ALPEN-ADRIA UNIVERSITÄT KLAGENFURT

Docker Containers Tutorial part2

Narges Mehran, MSc.

Current Topics in Distributed Systems:
Internet of Things and Cloud Computing,

SS2021

26.05.2021

Docker Compose

Docker Compose is a tool for defining and running multi-container Docker applications on one physical or virtual machine.

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.

Docker Compose (cont.)

Using Compose is basically a three-step process:

- Define your app's environment with a *Dockerfile* so it can be reproduced anywhere.
- Define the services that make up your application in *docker-compose.yml* so they can be run together in an isolated environment.
- Run *docker-compose up*
- Then *Compose* starts and runs your entire application.

Overview of Docker Compose

Compose has commands for managing the whole lifecycle of your application:

- Start, stop, and rebuild services
- View the status of running services
- Stream the log output of running services
- Run a one-off command on a service

The *Compose* file is a YAML file defining

- services,
- networks,
- volumes.

The default path for a *Compose* file is *./docker-compose.yml*.
Tip: you can use either a *.yml* or *.yaml* extension for this file.

1st example

Python/Flask application and Redis

You can build a simple Python web application running on Docker Compose.

The application uses the Flask framework and maintains a hit counter in Redis:

- Redis (Remote Dictionary Server)
  - is an in-memory data structure project implementing a distributed, in-memory key-value database with optional durability.
- Flask is a micro-web framework written in Python

Python/Flask application and Redis

This Compose file defines two services: web and Redis.

- **Web service,**
  uses an image built from the Dockerfile in the current directory. It then binds the container and the host machine to the exposed port, 5000. This example service uses the default port for the Flask web server, 5000.

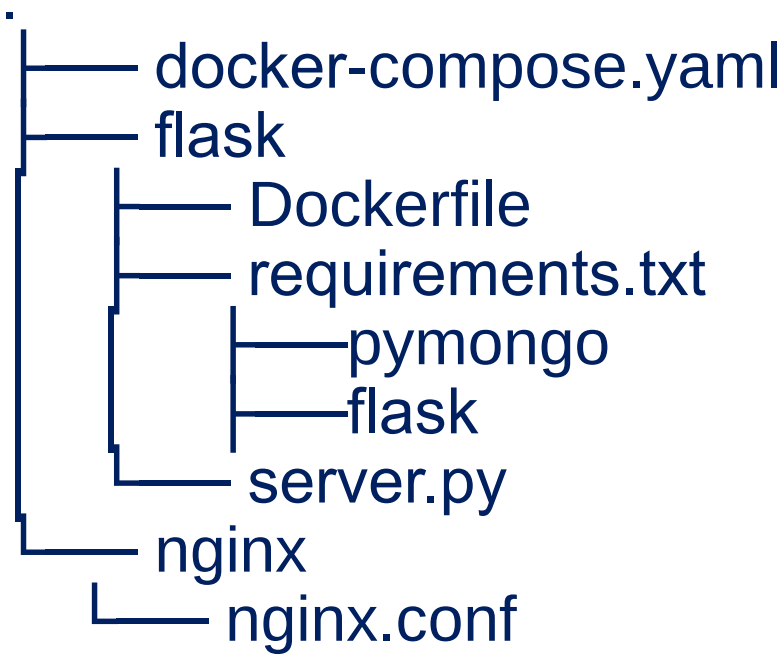
- **Redis service,**
  uses a public Redis image pulled from the Docker Hub registry.

```
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
  redis:
    image: "redis:alpine"
```

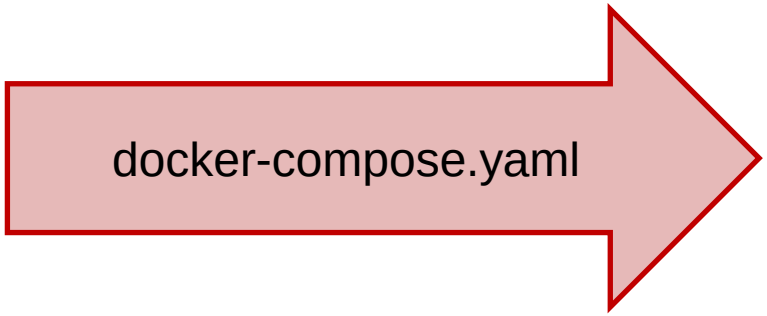
2nd example

Python/Flask application with Nginx proxy and a Mongo database

nginx-flask-mongo project structure:



Briefly-explained YAML file




```
services:
  web:
    build: app
    ports:
    - 80:80
  backend:
    build: flask
    ...
  mongo:
    image: mongo
```

Complete YAML file

```yaml
version: "3.7"
services:
  web:
    image: nginx
    volumes:
      - ./nginx/nginx.conf:/tmp/nginx.conf
    environment:
      - FLASK_SERVER_ADDR=backend:9091
    command: /bin/bash -c "envsubst < /tmp/nginx.conf > /etc/nginx/conf.d/default.conf && nginx -g 'daemon off;'"
    ports:
      - 8081:8081
    depends_on:
      - backend
  backend:
    build: flask
    environment:
      - FLASK_SERVER_PORT=9091
    volumes:
      - ./flask:/src
    depends_on:
      -  mongo
  mongo:
    image: mongo
```

Complete YAML file

```yaml
version: "3.7"
services:
  web:
    image: nginx
    volumes:
      - ./nginx/nginx.conf:/tmp/nginx.conf
    environment:
      - FLASK_SERVER_ADDR=backend:9091
    command: /bin/bash -c "envsubst < /tmp/nginx.conf > /etc/nginx/conf.d/default.conf && nginx -g 'daemon off;'"
    ports:
      - 8081:8081
    depends_on:
      - backend
  backend:
    build: flask
    environment:
      - FLASK_SERVER_PORT=9091
    volumes:
      - ./flask:/src
    depends_on:
      -  mongo
  mongo:
    image: mongo
```

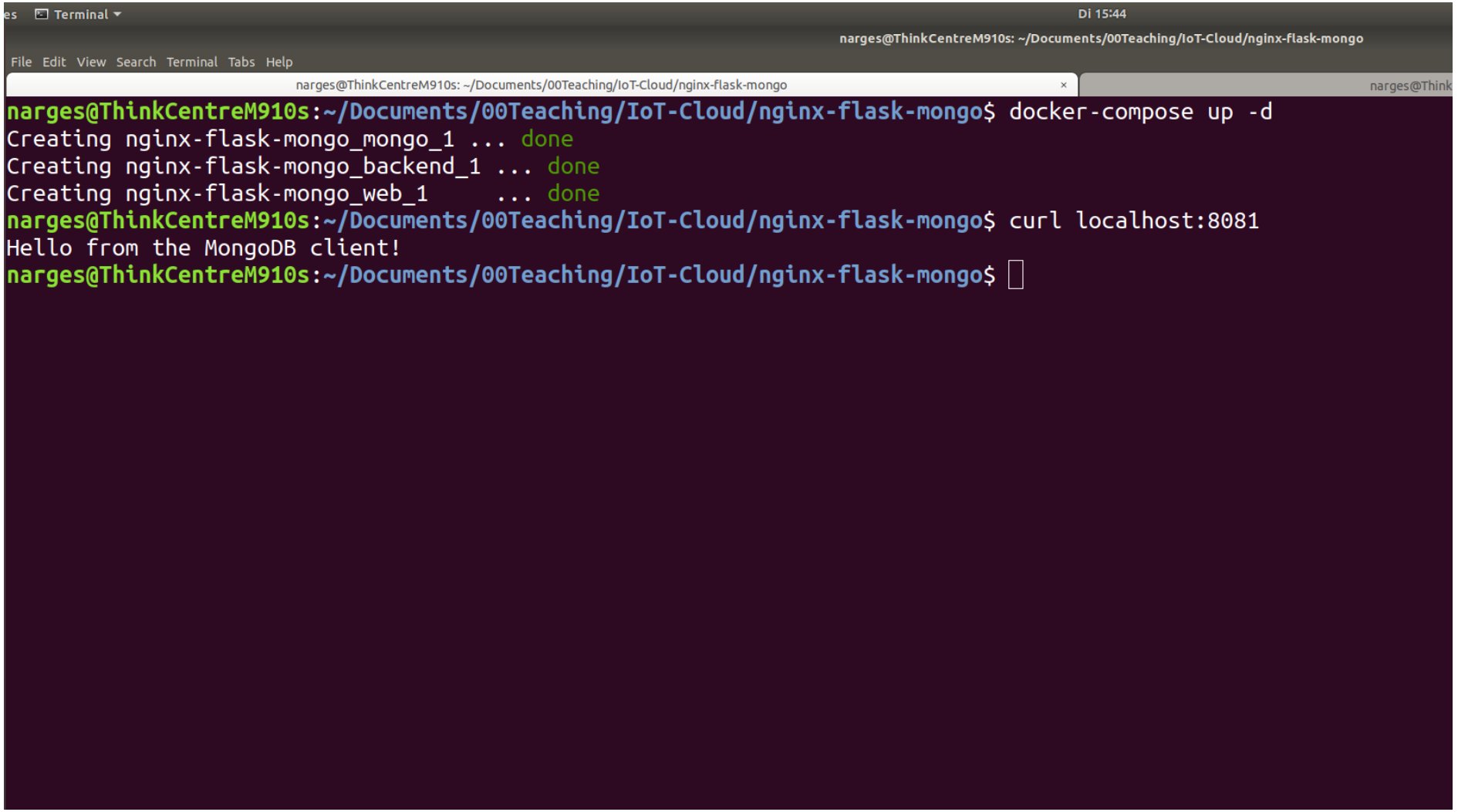
Complete YAML file

```yaml
version: "3.7"
services:
  web:
    image: nginx
    volumes:
      - ./nginx/nginx.conf:/tmp/nginx.conf
    environment:
      - FLASK_SERVER_ADDR=backend:9091
    command: /bin/bash -c "envsubst < /tmp/nginx.conf > /etc/nginx/conf.d/default.conf && nginx -g 'daemon off;'"
    ports:
      - 8081:8081
    depends_on:
      - backend
  backend:
    build: flask
    environment:
      - FLASK_SERVER_PORT=9091
    volumes:
      - ./flask:/src
    depends_on:
      -  mongo
  mongo:
    image: mongo
```

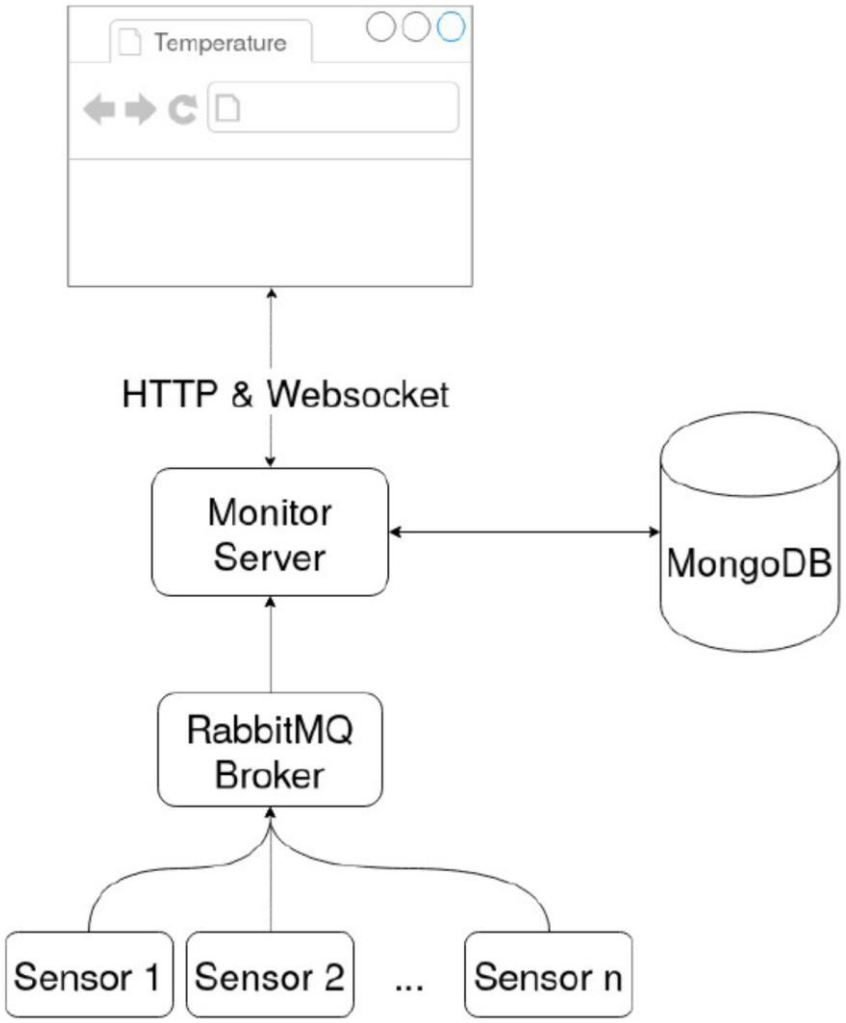
server.py

```
import os

from flask import Flask
from pymongo import MongoClient

app = Flask(__name__)

client = MongoClient("mongo:27017")

@app.route('/') #route() decorator tells Flask what URL should trigger our function
def todo():
    try:
        client.admin.command('ismaster')
    except:
        return "Server not available"
    return "Hello from the MongoDB client!\n"


if __name__ == "__main__":
    app.run(host='0.0.0.0', port=os.environ.get("FLASK_SERVER_PORT", 9091), debug=True)
```

Execution and an output



Temperature sensor monitoring modules



Services:

MongoDB:

MongoDB stores all the sensor data found in the thermal solar plant github repository as documents.

RabbitMQ Broker:

RabbitMQ is used to communicate between the monitoring app backend and the individual sensors.

Sensor Publishers:

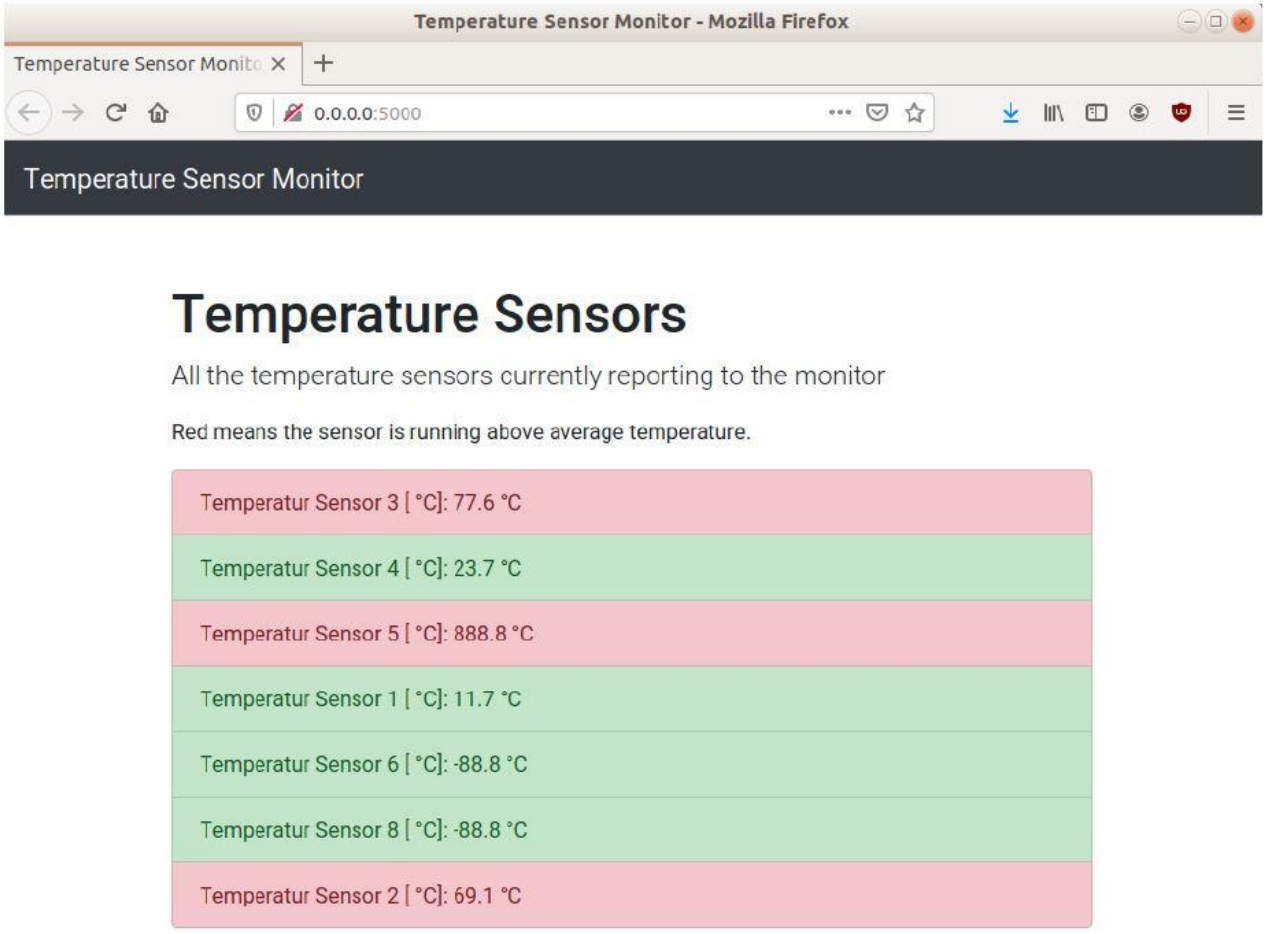
Sensor publishers publish their values to the RabbitMQ channel.

Monitor Server:

The monitor server serves a webpage that uses JavaScript and Socket.io to connect via websocket to the monitor server. The monitor server subscribes via RabbitMQ to the sensor_data topic and determines whether the temperature is still okay or not. Then emits an event via socket.io to update the list on the web-page.

Monitoring Page:

The monitoring page connects via websocket to the server and waits for update events, which contains an ID as well as the sensor name and temperature and information whether it has passed the threshold or not. If It has passed the threshold the entry is colored red.



How to use:

Run `./initialize.bash` to download the necessary data from GitHub and import the data.

Run `docker-compose up --force-recreate` to run the app. The monitoring interface will be served at `http://localhost:5000`

```yaml
version: '3.7'
services:
  mongodb-server:
    image: mongo:4.2.6-bionic
    ports:
      - 27017:27017
    volumes:
      - ./mongo-data/:/data/db/
    networks:
      - mongodb-network

  rabbitmq-broker:
    image: rabbitmq:3-management
    networks:
        - rabbitmq-network

  sensor-monitor:
    build:
      context: ./sensor-python/
      dockerfile: monitoring.Dockerfile
    depends_on:
      - mongodb-server
      - rabbitmq-broker
    ports:
      - 5000:5000
    command:
        - '--host=mongodb-server'
        - '--port=27017'
        - '--username=root'
        - '--password=example'
        - '--broker_host=rabbitmq-broker'
    networks:
      - mongodb-network
      - rabbitmq-network

  # Sensors starting here...
  sensor-publisher-1:
    build:
        context: ./sensor-python/
        dockerfile: sensor.Dockerfile
    depends_on:
        - mongodb-server
        - rabbitmq-broker
    command:
        - '--host=mongodb-server'
        - '--port=27017'
        - '--username=root'
        - '--password=example'
        - '--broker_host=rabbitmq-broker'
        - '--rate=10'
        - '--sensor_key=Temperatur Sensor 1 [ °C]'
    networks:
        - mongodb-network
        - rabbitmq-network
networks:
  mongodb-network:
  rabbitmq-network:
```

3rd example

Overlay: The overlay driver creates a named network across multiple nodes.

Deploy: Setting the configuration related to the deployment and execution of services. This just takes effect when deploying on multiple nodes with docker stack deployment and is ignored by *docker-compose up* and *docker-compose run*.

```
version: "3.8"

services:
  wordpress:
    image: wordpress
    ports:
      - "8080:80"
    networks:
      - overlay
    deploy:
      mode: replicated
      replicas: 2
      endpoint_mode: vip

  mysql:
    image: mysql
    volumes:
       - db-data:/var/lib/mysql/data
    networks:
       - overlay
    deploy:
      mode: replicated
      replicas: 2
      endpoint_mode: dnsrr
```

Deploy:

replicas: 2 - If a service is ***replicated***, specify the number of containers that should be running at any given time

endpoint_mode: vip - Docker assigns the service a virtual IP (VIP) that acts as the front end for clients to reach the service on a network. Docker routes requests between the client and available worker nodes for the service, without client knowledge of how many nodes are participating in the service or their IP addresses or ports. (This is the default.)

endpoint_mode: dnsrr - DNS round-robin (DNSRR) service discovery does not use a single virtual IP. Docker sets up DNS entries for the service such that a DNS query for the service name returns a list of IP addresses, and the client connects directly to one of these. DNS round-robin is useful in cases where you want to use your own load balancer, or for Hybrid Windows and Linux applications.

```
version: "3.8"

services:
  wordpress:
    image: wordpress
    ports:
      - "8080:80"
    networks:
      - overlay
    deploy:
      mode: replicated
      replicas: 2
      endpoint_mode: vip

  mysql:
    image: mysql
    volumes:
       - db-data:/var/lib/mysql/data
    networks:
       - overlay
    deploy:
      mode: replicated
      replicas: 2
      endpoint_mode: dnsrr
```

Install Docker Compose

You can run Compose on macOS, Windows, and 64-bit Linux:

- ✓ pip install docker-compose
- ✓ https://docs.docker.com/compose/install/#install-using-pip

Assignment 10

Option 1:

Prepare a multi-container application by the following containers:

- One container for publishing to IBM Watson IoT platform
- On container for subscribing to the broker

Assignment 10

Option 2:

Prepare a multicontainer application by the following containers:

- One container: Python/Flask application for SysAdmin to connect (Extra point for authentication of SysAdmin),
- Another container: to connect to MongoDB database (you have the data from your previous assignment).
- Process the sensed data and print it for SysAdmin.