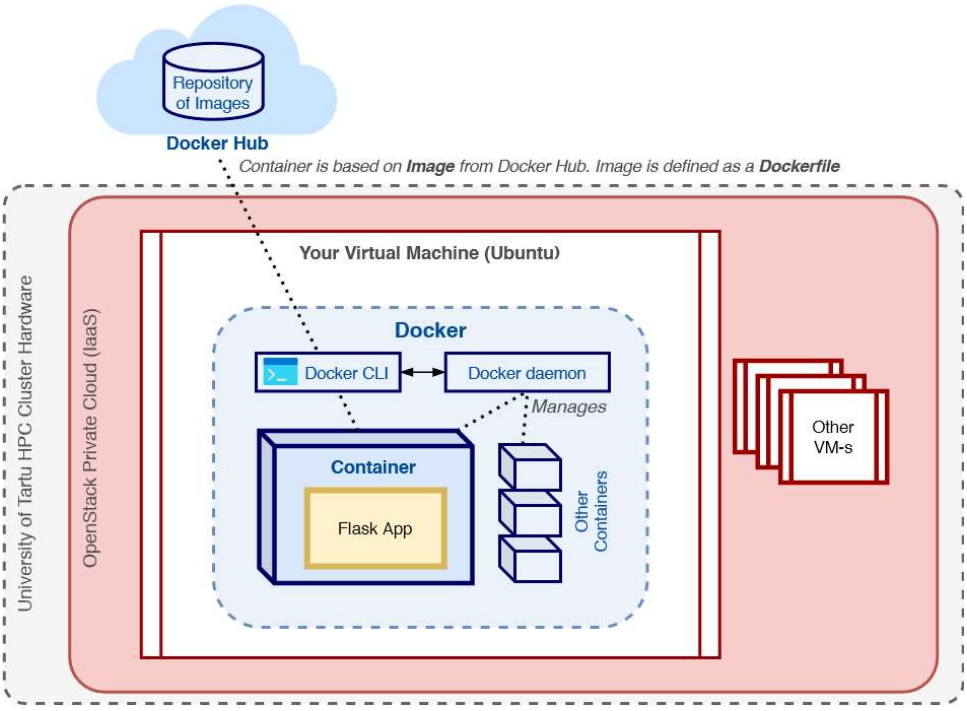
Cloud Computing 2022/23 spring

Practice 2 - Working with Docker

In this lab, we will take a look at how to install Docker, use Docker CLI commands, and how to containerize applications. Docker containers allow the packaging of your application (and everything that you need to run it) in a "container image". Inside a container, you can include all necessary libraries, files and folders, environment variables, volume mount points, and your application binaries.



Key terminology:

- Docker **image**
  - Lightweight, stand-alone, executable package that includes everything needed to run a piece of software
  - Includes code, a runtime, library, environment variables, and config files
- Docker **container**
  - Runtime instance of an image - what the image becomes in memory when actually executed.
  - Completely isolated from the host environment.

Build, Ship, and Run Any App, Anywhere

- Docker is available in two editions: Community Edition (CE) and Enterprise Edition (EE)
- Supported Platform: macOS, Microsoft Windows 10, CentOS, Debian, Fedora, RHEL, Ubuntu and more.

References

Referred documents and websites contain supportive information for the practice.

Manuals

1. Docker fundamentals: https://docs.docker.com/engine/docker-overview/
2. Docker CLI: https://docs.docker.com/engine/reference/commandline/cli/
3. Building docker image: https://docs.docker.com/engine/reference/builder/
4. Docker architecture: https://docs.docker.com/engine/docker-overview/

In case of issues check:

1. Check pervious messages in the `#practice-session-2` Zulip channel.
   - Ask in the `#practice-session-2` Zulip channel.
2. Possible solutions to common issues section at the end of the guide.

Exercise 2.1. Installation of docker inside OpenStack instance

In this task, you are going to install docker in *Ubuntu OS* and try to run the basic commands to make you comfortable with the Docker commands used in the next tasks.

