Lab: Docker Compose

Problems for the lab for the "Containers and Clouds" course @ SoftUni.

In the exercises below we will learn how to **work with Docker images**, **Dockerfiles**, **volumes**, **networks**, **Docker Compose**, etc. to run **multi-container apps in Docker**.

1. MyWebsite App: Building a Custom Image

Step 1: Create a Dockerfile

Our first task is to **create a Dockerfile for a Vue.js app**, which will allow us to **run it in a Docker container**.

First, we have to go to the **root** folder of the **Vue.js** app that we created and ran in our previous session.

Our next step is **creating a Dockerfile** in this **directory**. The **Dockerfile contains instructions** on how an **image for the app should be created**. As we know, **Dockerfiles** are **just text files**, so we can create our own and open it with a text editor of our choice. Note that the name of the file should be "**`Dockerfile`**" without any extensions.

The content should be as shown below:

```
FROM node:16

WORKDIR /app

COPY . .

RUN npm install

CMD ["npm", "run", "dev"]
```

Each Dokerfile starts with "**FROM**", so we start creating an image, based on the existing image **`node:16`.**

After that, we will **set** the **"app" folder** as the **current working directory** and we'll **copy** all of the **project files and folders to it.** This will add a layer.

Then, we run the **`npm install`** command in order to **install the necessary dependencies**, so that our app can run. This will form another layer, too.

Finally, we execute the **`npm run dev`** commands, in order to run the scripts that defined inside our app's **`package.json`** file.

Step 2: Build and Publish the Image to Docker Hub

Build the Image

We can now **build a custom image** with this **Dockerfile**. Open a CLI, for example **Powershell**, and fulfill the **following steps** to do it:

- Navigate to the **`MyWebsite` directory**
- Use the **`docker image build`** command to **build the image**
- Set the **local directory** as the **working directory**
- With the **`-f`** option, set the **path to the `Dockerfile`**
- With the **`-t`** option, set the **name of the image** in format **`{your_docker_hub_username}/{app_name}`**, as we will later **add our image to Docker Hub**

The **whole command** should look like this:

```
PS C:\Users\     \MyWebsite> docker image build -t my-webapp .
[+] Building 57.7s (9/9) FINISHED
 => [internal] load .dockerignore                                          0.0s
 => => transferring context: 2B                                            0.0s
 => [internal] load build definition from Dockerfile                       0.0s
 => => transferring dockerfile: 117B                                       0.0s
 => [internal] load metadata for docker.io/library/node:16                 0.0s
 => [1/4] FROM docker.io/library/node:16                                   0.1s
 => [internal] load build context                                         32.8s
 => => transferring context: 99.27MB                                      31.9s
 => [2/4] WORKDIR /app                                                     0.0s
 => [3/4] COPY . .                                                        15.4s
 => [4/4] RUN npm install                                                  8.2s
 => exporting to image                                                     1.3s
 => => exporting layers                                                    1.2s
 => => writing image sha256:975d2fe533817deb55065688f27abc59d3f2ef925eb38e27fbaa4168e  0.0s
 => => naming to docker.io/library/my-webapp                               0.0s
```

Note that we can examine how the **instructions from the Dockerfile** are executed to **build the image**.
We can check the **ready image** with the **`docker images`** command.

```
PS C:\Users\     \MyWebsite> docker images
REPOSITORY          TAG       IMAGE ID       CREATED          SIZE
my-webapp           latest    975d2fe53381   7 minutes ago    1.02GB
```

Publish the image

Now let's see how to **push our custom image to Docker Hub**. Note that this is **not needed** for running a container with that image – you can have the **image only locally** and still use it. However, it is good to know **how to push images**.

To **push our image to Docker Hub**, we should first **log-in to Docker Hub** with the command below. If this is the **first time** you log in, you should **enter your credentials**. Make sure that **login is successful**:

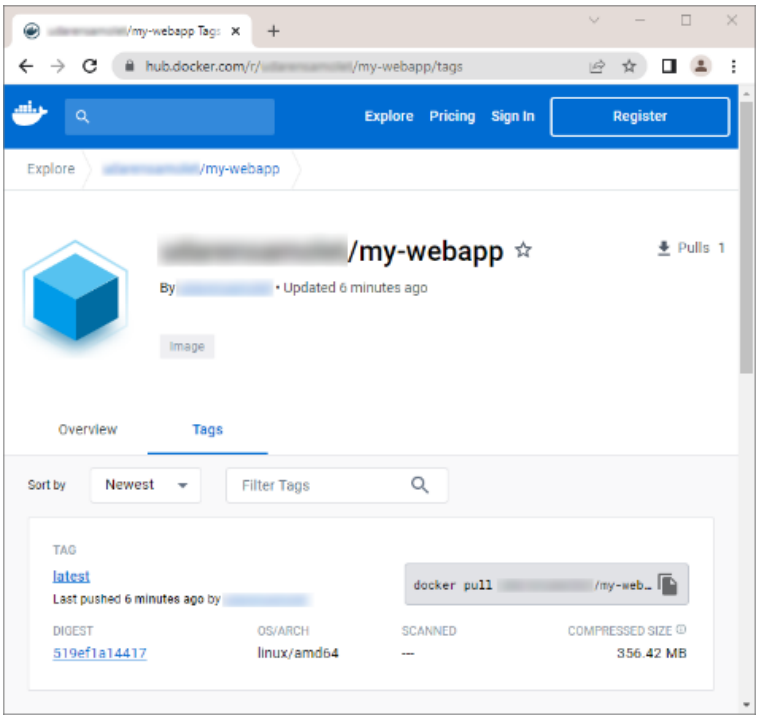
```
PS C:\Users\     \MyWebsite> docker login
Authenticating with existing credentials...
Login Succeeded

Logging in with your password grants your terminal complete access to your account.
For better security, log in with a limited-privilege personal access token. Learn more at ht
tps://docs.docker.com/go/access-tokens/
```

Now you should only **push the image**:

```
PS C:\Users\     \MyWebsite> docker tag my-webapp [redacted]/my-webapp
PS C:\Users\     \MyWebsite> docker push [redacted]/my-webapp
Using default tag: latest
The push refers to repository [docker.io/[redacted]/my-webapp]
f018d0224715: Pushed
60924be50ac8: Pushed
4179ffb128d8: Pushed
c4d7495580fd: Mounted from library/node
4aa2274f91a0: Mounted from library/node
75b410f76042: Mounted from library/node
5c7da2ce555d: Mounted from library/node
c73ad13a1488: Mounted from library/node
f584c095e67e: Mounted from library/node
ee4d330edba0: Mounted from library/node
f689d32da261: Mounted from library/node
latest: digest: sha256:519ef1a14417d7caf25b814e7af48cac1e0ab1be98205d2a0c2
d7808efd18957 size: 2633
```

And it now is **available at Docker Hub** as a **public image**:



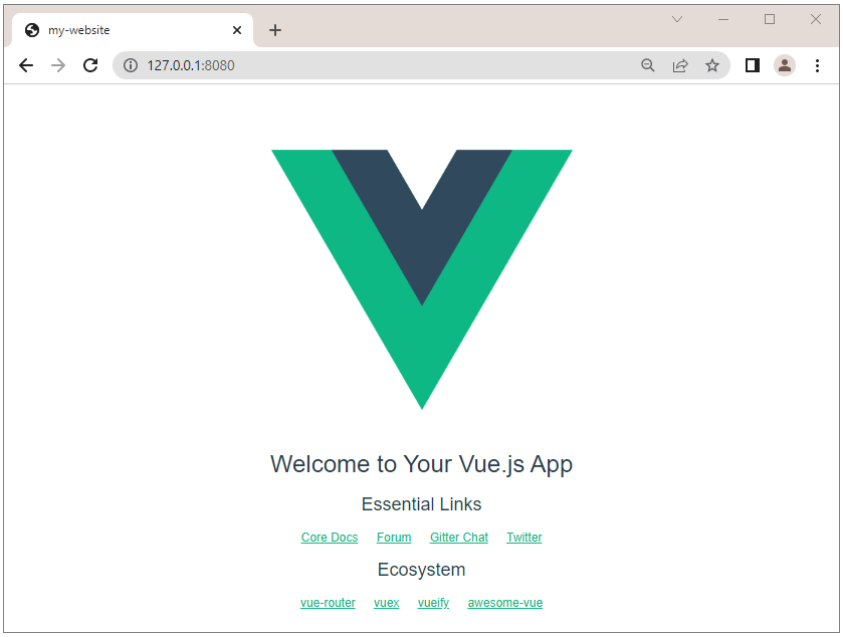
Run the Image as a Container

Finally, let's run the newly-created **image** as a **container** on the **right port**, using the command below:

```
PS C:\Users\      \MyWebsite> docker run -d -p 8080:8080                /my-webapp
ab2b08161571a35ad6faa02dbf823e2524e2742d27e23b8d6c38d519397f5c1b
```

Name	Image	Status	Port(s)	Last started	Actions
kind_goldstine ab2b08161571	/my-webapp	Running	8080:8080	3 minutes ago	

Open your web browser and go to **127.0.0.1:8080**. You should be able to see the **running Vue.js app**:



2. WordPress App with MySQL Database: Connecting Containers in a Network

In this exercise, our task is to **set up and run a WordPress container** in **Docker** with a **MySQL database** by connecting them in a network.

Step 1: Create a Network

First, we have to create a network. Open a CLI and first create a new folder, which will contain the files for our app. Then, create a new network with the **`docker network create {network_name}`** command.

```
PS C:\Users\     \MyApp> docker network create my_network
fdc7a22f370239d21b9c25440251635052ac06f04dde24f4087fc0b40905b49f
```

Step 2: Add MySQL Container to Network

Our next step is **adding** the **MySQL** container to the network that we just created.

The **commands from the resources** are the following:

- **`docker run -dit`** → runs the image detached and in interactive mode;
- **`--name wordpress_db`** → names the container **`wordpress_db`**;
- **`-e MYSQL_ROOT_PASSWORD=pass`** → sets the password for the root MySQL user;
- **`-e MYSQL_DATABASE=wordpressdb`** → sets the name of the MySQL database that we'll use for the WordPress installation;
- **`-e MYSQL_USER=wordpress`** → sets the MySQL user that we'll use for the WordPress installation;
- **`-e MYSQL_PASSWORD=wordpress`** → sets the password for that user;
- **`--expose 3306`** → sets the port of the container;
- **`--expose 33060`** → sets the SSL port of the container;
- **`--network my_network`** → sets the network that we want to attach our container to;
- **`-v ${PWD}/data:/var/lib/mysql`** → maps the directory on our local machine to the directory of the container, so that we can store data outside of it;
- **`mysql`** → the name of the image.

```
PS C:\Users\     \MyApp> docker run -dit `
>> --name wordpress_db `
>> -e MYSQL_ROOT_PASSWORD=pass `
>> -e MYSQL_DATABASE=wordpressdb `
>> -e MYSQL_USER=wordpress `
>> -e MYSQL_PASSWORD=wordpress `
>> --expose 3306 `
>> --expose 33060 `
>> --network my_network `
>> -v ${PWD}/data:/var/lib/mysql `
>> mysql
fd0e62a8b1ee19c06a0125ec831388eb24e95c36bc02ee2c137252fe496b8c7e
```

Step 3: Inspect Network

Now let's inspect our network in order to check if our **`wordpress_db`** container is attached to it:

```
PS C:\Users\     \MyApp> docker network inspect my_network
[
    {
        "Name": "my_network",
        "Id": "36150dcdf9edafba2fe662bdfe50d378a99829be02d3d1b9e9db6f16e3b392c2",
        "Created": "2023-05-14T16:34:32.368057008Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "172.21.0.0/16",
                    "Gateway": "172.21.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```

```
"Containers": {
    "fd0e62a8b1ee19c06a0125ec831388eb24e95c36bc02ee2c137252fe496b8c7e": {
        "Name": "wordpress_db",
        "EndpointID": "8d6deb5cd7482517e7c2cb9ddbd12ffd27d90dec7566fbc0671443acd9bdc0cf",
        "MacAddress": "02:42:ac:15:00:02",
        "IPv4Address": "172.21.0.2/16",
        "IPv6Address": ""
    }
},
```

Step 4: Add WordPress Container to Network

Our next step is adding the WordPress to our network. You can do it with the following command:

```
PS C:\Users\     \MyApp> docker run -dit `
>> --name wordpress-website `
>> -e WORDPRESS_DB_HOST=wordpress_db `
>> -e WORDPRESS_DB_USER=wordpress `
>> -e WORDPRESS_DB_PASSWORD=wordpress `
>> -e WORDPRESS_DB_NAME=wordpressdb `
>> -v ${PWD}/wp-data:/var/www/html `
>> -p 80:80 `
>> --network my_network `
>> wordpress
7ad6815ef6db7855f2e8b289ca80634a62dfd90218475d0387efa6c249ee10ac
```

- **`-e WORDPRESS_DB_HOST=wordpress_db`** → sets the WordPress database host, which matches the name of our MySQL container that we set up in Step 2;
- **`-e WORDPRESS_DB_USER=wordpress`** → sets the WordPress user that we previously set up;

- `-e WORDPRESS_DB_PASSWORD=wordpress` → sets the password for the user;
- `-e WORDPRESS_DB_NAME=wordpressdb` → sets the name of the WordPress database, that we created in Step 2.

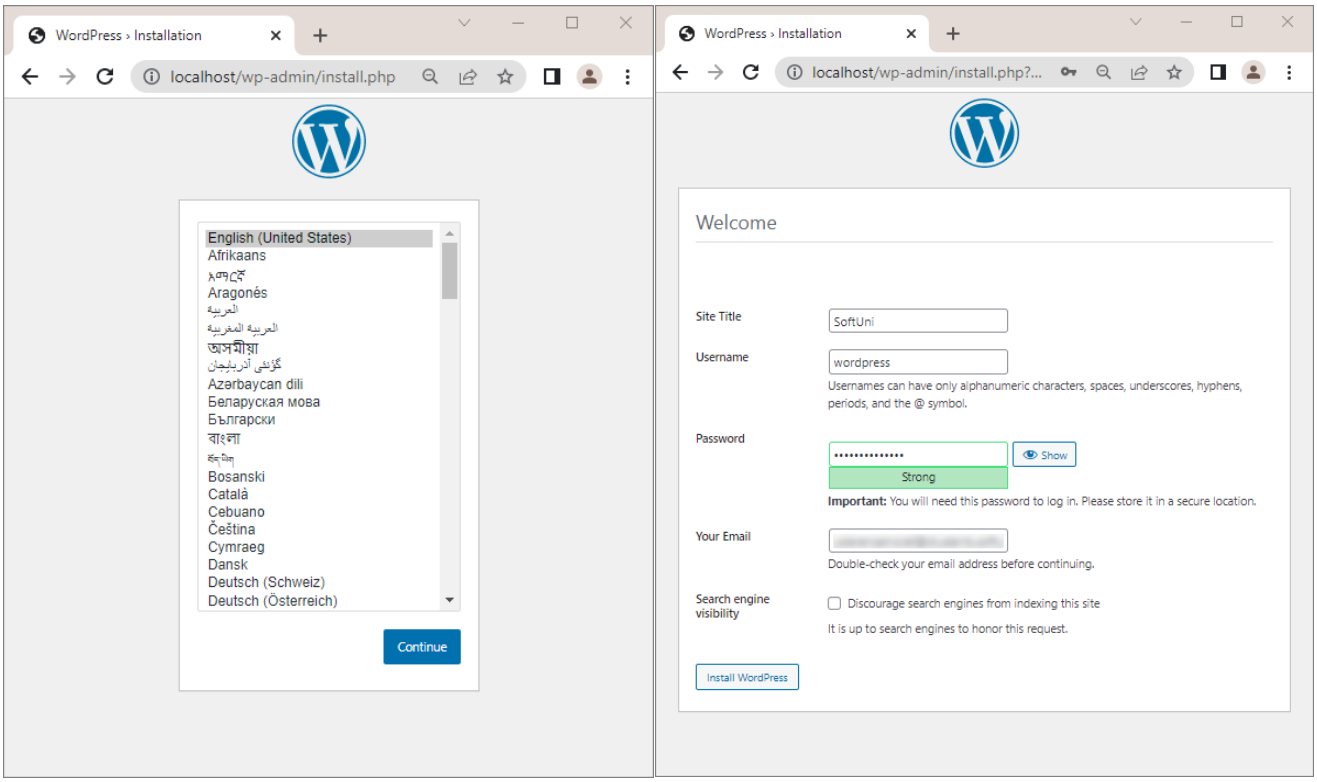
Step 5: Inspect Network

Now, if we execute the command for inspecting our network, we should see that the two containers are attached to it:

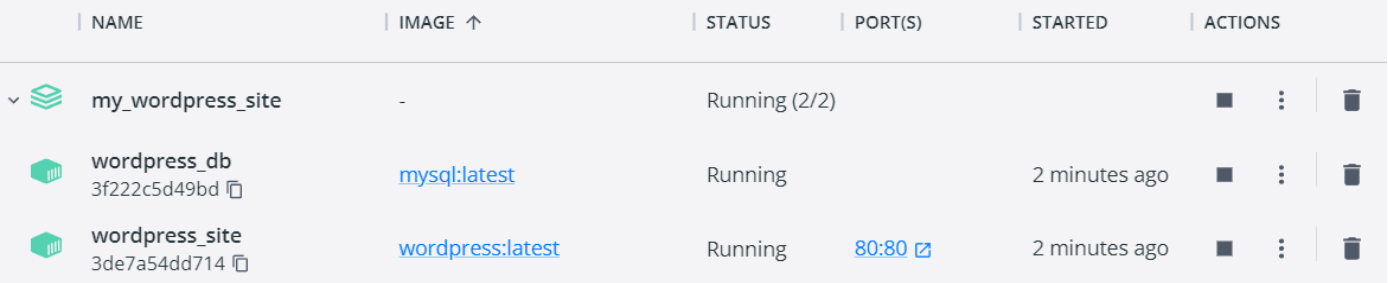

```
"Containers": {
    "7ad6815ef6db7855f2e8b289ca80634a62dfd90218475d0387efa6c249ee10ac": {
        "Name": "wordpress-website",
        "EndpointID": "dd55318b48397db449a0ec3f1fbe198b3483b80c8d2df44574d3155c99d85bb2",
        "MacAddress": "02:42:ac:15:00:03",
        "IPv4Address": "172.21.0.3/16",
        "IPv6Address": ""
    },
    "fd0e62a8b1ee19c06a0125ec831388eb24e95c36bc02ee2c137252fe496b8c7e": {
        "Name": "wordpress_db",
        "EndpointID": "8d6deb5cd7482517e7c2cb9ddbd12ffd27d90dec7566fbc0671443acd9bdc0cf",
        "MacAddress": "02:42:ac:15:00:02",
        "IPv4Address": "172.21.0.2/16",
        "IPv6Address": ""
    }
},
```


Step 6: Run the App

You can **access the WordPress site** on **http://localhost:80** and you will see the **WordPress setup page:**



When you check **Docker Desktop**, you should be able to see the **two containers** combined in a **single network**:



3. WordPress App with MySQL Database: Docker Compose YAML file

Our next task is, using **Docker Composer**, to **create a docker-compose.yaml file** with instructions for **creating the MySQL and WordPress containers** from the previous task**, together** in a single network, called **`my_network`**.

Step 1: Create a Network

Using the **`docker network create`**, create a new network, called **`my_network.`**

Step 2: Create the docker-compose.yml file

First, we have to create a **docker-compose.yml** file in the folder of our app. The docker file should look like this:

docker-compose.yml

```yaml
version: "1.0"

services:
    wordpress_db:
        image: mysql:latest
        command: '--default-authentication-plugin=mysql_native_password'
        volumes:
            - db_data:/var/lib/mysql
        restart: always
        environment:
            - MYSQL_ROOT_PASSWORD=somewordpress
            - MYSQL_DATABASE=wordpress
            - MYSQL_USER=wordpress
            - MYSQL_PASSWORD=wordpress
        expose:
            - 3306
            - 33060
        networks:
            - my_network
    wordpress_site:
        image: wordpress:latest
        volumes:
            - wp_data:/var/www/html
        ports:
            - 80:80
        restart: always
        environment:
            - WORDPRESS_DB_HOST=somewordpress_db
            - WORDPRESS_DB_USER=somewordpress
            - WORDPRESS_DB_PASSWORD=somewordpress
            - WORDPRESS_DB_NAME=somewordpress
        networks:
            - my_network
volumes:
    db_data:
    wp_data:
networks:
    my_network:
```

Step 3: Build and Run the Multi-Container App

Next step is building and running our multi-container app. First, build all of the images with the **`docker-compose build`** command:

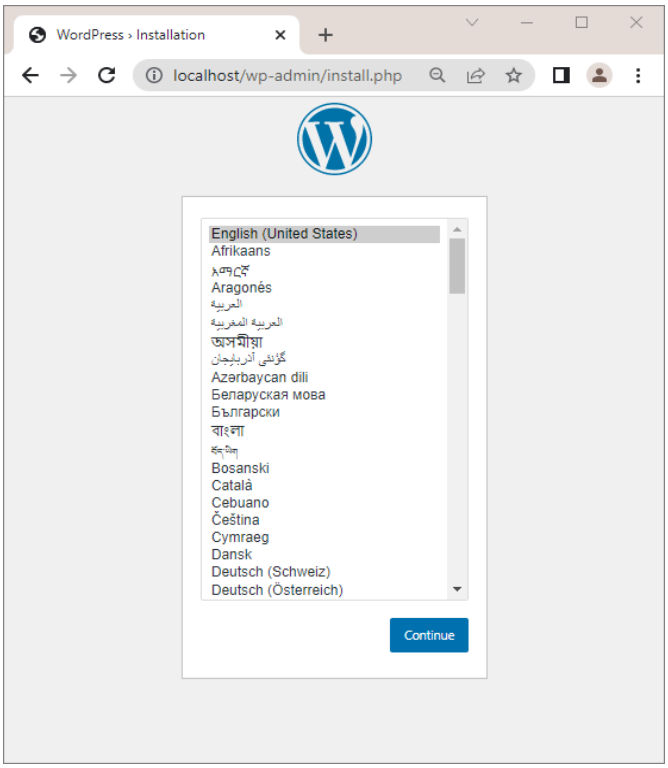
```
PS C:\Users\      \mywebsitewithdb> docker-compose build
```

Then, run the containers with the **`docker-compose up`** or **`docker-compose up -d command`**.

```
PS C:\Users\      \mywebsitewithdb> docker-compose up -d
[+] Running 3/3
 ⠿ Network mywebsitewithdb_my_network          Created
 ⠿ Container mywebsitewithdb-wordpress_site-1  Started
 ⠿ Container mywebsitewithdb-wordpress_db-1    Started
```

Step 4: Run the App

You can **access the WordPress site** on **http://localhost:80** and you will see the **WordPress setup page.**



You should be able to **configure and create your website**, following the guide: