Lab 4: OS-level virtualisation with Docker

What Is Docker?

Docker describes themselves as "an open platform for developers and sysadmins to build, ship, and run distributed applications".

Exercise 1. Setup

1. **Install** Docker Engine - Community (docker-ce) on your operating system. If you have Linux do not forget to do the Post-installation steps!

Terminology

1. **Images.** Image is a snapshot of virtual machine. You can pull existing from registry (eg docker hub), build new from scratch or, which is more commonly used, build your own based on some existing.
   - To pull image from hub.docker.com you can use

     ```
     $ docker pull <name>
     ```

     For example:

     ```
     $ docker pull python:3.7
     ```

   - To list all images on your machine use

     ```
     $ docker images
     ```

   - To delete particular image use

     ```
     $ docker image rmi <name>
     ```

2. **Containers.** When you run an image – you create the instance of that image, called container. To create a new instance of an image you need to run:

   ```
   $ docker run --name <container_name> -it ubuntu
   ```

   Where `--name <container_name>` specifies name of new container and `-i` stands for interactive mode with `-t` used for allocate tty (provide output from container). Another useful option could be `--rm` to delete instance right after you close it. Usually container is a sandboxed process running an application and its dependencies. Also container can be compared with virtual machine: it has its own disk space, isolated runtime, can be stopped and resumed.

   - To find currently running container run:

     ```
     $ docker ps
     ```

   - To find all existing containers on your machine run:

     ```
     $ docker ps -a
     ```

   - To resume/pause existing containers run:

     ```
     $ docker start/stop
     ```

   - To attach local standard input, output, and error streams to a running container run:

     ```
     $ docker attach <name>
     ```

3. **Volumes.** Can be thought as external storage independent from container instance. In simple case it could be just mapping of folder from host machine to folder inside container. Or you can give docker maintain volumes it self. In the second case you may configure it more precisely for instance create read-only volumes or use Amazon s3 as an actual file storage.

4. **Dockerfile**. Now let’s take a look at how we can build our own image with needed application:

   ```
   # Specify base image
   FROM python:3.7-alpine
   # Copy particular file
   COPY requirements.txt /tmp/
   # Execute commands inside container, so that you can customize it
   RUN pip3 install --no-cache-dir -r /tmp/requirements.txt
   # Create and cd to this directory, set default command to run container
   WORKDIR /app
   # Copy files from project dir into container’s folder
   COPY ./ /app
   # Specify port that could(!) be opened from container later
   EXPOSE 8080
   CMD python3 main.py
   ```

   Save it to file called ‘Dockerfile’ and you’re ready to build your own image with

   ```
   $ docker build -t <name> ./
   ```

   (note the `./` at the end of command, this specifies there is Docker file located).

   It’s **very important** to know that docker images works pretty similar to how git does. After each of commands ADD, COPY, RUN docker creates new layer in image (=commit in git terminology). So that if you have two images based on ubuntu – docker will download base image only once and reuse it for all successors (you can think of it like branches in git). Another consequence is that if you’ve added something in one layers it will never leave this image (because it has been committed).

   Hint: To remove all unused images after experiments you can use

   ```
   $ docker system prune -a
   ```

Exercise 2. Web App

1. Create python application which serves simple web page. You may use *http.server* module or choose any web framework (django, flask, …).

   Hint: if you have difficulty, look here.

2. Wrap it to docker. Create Dockerfile similar to shown above. Build image and run the container.
3. Run it. Don’t forget to expose 80 port to host machine. You can do it with `-p 80:80` (`<host port> : <container port>`).
4. Now add static files: favicon.ico and any png with your favorite meme. Include the meme to your html page.

What Is Docker Compose?

Docker-compose is used to run several containers using configuration file instead of executing commands in shell. The configuration is stored in yaml file. There we can specify exactly the same things as we do through **cli**.

File structure and commands

1. The structure of the **docker-compose.yaml** file:

   ```
   version: '3.3'
   services:
      db:                             # name of the service
        image: mysql:5.7              # image from docker registry
        volumes:                      # mounted folders or disks
          - db_data:/var/lib/mysql    # from host : in container
        restart: always               # restart policy
     web:
       build: .                       # build from Dockerfile
       command: python main.py        # command to start container
       volumes:
         - .:/code
       ports:                         # expose ports to host from container
         - "8000:8000"                # host : container
       depends_on:
         - db                         # specify order of running services
   volumes:
       db_data: {}                    # define volumes (virtual "disks")
   ```

2. To build necessary images, creates network, volumes, containers and starts them run:

   ```
   $ docker-compose up
   ```

3. To stop and delete everything created by *up* run:

   ```
   $ docker-compose down
   ```

Exercise 3. Docker Compose

1. Install docker-compose on your operating system.
2. Add your web application to docker-compose.yaml. Make sure it still works.

   Hint: if you have difficulty, look here.

3. Add **nginx** as the second service to docker-compose. It is recommended to use `nginx:alpine` – image with the smallest size.
4. Make changes in configuration: don't expose any ports from web application service, setup dependency between nginx and your app (so that nginx will start only when your app is ready).

5. Docker-compose creates virtual network between services and uses services' names as domain names for them. For instance if your service called *app* it will be available from all neighbors as http://app:8080.
6. Configure nginx to serve your static files (for now only .ico, .jpg and .png files) other requests should be passed to python application. Hint: If you're not familiar with nginx configuration look here. Save it as **nginx.conf** and map it into container as **/etc/nginx/nginx.conf**. Mount folder with your static files to **/www/media** as well.
7. Run `docker-compose up` and make sure that static files and web page both available.

What Is Docker Hub?

It is the official registry for docker. Images which we've used on previous steps were downloaded from there. In free plan docker hub provides one private repository and unlimited public repositories.

Exercise 4. Docker Hub

1. Register on Docker Hub.
2. Tag your image

```
$ docker tag <current_image_name> <your_username_on_hub>/lab4_pyapp
```

3. Push it to the docker hub:

```
$ docker push <your_username_on_hub>/lab4_pyapp
```

Exercise 5. Deploy Docker Image

1. Create an AWS EC2 instance and install `docker` and `docker-compose` there.
2. Modify **docker-compose.yaml** file to use image of the application from your docker hub.
3. Run whole application as daemon

```
$ docker-compose up -d
```

Assignment

Add "visitor counter" to the application from previous steps. You can use Redis with *INC* function and just count page loading. Database should be located in separate container.

As the submission provide **url** to running app on aws instance and **screenshot** of output `docker ps` command on that instance.