- Create a virtual machine with *ubuntu20.04 OS* as carried out in Practice1 and connect to the virtual machine remotely using via SSH.
  - NB! DO NOT use your previous lab image/snapshot!
- NB! Extra modifications required to change docker network settings:
  - Create a directory in the virtual machine in the path: `/etc/docker`
    - `sudo mkdir /etc/docker`
  - Create a file in the docker directory: `sudo nano /etc/docker/daemon.json` with the following content:

```
{
  "default-address-pools": [{ "base":"172.80.0.0/16","size":24 }]
}
```

  - This change is required because otherwise, Docker will use network addresses that collide with the university networks, and you WILL lose access to the instance.
- Update the apt repo `sudo apt-get update`
- Install packages to allow apt to use a repository over HTTPS:
  - `sudo apt-get install apt-transport-https ca-certificates curl gnupg-agent software-properties-common`
- Add Docker's official GPG key
  - `curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -`
- Use the following command to set up the stable repository.
  - `sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"`
- Update the apt package index, install Docker
  - `sudo apt-get update`
  - `sudo apt-get install docker-ce`

NB! To run a docker command with non-root privileges

- Add the current user to docker group: `sudo usermod -aG docker $USER`
  - If the docker group does not exist for some reason, you can create it by using:
    - `sudo groupadd docker`
- To Activate the changes:
  - Exit the terminal/ssh session and rejoin (ssh) into the virtual machine or run `newgrp docker`.
- Check the installation by displaying docker version: `docker --version`

Exercise 2.2. Practicing docker commands

This task mainly helps you to learn basic commands used by docker CLI such as run, pull, listing images, attaching data volume, working with exec (like ssh a container), checking IP address, and port forwarding. You can have a look into basic docker commands here: Docker commands

The following tasks include: Pulling an image from Docker Hub and running an Ubuntu container' *in detached mode* *https://docs.docker.com/engine/reference/run/#detached-vs-foreground detached mode and assigning your name as container-name, further installing HTTP server (use* docker exec'' command) and use port forwarding to access container-HTTP traffic via host port 80.

- Create a login account at Docker Hub sign-up page
- Login into your docker account from the Docker host terminal: `docker login`
  - Provide input to the following:
    - Username: your docker hub id
    - Password: Docker hub password
- NB! The login part is not mandatory but needed/recommended as recently docker hub limits the number of Docker pulls from a particular IP. As you are using the university network through VPN, it serves as a single IP for the Docker hub. (*Error response from daemon: toomanyrequests: You have reached your pull rate limit. You may increase the limit by authenticating and upgrading: https://www.docker.com/increase-rate-limit*)
- Pull an image from docker hub:`docker pull ubuntu`
- Check the downloaded images in the local repository: `docker images`

- Run a simple ubuntu container, here detached mode (-d),-it runs interactively (so you get a pseudo-TTY with STDIN) : `docker run -dit -p 80:80 --name <<yourname>> ubuntu`
    - <<yourname>> = please type your name
- Get the bash shell of the container with non-detached mode: `docker exec -it <container_name> sh`
    - Here you could start configuring your container: install packages, modify files, etc.. Alternatively, you can also run commands "outside" of the container using *docker exec*, as we will do in the next steps.
    - Exit from the container: `exit`
- Connect to container and update the apt repo: `docker exec -dit <container_name> apt-get update`
- Install HTTP server: `docker exec -it <container_name> apt-get install apache2`
- Check the status of HTTP server: `docker exec -it <container_name> service apache2 status`
- If not running, start the HTTP server: `docker exec -it <container_name> service apache2 start`
- Check the webserver running container host machine: `curl localhost:80`
- Check the IP address of the container: `docker inspect <container_id> | grep -i "IPAddress"`
- Commit the docker container changes in to docker image `docker commit -m "added apache2 web server" -a "your_name" <container_id> "image_name"`
- Check the newly created image using `docker images`
- Now, stop and delete your containers using
    - `docker stop CONTAINER_NAME_OR_ID`, `docker rm CONTAINER_NAME_OR_ID`

- **Host directory as a data volume:** Here you are mounting a host directory in a container and this is useful for testing the applications. For example, if you store source code in the host directory and mount it in the container, the code changed in the host directory file can affect the application running in the container.
    - Accessing a host file system on the container with read-only and read/write modes:
        - Create directory with name test: `mkdir test && cd test`, Create a file: `touch abc.txt`
        - Run a container and -v parameter to mount the host directory to the container
            - Read only: `docker run -dit -v /home/ubuntu/test/:/data/:ro --name vol1 ubuntu sh`
            - Access the file in a container in the path /data and try to create a new file (you should see access denied) from container `docker exec -it vol1 sh`, `cd /data`, `ls`, `exit`.
            - Read/write: `docker run -dit -v /home/ubuntu/test/:/data/:rw --name vol2 ubuntu`, `docker exec -it vol2 sh`, `cd /data`, `ls`. Try to create some text files. You should then see the created files in the host machine `/home/ubuntu/test/` folder.
    - NB! Take the screenshot here after running `docker ps` command
- Now, stop and delete your containers using
    - `docker stop CONTAINER_NAME_OR_ID`, `docker rm CONTAINER_NAME_OR_ID`

- **Data volume containers:** A popular practice with Docker data sharing is to create a dedicated container that holds all of your persistent shareable data resources, mounting the data inside of it into other containers once created and set up.

    - Create a data volume container and share data between containers.
        - Create a data volume container `docker run -dit -v /data --name data-volume ubuntu`, `docker exec -it data-volume sh`
        - Go to volume and create some files: `cd /data && touch file1.txt && touch file2.txt` Exit the container `exit`
        - Run another container and mount the volume as earlier container: `docker run -dit --volumes-from data-volume --name data-shared ubuntu`, `docker exec -it data-shared sh`
        - Go to the data directory in the created container and list the files: `cd /data && ls`
    - NB! Take the screenshot here after using `docker ps`
- Now, stop and delete your containers using
    - `docker stop CONTAINER_NAME_OR_ID`, `docker rm CONTAINER_NAME_OR_ID`

Exercise 2.3. Creating Docker Image using Dockerfile

The task is to create your own docker image and try to run it. More information about Dockerfile (https://docs.docker.com/engine/reference/builder/). A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Using docker build users can create an automated build that executes several command-line instructions in succession.

Some of Dockerfile commands:

FROM	Set the base image
LABEL	Add a metadata to image
RUN	Execute the command and execute the commands
CMD	Allowed only once
EXPOSE	Container listen on the specific ports at runtime
ENV	Set environment variables
COPY	Copy the files or directories into container's filesystem
WORKDIR	Set working directory

The scenario is to create a docker image and deploy the flask application from Practice Session 1 (Message Board) using the docker container. In this exercise, we are going to perform three tasks:

1. Create docker **Image** for flask application (Message Board)
2. **Deploy** flask application container using the host directory as a volume (bind volumes) to store messages in the host directory (data.json in flask application).
3. **Deploy** a different version of the flask application, linking to PostgreSQL relational **database container** to store the data, instead of a JSON file.

Task 2.3.1: Create a Docker image for the flask application

- First, fetch the source code of the application with git, storing it in the folder "lab2app":
  - `git clone https://bitbucket.org/jaks6/cloud-computing-2022-lab-1.git lab2app`
- change to lab2app `cd lab2app`
- Modify the `home.html` file in your own style to display your name.
  - This file is located in the templates sub-folder
- Now, lets write a Dockerfile to create a docker image as shown below:
  - Create a Dockerfile (**inside** the lab2app directory. NOT templates folder) `nano Dockerfile`
  - Add the following set of Docker commands:
    FROM command chooses the base image for the container, here its python:slim-buster
    COPY command will copy the code from the host directory to the container directory lab2app (Your copying the Messageboard code to the container image)
    WORKDIR Set the current working directory when your container starts.
    RUN command will run the specified commands on the shell.
    CMD command will run the flask application at a specified host address and port.

```
FROM python:slim-buster
WORKDIR /lab2app
COPY . .
RUN pip3 install -r requirements.txt
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
EXPOSE 5000
CMD [ "flask", "run"]
```

