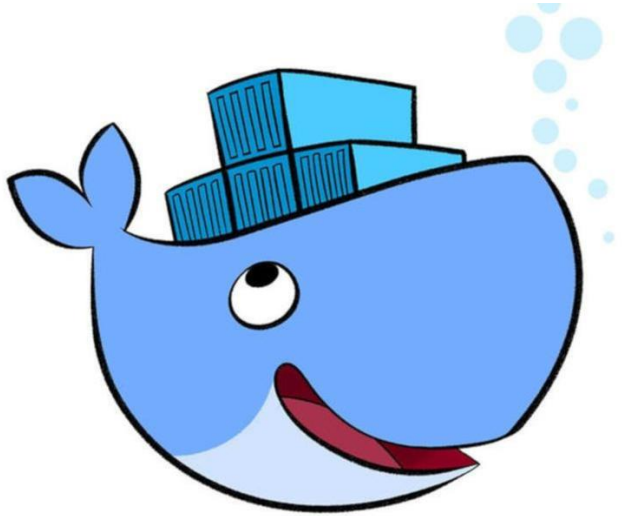
What will you do?

- Complete a multi-service application.
- Write `Dockerfile`(s) to create images to deploy your services.
- Use Docker Compose to run your multi-service application.



Software Prerequisites

- Docker Engine
- Docker image `python:3.9.18-slim`



Today's Lab

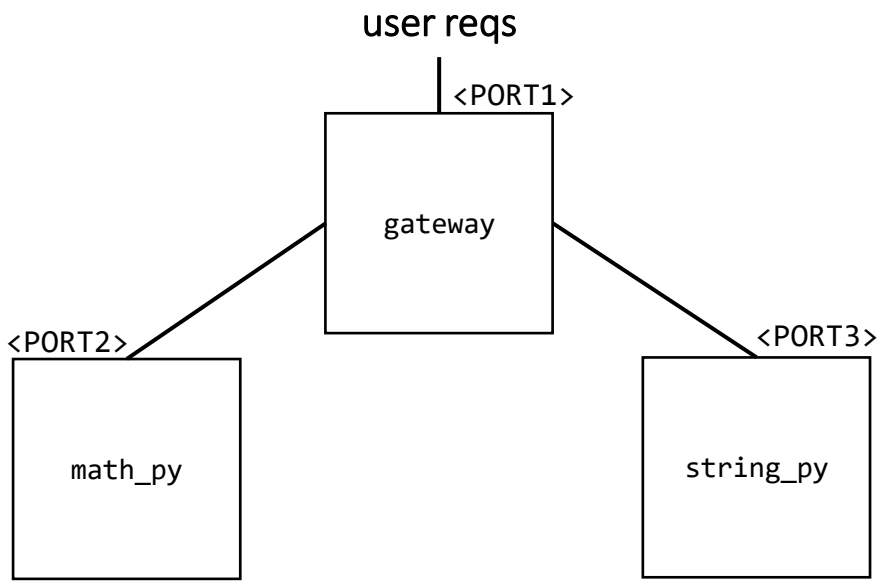
Download the code of the **`microase2324`** app from the Moodle.

PART ONE

1. Complete the **`math`** service code to feature the requested operations,
2. Write the **`Dockerfile`** for the **`math`** service,
3. Build the image of the **`math`** service,
4. Run the **`math`** service and call its API.

PART TWO

1. Complete the **`gateway/urls.py`** file with suitable service name and ports.
2. Write **`Dockerfile`** for the **`gateway`** and **`string_py`** services.
3. Complete the **`docker-compose.yml`** file to run all three services.
4. Run the whole application through Docker Compose.



