Intro to Docker workshop - worksheet 3

At this point you have a working knowledge of how to manipulate Docker containers and how to write Dockerfiles, allowing you to build your own docker images, allowing you to distribute and deploy services and applications you write to any docker-enabled platform.

Docker-compose builds on top of Docker to allow you to specify and deploy multi container systems quickly. This is very useful since nowadays most useful applications contain multiple components. For example, a web application will usually have at least three components: a application server processing the program logic, a web server dealing with HTTP requests, and a database for data persistence. While you could certainly build a single docker image containing all these components, a better solution is to split them up into individual containers and connect them using a docker network. Docker offers a tool to facilitate this task: Docker-compose.

Installing Docker-compose
Follow the relevant instructions in the installation page.

Building a simple web app with Docker Compose
Let's build a simple todo app with Docker-compose, using Node.js as our application server. We will make use of the express.js framework to simplify things. Start by creating a folder to hold the application code for our todo web app. Inside it create a **package.json** file, which contains the node.js dependencies and project characteristics. The only dependency we need is express.js, so the following very simple file is all we need:

```
{
   "name": "todo-app",
   "version": "0.0.1",
   "main": "server.js",
   "scripts": {
      "start": "nodemon server.js" //used to bring the server up
   },
   "dependencies": {
      "express": "4.16.4",
       "pg": "^7.0",
      "nodemon": "^1.18"
    }
}
```

Now let's define a simple application. We can't persist tasks yet, so let's begin with a simple hello-world application. Create a **index.js** file with the following content. Try to understand what the purpose of this program, which is fairly straightforward.

```
oonst express = require('express')
const app = express()
const port = 3000
app.get('/', (req, res) => res.send('Hello World!'))

app.listen(port, () => console.log(`Application running!`))
```

Now the first version of our application is written. To run it, we could install node.js on our machine, have the node package manager download our dependencies and run the application. However, we know a better way to do this: **let's build a Dockerfile!** The docker registry already contains a node.js image which we can use as a base for our own image. We just need to add our code and have it download our dependencies as we build it. Let's try it: save the following in a file name **Dockerfile.**

```
# this tells docker to base our image on the official node image, version 10
FROM node:10
# tells docker which folder to put our code on and run our commands from
WORKDIR /usr/src/app
#copy the package.json file we wrote to the WORKDIR on the image
COPY package.json .
#tells docker to install our dependencies to the image as it is built
RUN npm install
#copies our application code into the container.
COPY server.js .
#this is the port our application will be served on
EXPOSE 3000
#default command to be run when a container is started with this image.
CMD npm run start
```

To build this image simply use **docker build -t todoapp <path to Dockerfile>**
To check if its working simply start a container in the background and connect it to the correct port: **docker run -td -p 3000:3000 todoapp**. Go to http://localhost:3000/ to see the Hello World message.

Building our stack

What we've seen so far doesn't deviate significantly from what you already knew about Docker. We have a simple service which is tidily containerized inside its own image, which we can build and deploy on any machine which has docker installed. However, most applications of any usefulness use not only a single service but many services. We'd prefer if we could have each single service in our application isolated in its own container. This is where docker-compose

comes in. It lets us declare a series of services which together make up our application. Let's start by defining a single service application, containing only our Hello world app. Create a file named **docker-compose.yml** in your source code folder and define the web service.

```
version: '3' #the version of docker-compose syntax we are using
services: #the list of services which make up our application
 web: #at this moment, it consists of a single service
  build: . #tells docker-compose to use our Dockerfile to build this service
  ports:
   - "3000:3000" #bind the container's port to our networks port, same as the -p
3000:3000 argument
```

Take your time to understand the structure of the docker-compose.yml file with the help of the comments. If you don't understand something, call an instructor over. When you understand what the file does, try it out! The application can be started with the **docker-compose up** command. This command will build a image for each service, create a container from each image and start them. Make sure to stop any containers which are running our application or the service won't be able to bind to the port 3000, since it is already in use.
The hello world should still be there when you check localhost:3000 again.

Adding a new service

Suppose we are expecting to receive thousands of visits every second to our web application. It could be advisable not to use the application service to handle every request. We may instead want to place a more robust web server, such as nginx, between the incoming requests and our application server. This is where docker-compose comes in handy. Let's add a nginx service to our list of services, which so far was composed only of our web service. Additionally, let's make it available on port 80, and remove the port binding from our application. Try it out!

```
version: '3' #the version of docker-compose syntax we are using
services: #the list of services which make up our application
 web: #at this moment, it consists of a single service
  build: . #tells docker-compose to use our Dockerfile to build this service
 nginx:
  image: nginx #we don't need to build our own image, the one available on the docker
registry suffices
  ports:
   - "80:80"
```

Just like that, after running docker-compose up you will have a nginx server ready and listening for requests on port 80. We just need to configure it to redirect our requests to our application. Notice how the application is no longer available on port 3000. Let's write a config file for nginx:

```
http{
        server {
```

```
        listen 80;
        server_name example.com;

        location / {
        proxy_pass http://web:3000;
        }
        }
}

events {
}
```

Save this configuration file as nginx.conf in your directory. Now update the docker-compose.yml file by adding the following line to the nginx service:

```
volumes:
        - ./nginx.conf:/etc/nginx/nginx.conf
```

This allows us to keep using the standard nginx image and just mount the configuration file we want directly to the container. Restart the service and now visiting http://localhost should display our web application in its full glory, with nginx acting as the web server.

Building our Todo app.

Hello World applications are not very exciting. Let's start building our slightly more interesting todo app. There is just one component missing: we need a database on which to store our tasks.

Let's take a look at the code for our todo application. Replace our simple server.js file with the code that follows. Take your time to familiarize yourself with it, if you have doubts, ask.

```
var express = require('express')
const { Pool, Client } = require('pg')
const app = express()
const port = 3000

const config = {
  user: 'pguser',
  password: 'pgpassword',
```

```
  host: 'postgres',
  database: 'db'
}
const pool = new Pool(config)

app.get('/', async (req, res) => {
  const result = await pool.query('SELECT text FROM TASKS');
  const texts = result.rows.map(task => '<li>' + task.text + '</li>');
  const tasks = "<ol>".concat(texts) + '</ol><br>'
  res.send(tasks + `<form method action="/create_task"><input name="text"
type=text placeholder="task text"> <input type="submit" value="Submit"></form>`)
})

app.get('/create_task', async (req, res) => {
  const task_text = req.param('text')
  const result = await pool.query('INSERT INTO TASKS(text) VALUES ($1)',
[task_text])
  res.redirect('/')
})

app.listen(port, () => {
      setTimeout(function() {pool.query(`CREATE TABLE IF NOT EXISTS tasks (
         id SERIAL PRIMARY KEY,
         text VARCHAR(100) NOT NULL
      )`)}, 5000)
  }
)
```

Please note that this is not a scalable way to develop a website and is merely here for illustrative purposes. The two main takeaways from this is that we require that a postgres database be available at the URL "postgres", that we require the credentials for the database to be "pguser" and "pgpassword", and that the database on the server we want to connect is called "db".
Let's use docker-compose to painlessly setup this database in an isolated fashion. Add the following lines to your docker-compose file:

```
 postgres:
  image: postgres
  environment:
```

```
    - POSTGRES_PASSWORD=pgpassword
    - POSTGRES_USER=pguser
    - POSTGRES_DB=db
```

The official postgres makes use of environment variables to setup the credentials and the name of the default database. You can specify any environment variables you want under the “environment” key. In this case we provide the credentials and database name we want to use in our server.

Now let’s run our updated application. Start by refreshing our “web” image with the new server code by running “docker-compose build”. Kill any running containers by running “docker-compose down”. Then restart our application by running “docker-compose up”. When you visit http:localhost you should be greeted by the new application.

Exercise

Investigate how you might be able to use volumes with docker-compose to setup our project so that any changes performed to the “server.js” file are automatically reflected inside our docker container, so that we do not need to take down the stack and perform a “docker-compose build” command each time we update our code.