- Build the container image using the following command **docker build** as `docker build -t <flask_task1_lastname> .` **inside the lab2app** folder.
  - Now, look at the output of the build command where a set of docker commands are executed in the order you wrote in Dockerfile.
  - After a successful build, you should see your built image in the list of local images when calling `docker images`
- Now, run the container with the following options:
  - port mapping: Map the host 80 port to container port 5000 (where flask is listening for traffic)
    - For port mapping, add `-p 80:5000` to the docker run command
  - image name: `<flask_task1_lastname>`
  - Name of container: `<task1_your_lastname>`
- After this, you should see the application running at `http://VM_IP:80` through browser
- Stop and delete the container

Task 2.3.2: Deploy Flask app container with a mounted-volume-based storage

Deploy flask application container using the host directory as a volume(bind volumes) to store messages in the host directory (data.json in flask application).

- Copy the data.json file (inside the lab2app folder) into the ubuntu user home folder inside the VM in the path (`/home/ubuntu/data.json`).
- Now run the container with the following options:
  - detached mode (-dit)
  - Container image `<flask_task1_lastname>`
  - Name of container `<task2_your_lastname>`
  - port mapping host(80), container(5000)
  - Volume (-v) `/home/ubuntu/data.json:/lab2app/data.json`
    - This will mount the file you previously created (inside the host file) into the correct location inside the container.
- By now, you should see the application running at `http://VM_IP:80` and add few messages
- Check the content of data.json `cat /home/ubuntu/data.json`, here you can see the added messages.
- Stop and remove the container
- Run the container again and should see the data of the previous container's application in http://VM_IP:80 web page
- NB! Take the screenshot of `cat /home/ubuntu/data.json` inside the VM and `docker exec -it <task2_your_lastname> cat /lab2app/data.json`
- Stop and delete the container

Task 2.3.3: Deploy Flask app container with a database container for storage

Deploy flask application with linking to PostgreSQL relational database container to store the data.

- Make a new folder for storing database data:
  - `sudo mkdir /mnt/data`
  - We will mount this to postgre container to store database files there
- Firstly run the Postgresql database container using the following command and set the value of PostgreSQL configuration variable `POSTGRES_PASSWORD` according to your convenience in `docker run -d --name postgres -e POSTGRES_USER=postgres -e POSTGRES_PASSWORD=<your_paasword> -e PGDATA=/var/lib/postgresql/data/pgdata -v /mnt/data:/var/lib/postgresql/data -p 5432:5432 -it postgres:14.1-alpine`
- Now let us clone the modified code to store messages in the postgresql container. You use the clone as `git clone https://bitbucket.org/shivupoojar87/task3lab2app.git task3lab2app`. In this code, we used SQLAlchemy with the Flask framework to interact with Postgresql.
- Change the current directory to task3lab2app and copy the Dockerfile of Task 2 in this directory.
- Modify the home.html to display your name and task number
- Rebuild the image with the name <flask_task3_lastname>
- Now run the container with the following options:
  - detached mode (-dit)
  - Container image: `<flask_task3_lastname>`

- Name of container: `<task3_your_lastname>`
- port mapping : host(80), container(5000)
- Set environment variable for linking the postgres database (-e): `DATABASE_URL=postgresql://postgres:<your_paasword>@<VM_IP>:5432/postgres`
- The whole command for example should look like: `docker run -dit -p 80:5000 --name flask -e DATABASE_URL=postgresql://postgres:4y7sV96vA9wv46VR@172.17.67.119:5432/postgres flask_poojara:latest`
  - NB! make sure to change image name (flask_poojara) in the example to match your image name
  - NB! make sure to change the password and the IP address in this command to match the information of your postgres container.
  - For IP address you can use the IP of the host VM (replacing 172.17.67.119 in the example command)

- By now, you should see the application running at http://VM_IP:80 and add a few messages.
- NB!! Take the screenshot of running containers (docker ps) and the web page (through browser)
- Stop and remove the postgresql container
- Again create postgresql container and still you can see the previous messages in the flask application. The volumes are important to persistent your data in the host even though the container is deleted or killed.

Exercise 2.4. Shipping a docker image to the Docker hub

Docker hub is a hosted repository service provided by Docker for finding and sharing container images with your team. (https://www.docker.com/products/docker-hub)

- Create a login account in the Docker hub (if you do not have one already) using the link https://hub.docker.com/signup .
- **Push the docker image to docker hub**
  - Initially login into your docker account from docker host terminal: `docker login`
    - Provide input to the following:
      - Username: your docker hub id
      - Password: Docker hub password
  - Tag your image before pushing into docker hub: `docker tag <flask_task3_lastname> your-dockerhub-id/flask1.0`
  - Finally push the image to docker hub: `docker push your-dockerhub-id/flask1.0`
- Test that you can start a new container from the DockerHub image you have just created and updated.
  - `docker run -dit -p 80:5000 --name flask -e DATABASE_URL=postgresql://postgres:4y7sV96vA9wv46VR@172.17.67.119:5432/postgres your-dockerhub-id/flask1.0:latest`
  - After replacing **your-dockerhub-id** with your own Dockerhub account

NB! Take a screenshot of the Docker Hub web page which shows information about your image

Bonus task - Working with docker-compose.

Docker Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration. To learn more about all the features of Compose, see the list of features.

The compose has commands for managing the whole lifecycle of your application:

1. Start, stop, and rebuild services
2. View the status of running services
3. Stream the log output of running services

In this exercise, we are going to use docker-compose commands to start/stop/view the status of multiple services (flask application and Postgresql database service). We have two tasks to perform:

Deliverables

- Upload the screenshot taken as mentioned in Exercise 2.2(2 screenshots), Exercise 2.3 (2 screenshots), Exercise 2.4.
- Source code of two flask applications and docker-compose files should be included (You can copy to host machine from VM using `scp` command and Windows users can use WinSCP tool).
- Pack the screenshots into a single zip file and upload them through the following submission form.
- Your instance must have been terminated!

Task	Lab 2 - Docker
Current submission	ZIP (23:33 16.02.2023)
	If your homework consists of multiple files (for example HTML + CSS + JS) it should be archived before submitting.
File	Choose File No file chosen
Comment	

Submit

In case of issues, check these potential solutions to common issues:

1. If you run into an issue about `missing requirements.txt` when building the Docker file, it is likely because of the location of the Docker file and the folder in which you run the docker build command.
    - I would suggest making sure that:
        - Dockerfile is inside the lab2app folder
        - run the docker build command inside the folder
    - Otherwise the COPY command has to use slightly different relative paths inside the example Docker file.
2. Seems that there is an issue with the university VPN after the software was updated. the old guide no longer works.
    - You can use the suggested approach by one student:
        - Log into UT VPN service and download the profile file: https://tunnel.ut.ee/
        - Download and install the official OpenVPN Connect client: https://openvpn.net/download-open-vpn/ and import the profile file.
3. If you get an error: "Error response from daemon: toomanyrequests: You have reached your pull rate limit. You may increase the limit by authenticating and upgrading: https://www.docker.com/increase-rate-limit)"
    - Make sure you have loged into docker from the SSH command line using your Docker Hub credentials
4. If you get an error:
    - permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Get "http://%2Fvar%2Frun%2Fdocker.sock/v1.24/containers/json": dial unix /var/run/docker.sock: connect: permission denied
    - Make sure you have added the Ubuntu user to the docker group.
    - refresh the group list of the user by logging out and logging back over SSH
    - Or use sudo command to run docker commands in elevated permissions ()
5. You get error when logging into Docker from command line:
    - Error response from daemon: Get "https://registry-1.docker.io/v2/": unauthorized: incorrect username or password
    - Use username instead of email for the username
6. If you get an error that container with a specific name already exists:
    - docker: Error response from daemon: Conflict. The container name "/data-volume" is already in use by container "4e4c97883e82ecaa93fa1f89d7538011c8889f80874c8ab49f257d28c8459c03". You have to remove (or rename) that container to be able to reuse that name.
    - Then you can delete the existing container using `docker rm` or use a different name
        - If needed also stop the container first using `docker stop`

The courses of the Institute of Computer Science are supported by following programs: