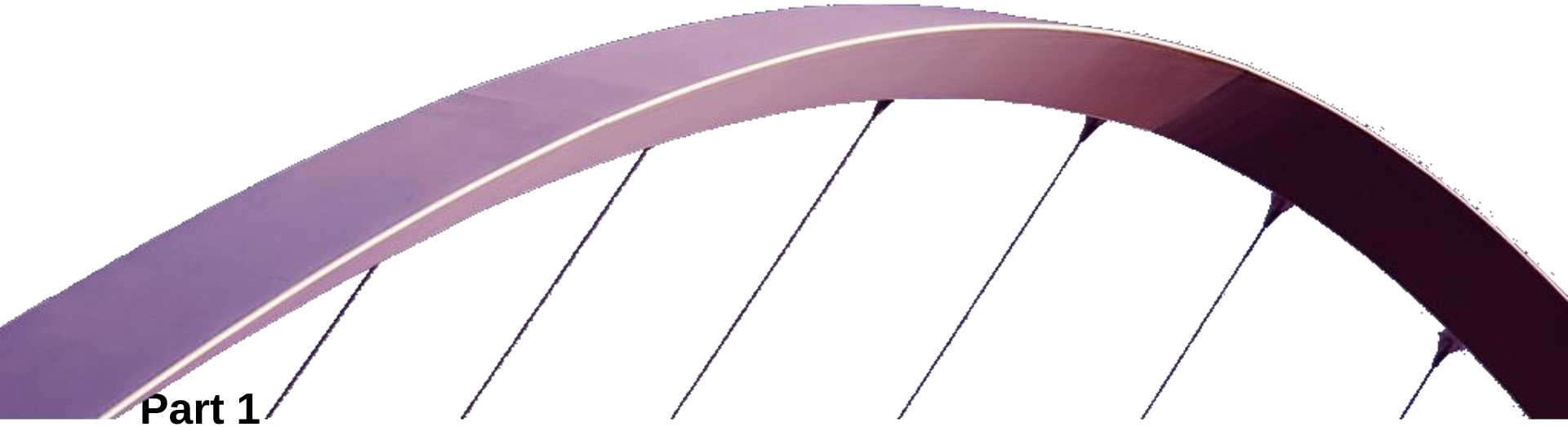

Northeastern University


# CSYE 7220 DevOps
# Lecture 4 Docker Networks & Docker Compose

***23 May 2024***



Part 1

# DOCKER NETWORKS

# Creating a network for containers to share

- Communication issues? Create a network!
  - `docker network create --driver bridge sa_network`
  - `docker network inspect sa_network`
- Create a network with a specific subnet and gateway:
  - `docker network create --driver=bridge --subnet=192.168.2.0/24 --gateway=192.168.2.10 sa_network`
- Connect a container to a network
  - `docker run --network= sa_network -itd --name=sa-fe-container sa-fe-image`
  - Any other container you create on this network would be able to automatically connect to one another
- To remove a user-defined bridge network
  - If containers are connected to the network, disconnect them first: `docker network disconnect network sa-fe-container`
  - `docker network rm sa-network`
- https://docs.docker.com/network/bridge/

# A note about networks

- A network is *not* an IP!
- A network is like a city with many houses, *where each house has an IP*
- We create a network with docker network so that all the containers living in the network can talk to each other

# Containers communicating *through the host*

*For Docker Desktop, replace http://192.168.99.100 with http://localhost*

- **sa-logic:**
  - `docker build -t nsalo .`
  - `docker run -p 5050:5000 nsalo`
  - `http://192.168.99.100:5050/testHealth` works?
  - `http://192.168.99.100:8080/testComms` works?

- **sa-webapp:**
  - `mvn install`
  - `docker build -t nsawa .`
  - `docker run -p 8080:8080 -e "SA_LOGIC_API_URL=http://192.168.99.100:5050" nsawa`
  - `http://192.168.99.100:8080/testHealth` works?
  - `http://192.168.99.100:8080/testComms` works?

- **sa-frontend:**
  - `npm install`
  - `npm run build`
  - `docker build -t nsafe .`
  - `docker run -p 8085:80 nsafe`
  - **Does front end Send button work?: http://192.168.99.100:8085**

# Containers communicating *through a bridge network*

- **sa-logic:**
  - `docker network create --driver bridge sanet`
  - `docker build -t nsalo .`
  - `docker run -p 5050:5000 --network=sanet nsalo`
  - `docker network inspect sanet`
    - Get the IP of the container: `a.b.c.d`
  - `http://a.b.c.d:5000/testHealth` works?
  - How about `http://192.168.99.100:5050/testComms` ?

- **sa-webapp:**
  - `mvn install`
  - `docker build -t nsawa .`
  - `docker run -p 8080:8080 --network=sanet -e "SA_LOGIC_API_URL=http://a.b.c.d:5000" nsawa`
  - `http://p.q.r.s:8080/testHealth` works?
  - How about `http://192.168.99.100:8080/testComms` ?

- **sa-frontend:**
  - `npm install && npm run build`
  - `docker build -t nsafe .`
  - `docker run -p 8085:80 --network=sanet nsafe`
  - `docker network inspect`
  - Does the front end *Send* button work?: `http://192.168.99.100:8085`

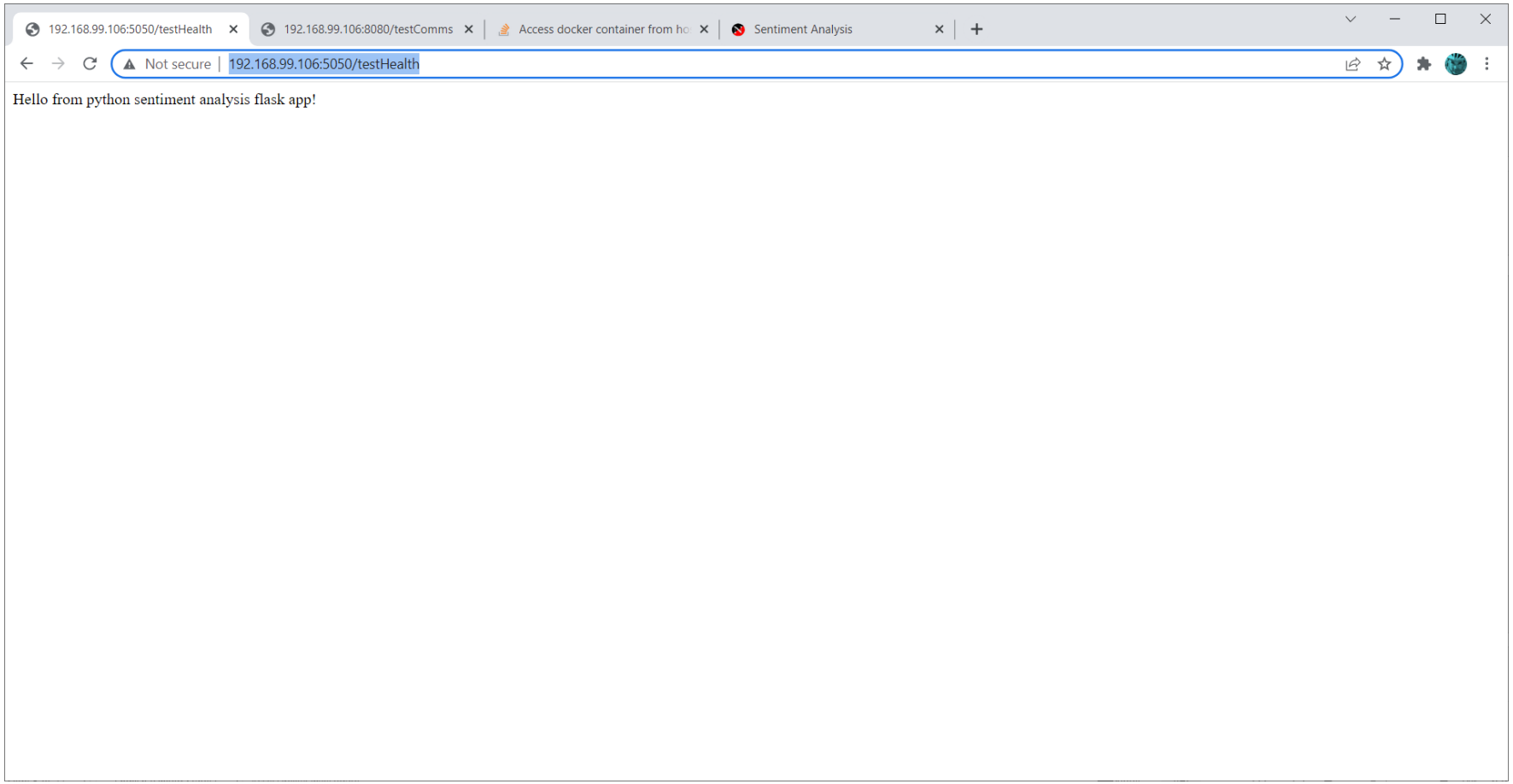
# Results

- Don’t forget to replace http://192.168.99.100 with http://localhost if you’re running on Docker Desktop!
- You should find that you cannot access any service endpoints at `http://a.b.c.d:5000/` or `http://p.q.r.s:8080/`, but your Send button at http://192.168.99.100:8085 should still work (without having to redirect through your host)!
  - But keep accessing your service endpoint UI like /testHealth and /testComms through your host and port redirection instead!

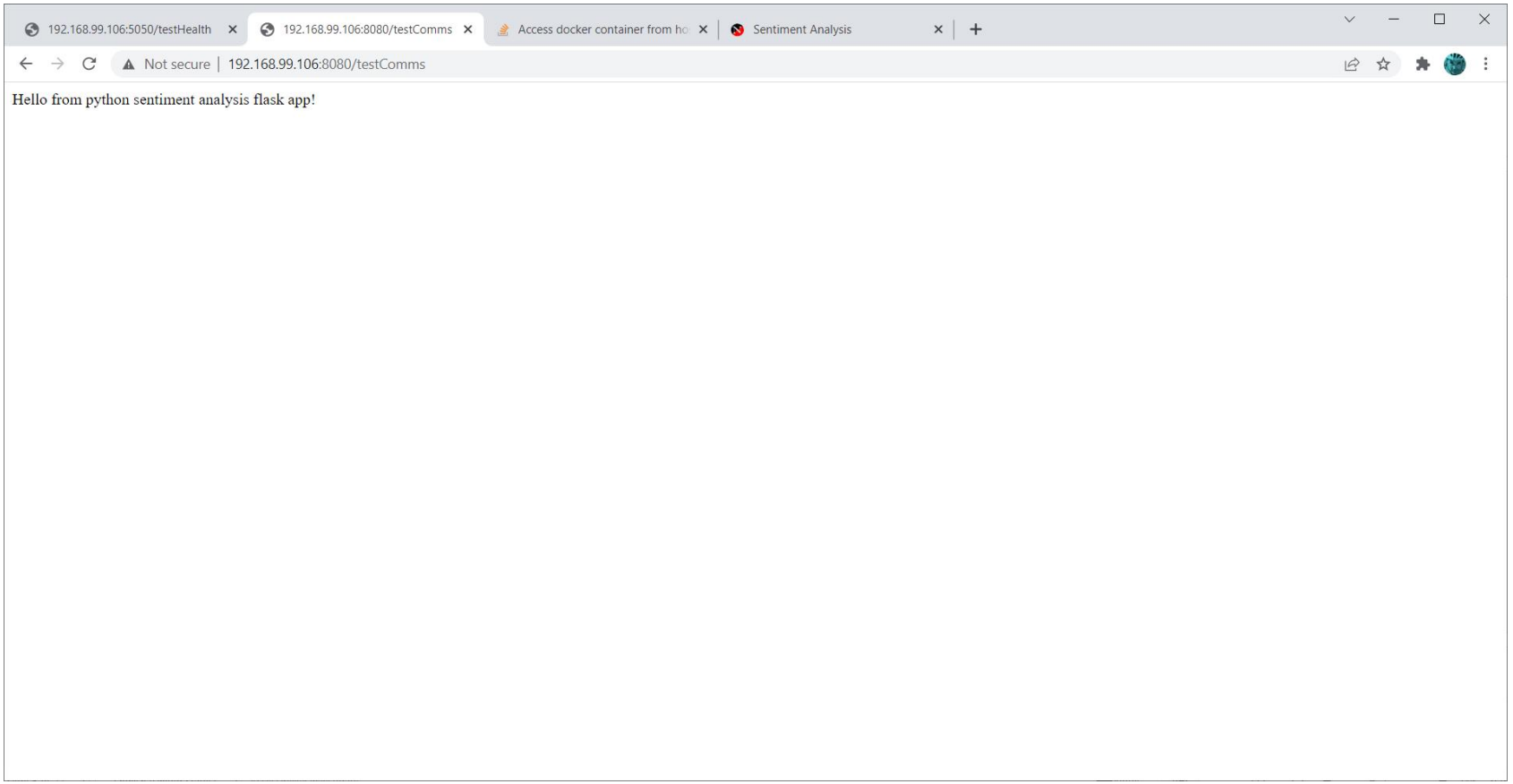
# About networks

- In our native (non containerized) runs, we were using the *host network* as the network for all 3 apps to connect on
  - That's your laptop's http://localhost (http://192.168.99.100 for me)
- In your docker attempts, you were using the host network (leveraging the –p port mapping command) to try to connect the containers
  - But maybe that didn't work because you actually do not know what IPs the containers are running at
  - It works for me, because I run them all on a linux VM that I've created using docker toolbox (hardware virtualization)
  - If you use Docker Desktop and do not create a new VM, you use your own OS to host your containers (container virtualization)
  - So you have to create a docker network to allow them to live in the same city (network) so they can talk to each other directly (not through the host computer)
- The `–p` port mappings are not really required anymore!
  - Only used to test the `/testHealth` and `/testComms` endpoints!
  - We do need it for the front end though, to access the UI!

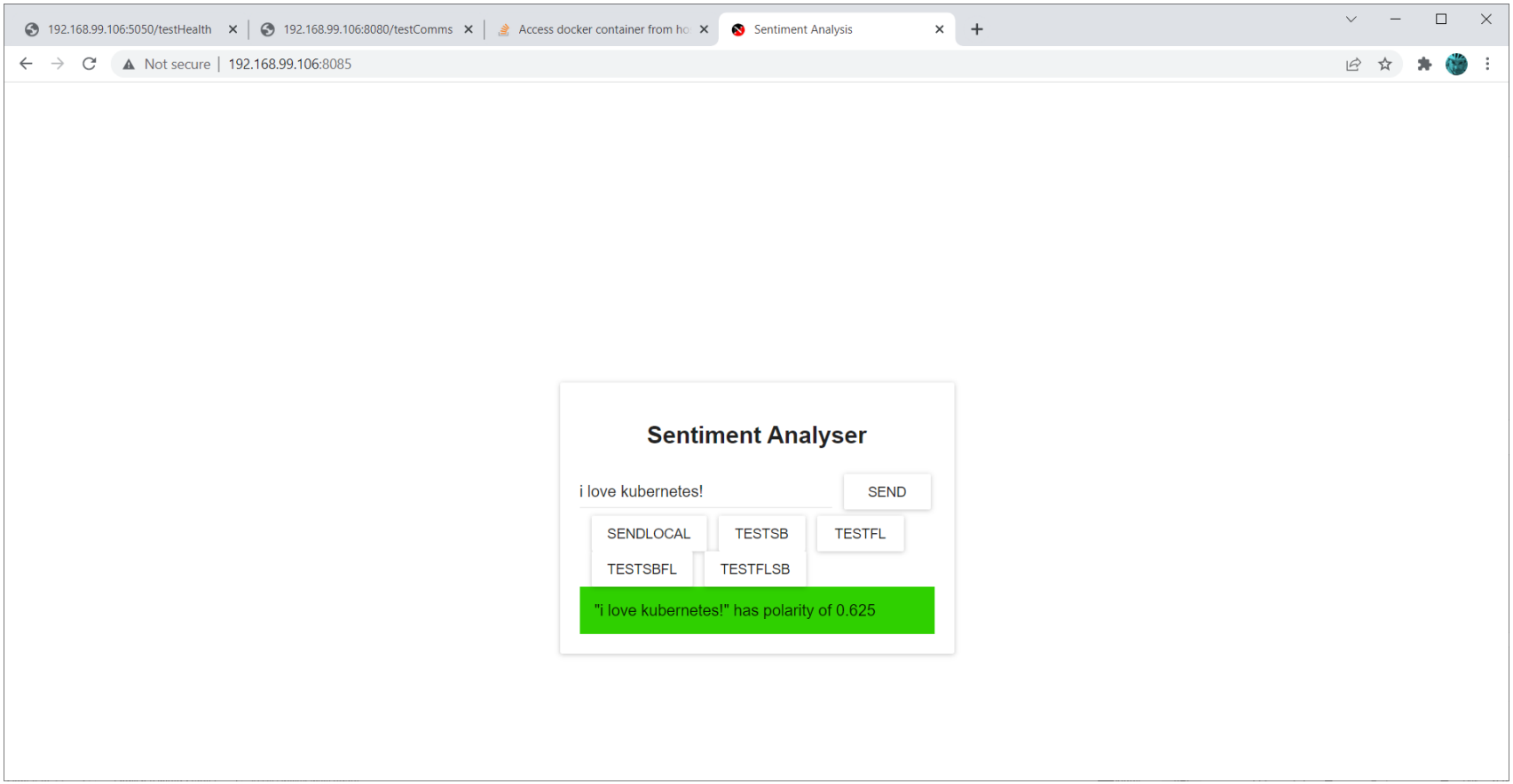
# `/testHealth` endpoints (192.168.99.106 host)



# `/testComms` endpoints (192.168.99.106 host)



# Front end



Part 2

# DOCKER-COMPOSE

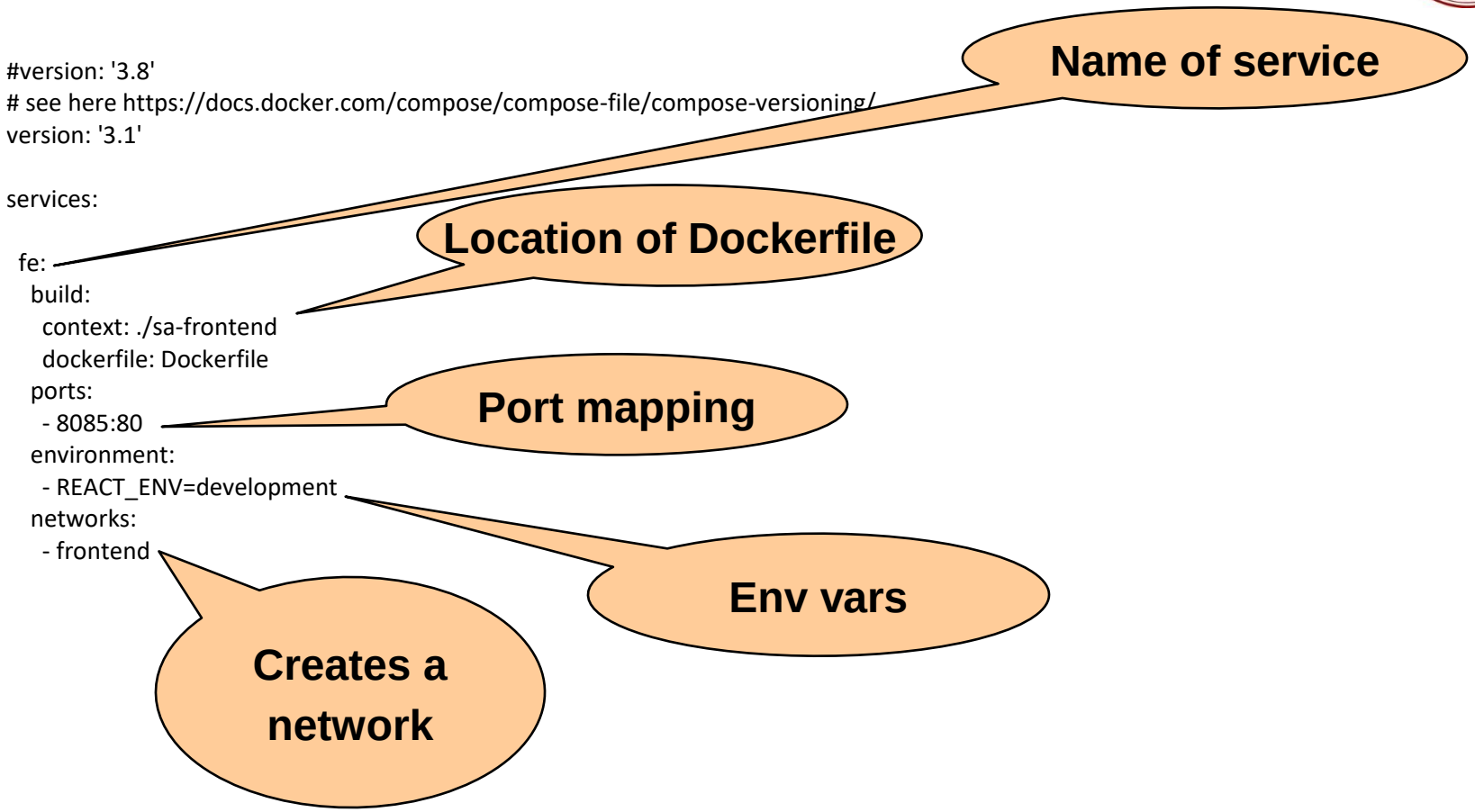
# Docker Compose

- **`Docker-compose`** is a tool for defining and running multi-container Docker applications
- With Compose, we can create a YAML file to define *all* our microservices and with a single command, and spin everything up or tear it all down
- Make sure you have docker-compose in your installation
  - **`docker-compose version`**

# docker-compose

- 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
- Also, ***easy*** to create **`volumes`** (map folders from host to container) and **`networks`** (to bridge containers)
- 3 steps:
  - **Write a `Dockerfile` for each μservice**
  - **Define all μservices that make up your ecosystem, in `docker-compose.yml`**
    - **Put that file outside of each service's folder**
  - **Run `docker compose up` and Docker starts and runs all your μservices**
- https://docs.docker.com/compose/

# `docker-compose.yml` (outside of services folders)



# docker-compose.yml (continued)

```
#continuing
 lo:
  build:
   context: ./sa-logic
   dockerfile: Dockerfile
  ports:
   - 8080:8080
  environment:
   - FLASK_ENV=development
  networks:
   - backend

 wa:
  build:
   context: ./sa-webapp
   dockerfile: Dockerfile
  ports:
   - 8080:8080
  environment:
   - SA_LOGIC_API_URL=http://localhost:5000
   - WA_NETWORK=backend
  depends_on:
   - lo
  # specifies frontend and backend as the networks the wa service will have access to
  networks:
   - frontend
   - backend
```

Name of service

Name of service

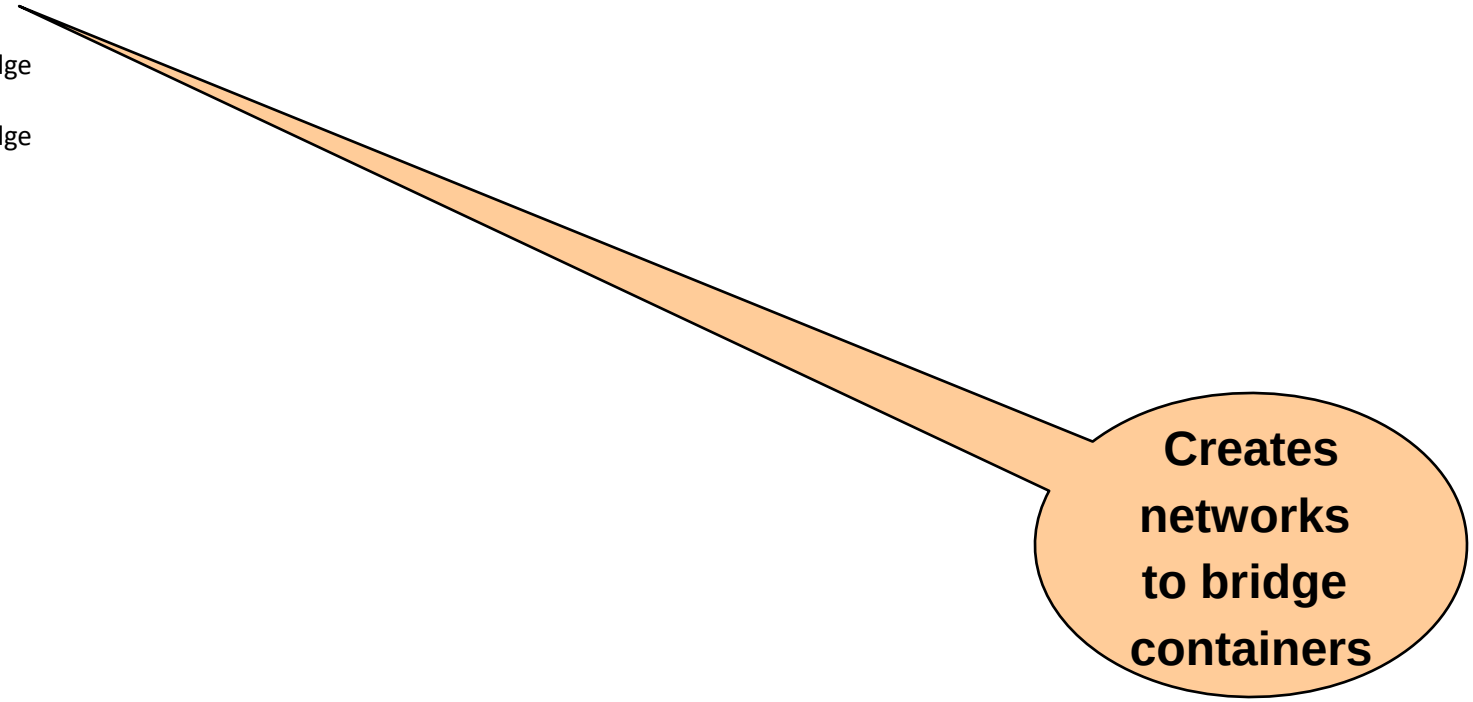
Network specified as env var so that source code could use it

networks that service wa has access to

# docker-compose.yml (finish)

```
#continuing

# bridge networks to allow the containers to communicate with each other
networks:
  frontend:
    driver: bridge
  backend:
    driver: bridge
```



# Comment out env vars from `fe` Dockerfile


```
FROM openjdk:8-jdk-alpine
# Environment Variable that defines the endpoint of sentiment-analysis python api:
#ENV SA_LOGIC_API_URL http://localhost:5000
ADD target/sentiment-analysis-web-0.0.1-SNAPSHOT.jar /
EXPOSE 8080
CMD ["java", "-jar", "sentiment-analysis-web-0.0.1-SNAPSHOT.jar", "--sa.logic.api.url=${SA_LOGIC_API_URL}"]
```

# Build μservice binaries first!

- sa-logic:
  - Nothing to do!
- sa-webapp:
  - `mvn install`
- sa-frontend:
  - `npm install`
  - `npm run build`

# docker-compose in action: Build step

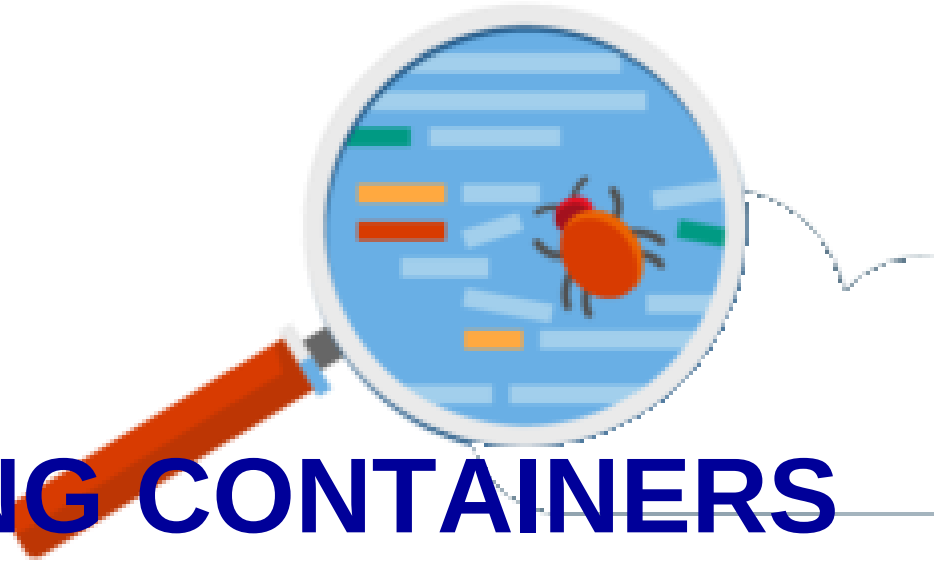
- **Build *all container* images in one step!**
  - `docker-compose build`

# docker-compose in action: Run step

- Run the containers once the build is complete:
  - `docker-compose up -d`
- `docker container ls`
- Navigate to http://localhost:8085 in your browser
  - (or http://192.168.99.100:8085)

# Docker teardown

- To stop the containers:
  - `docker-compose stop`
- To bring down the containers:
  - `docker-compose down`
- Want to force a build?
  - `docker-compose build --no-cache`
- Remove images:
  - `docker rmi $(docker images -q)`



Part 3

# DEBUGGING CONTAINERS

# Debugging containers

- The docker exec command will let you run arbitrary commands inside an existing container
  - `docker exec -it <container_name> bash`
  - `docker exec -it <container_name> sh`
  - `docker exec -it <container_name> /bin/sh`
- To get into the container interactively
  - `docker run -it --entrypoint /bin/bash <container_name>`

# More on debugging containers


- **View stdout history with the logs command**
  - **`docker logs <container_name>`**
  - This history is available even after the container exits, as long as its file system is still present on disk (until it is removed with **docker rm**). The data is stored in a json file buried under /var/lib/docker
- **Stream stdout with the attach command**
  - **`docker attach <container_name>`**
  - By default this command attaches stdin and proxies signals to the remote process. Options are available to control both of these behaviors. To detach from the process use the default `ctrl-p ctrl-q` sequence
- **Execute arbitrary commands with exec**
  - **`docker exec <container_name> cat /var/log/test.log`**
- interactive shell in the container
  - **`docker exec -it <container_name> /bin/sh`**

# Extreme: Debug a container from another


- Create a debug container with strace
  - ```
    FROM alpine
    RUN apk update && apk add strace
    CMD ["strace", "-p", "1"]
    ```
- Build the container
  - ```
    docker build -t strace .
    ```
- Run strace container in the same pid and network namespace
  - ```
    docker run -t --pid=container:baadf00d \
    --net=container: baadf00d \
    --cap-add sys_admin \
    --cap-add sys_ptrace \
    strace
    ```
  - This attached strace to the baadf00d process and follows it as it executes
    ```

# Root filesystem


- To get to the root filesystem of the remote container, use the alpine image and launch a shell, in the same pid and network namespace
  - ```
    docker run -it --pid=container:baadf00d \
    --net=container:baadf00d \
    --cap-add sys_admin \
    alpine sh
    ```
- **With this container attached to the original we can do more debugging**
  - **You can still debug the network but make sure you use localhost because your new sh process is running in the same network namespace**
  - ```
    apk update && apk add curl lsof
    curl localhost:2015
    ```
  - ```
    lsof -i TCP
    ```
  - All standard debugging tools should work from this 2nd container without tainting the original container
    ```


CLASS
DISMISSED
iStockphoto