PART ONE in details

From the **math_py** folder:

1. Code in the **app.py** file adding **subtraction**, **multiplication**, **division** and **modulus** operations exposing the following API:
   - `/sub?a=«float»&b=«float»`
   - `/mul?a=«float»&b=«float»`
   - `/div?a=«float»&b=«float»`
   - `/mod?a=«float»&b=«float»`
2. Write the `Dockerfile` for building a docker image starting from `python:3.9.18-slim` ending the file with: **CMD ["flask", "run", "--host=0.0.0.0", "--port=<your_port>"]**
   (DockerFile docs @ https://docs.docker.com/engine/reference/builder/)
3. Build the image of the `math` service (https://docs.docker.com/engine/reference/commandline/build/)
4. Run the `math` service (https://docs.docker.com/engine/reference/run/). Be careful with the ports!
5. Try it! Using a browser (or an HTTP client) invoke the service sending GETs to it, e.g:

   `http://127.0.0.1:PORT/add?a=2&b=1` should return a JSON with a field `s = 3`

Dockerfile commands cheat sheet

command	description
FROM *image*	base image for the build
COPY *path dst*	copy *path* from the context into the container at location *dst*
ADD *src dst*	same as COPY but accepts archives and urls as *src*
RUN *args…*	run an arbitrary command inside the container
CMD *args…*	set the default command
USER *name*	set the default username
WORKDIR *path*	set the default working directory
ENV *name value*	set an environment variable
EXPOSE *port(s)*	allow the container to listens on the network *port(s)*
ENTRYPOINT *exec args…*	configure a container that will run as an executable

The `Dockerfile` is a script file having (some of) those command that are executed in order. Be careful to call it 'Dockerfile' with capital D and without extension.

Docker cheat sheet

IMAGES

Docker images are a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Build an Image from a Dockerfile
```
docker build -t <image_name>
```

Build an Image from a Dockerfile without the cache
```
docker build -t <image_name> . –no-cache
```

List local images
```
docker images
```

Delete an Image
```
docker rmi <image_name>
```

Remove all unused images
```
docker image prune
```

CONTAINERS

A container is a runtime instance of a docker image. A container will always run the same, regardless of the infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging.

Create and run a container from an image, with a custom name:
```
docker run --name <container_name> <image_name>
```

Run a container with and publish a container's port(s) to the host.
```
docker run -p <host_port>:<container_port> <image_name>
```

Run a container in the background
```
docker run -d <image_name>
```

Start or stop an existing container:
```
docker start|stop <container_name> (or <container-id>)
```

Remove a stopped container:
```
docker rm <container_name>
```

Open a shell inside a running container:
```
docker exec -it <container_name> sh
```

Fetch and follow the logs of a container:
```
docker logs -f <container_name>
```

To inspect a running container:
```
docker inspect <container_name> (or <container_id>)
```

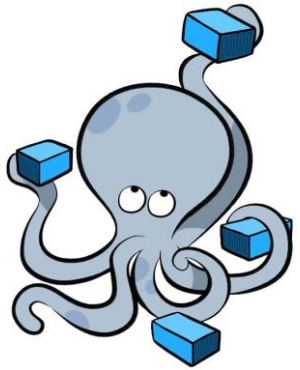
To list currently running containers:
```
docker ps
```

List all docker containers (running and stopped):
```
docker ps --all
```

View resource usage stats
```
docker container stats
```

from https://docs.docker.com/get-started/docker_cheatsheet.pdf

PART TWO in details



1. Complete the `gateway/urls.py` file with suitable service name and ports.
2. Write a `Dockerfile` for the `gateway` and `string_py` services (they will be very similar to the `math` one)
3. Complete the `docker-compose.yml` file to run all three services using the names and ports of point 1.
4. Run the whole application through Docker Compose (in the main folder `docker compose up`).
5. Try It!

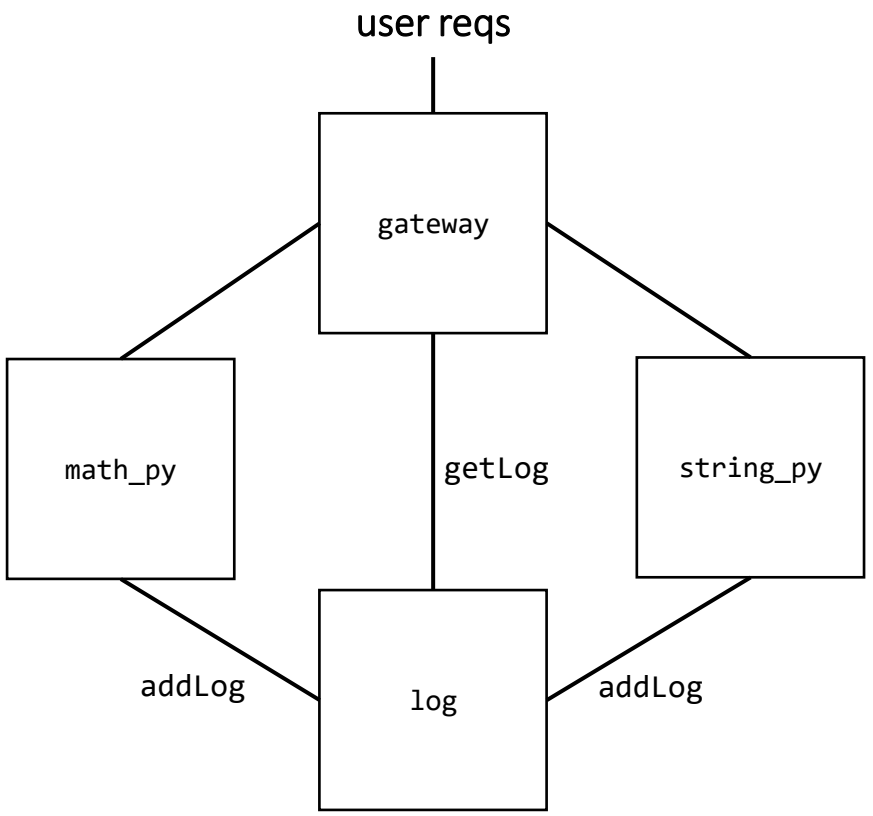
`http://127.0.0.1:PORT/math/add?a=2&b=1`	should return a JSON with a field `s = 3`
`http://127.0.0.1:PORT/math/div?a=2&b=1`	should return a JSON with a field `s = 2`
`http://127.0.0.1:PORT/math/div?a=2&b=0`	should return an error
`http://127.0.0.1:PORT/str/concat?a=2&b=1`	should return a JSON with a field `s = "21"`
`http://127.0.0.1:PORT/str/upper?a=ase`	should return a JSON with a field `s = "ASE"`
`http://127.0.0.1:PORT/str/lower?a=aSE`	should return a JSON with a field `s = "ase"`

BONUS STAGE!

Bonus stage

Add a new `log` service, preferibly in a language different from Python.

1. Write its code and its `Dockerfile`.
2. It should exposes two endpoints
   - `/getLog` which returns all the complete operations from the `math` and the `string` services, should be invoked by the `gateway` service after a user request,
   - `/addLog` which allows to send the log of a complete operation including the timestamp, the invoked service, the operation, the arguments and the result. It should be invoked by the `math` and the `string` services. The log could be sent in a JSON file or in the query string.
3. Change the code of all the previous services for the invocation of the log service and the docker compose file.
4. Try the new application!



Lab take away

- ❑ Write a `Dockerfile`
- ❑ Build and run a docker image
- ❑ Deploy and run a multiservice application with Docker Compose

Solution PART ONE: math's Dockerfile example

```
FROM python:3.9.18-slim

ADD . /math_py
WORKDIR /math_py

RUN pip3 install -r requirements.txt

EXPOSE 5000

CMD ["flask", "run", "--host=0.0.0.0", "--port=5000"]
```

Commands:

```
docker build -t math .
docker run -p 5000:5000 math
```

HTTP GET:

```
http://127.0.0.1:5000/add?a=1&b=2
```

Solution PART TWO: `docker-compose.yml` example

```
version: '1'
services:
  math-service:
    build: ./math_py
    container_name: math-service

  string-service:
    build: ./string_py
    container_name: string-service

  gateway:
    build: ./gateway
    container_name: gateway
    ports:
      - 5000:5000
    depends_on:
      - math-service
      - string-service
```

`version` could be **`'3'`** depending on which version you installed

gateway/urls.py:

```
GATEWAY_URL = 'http://gateway:5000'
MATH_URL = 'http://math-service:5000'
STRING_URL = 'http://string-service:5000'
```

(exposing port **`5000`** in all the Dockerfiles)