

# Applicazione demo

1

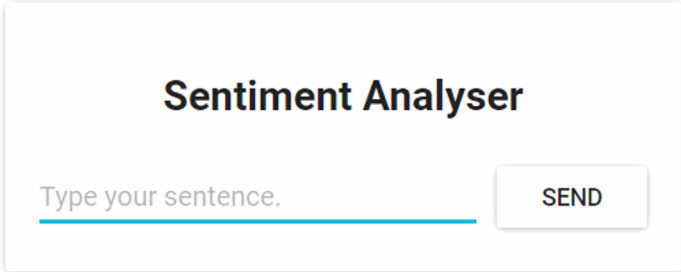
- Scaricare file zip
- E' presente una cartella "sentiment" con il codice della nostra applicazione demo

1

# Applicazione demo

2

L'applicazione ha solo una funzionalità: calcolare il sentimento di una frase



The screenshot shows a web application titled "Sentiment Analyser". Below the title is a text input field with the placeholder text "Type your sentence." and a "SEND" button to its right.

2

# Componenti dell'applicazione

3

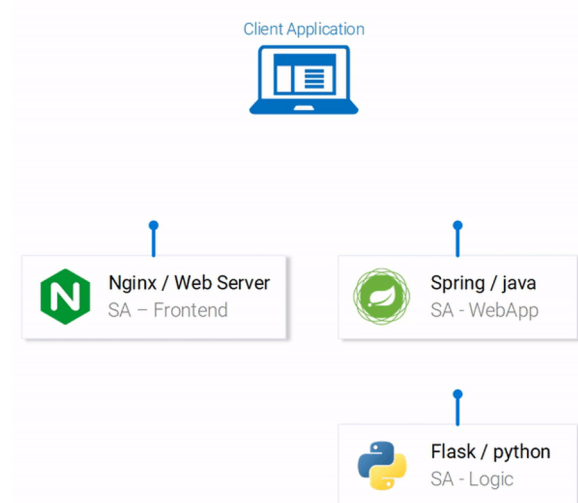
L'applicazione si compone di tre componenti, ognuno con una funzionalità specifica:

- **Frontend:** un server nginx che renderizza i file ReactJs statici
- **WebApp:** una Java Web Application che gestisce le richieste dal frontend
- **Logica:** un'applicazione python che esegue la Sentiment Analysis

3

# Componenti dell'applicazione

4



4

## Costruire i containers per ogni componente

5

Andremo a costruire le immagini di ogni componente

first: terminal > cd frontend > docker build . >

5

## Frontend

6

Per la creazione del frontend abbiamo bisogno di:

- installare due tools **nodejs** e **npm**
- installare le dipendenze con npm
- creare i file statici con il comando `npm run build`
- avviare il web server
- copiare i file nella cartella html del webserver

6

## Frontend docker image

7

Per la costruzione dell'immagine docker, dobbiamo scrivere un **Dockerfile** che si occuperà di

- generare i file
- copiare i file generati nella cartella del webserver

7

## Frontend docker image

8

Per la costruzione dell'immagine docker, dobbiamo scrivere i precedenti passi in un **Dockerfile**:

```
FROM nginx
RUN apt-get update && apt-get upgrade -y
RUN apt-get install -y build-essential
RUN curl -fsSL https://deb.nodesource.com/setup_lts.x | bash -
RUN apt-get install -y nodejs
COPY . .
RUN npm install
RUN npm run build
RUN cp -r /build/. /usr/share/nginx/html
RUN rm -rf node_modules
```

8

## Frontend docker image

9

Lanciare i comandi per la creazione dell'immagine docker che poi inviamo al registry:

```
docker login -u="$DOCKER_USER_ID" -p="$DOCKER_PASSWORD"  
docker build -t $DOCKER_USER_ID/frontend .  
docker push $DOCKER_USER_ID/frontend
```

9

## Frontend docker image

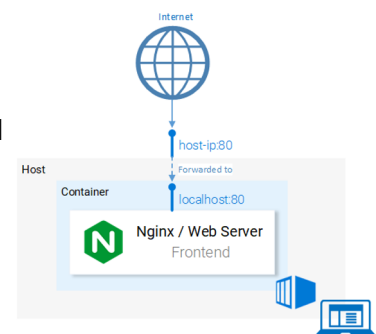
10

Verifichiamo che sia andato tutto bene facendo partire un container con l'immagine appena creata:

```
docker run -d -p 80:80 $DOCKER_USER_ID/frontend
```

Apriamo il browser alla pagina: localhost

**Sentiment Analyser**



10

# Webapp

11

Per la creazione della Webapp abbiamo bisogno di:

- installare il tool **mvn**
- installare le dipendenze e creare il file .jar con mvn
- avviare il file .jar

oss: we can't build the 3 app separately instead we use docker-compose file to help us keep track of our distributed system:  
terminal> sentiment > docker-compose -f ./docker-compose\_LOCAL.yml up

11

# Webapp docker image

12

Per la costruzione dell'immagine docker della Webapp dobbiamo creare un altro **Dockerfile**:

```
FROM maven:3.8.1-openjdk-11
# Environment Variable that defines the endpoint of sentiment-analysis python api.
COPY . .
RUN mvn install
ENV LOGIC_API_URL http://localhost:5000
WORKDIR /target
EXPOSE 8080
CMD ["java", "-jar", "sentiment-analysis-web-0.0.1-SNAPSHOT.jar", "--logic.api.url=${LOGIC_API_URL}"]
```

12

## Webapp docker image

13

Lanciare i comandi per la creazione dell'immagine docker che poi inviamo al registry:

```
docker build -t $DOCKER_USER_ID/webapp .  
docker push $DOCKER_USER_ID/webapp
```

13

## Logic docker image

14

Per poter testare la Webapp dobbiamo prima procedere alla creazione dell'immagine della logica python:

```
FROM python:3.6-slim  
COPY sa /app  
WORKDIR /app  
RUN pip3 install -r requirements.txt && \  
    python3 -m textblob.download_corpora  
EXPOSE 5000  
ENTRYPOINT ["python3"]  
CMD ["sentiment_analysis.py"]
```

14

## Logic docker image

15

Lanciare i comandi per la creazione dell'immagine docker che poi inviamo al registry:

```
docker build -t $DOCKER_USER_ID/logic .  
docker push $DOCKER_USER_ID/logic
```

15

## Webapp+Logic

16

Verifichiamo che sia andato tutto bene facendo partire prima il container della logica con il seguente comando:

```
docker run -d -p 5050:5000 $DOCKER_USER_ID/logic
```

16



## Webapp+Logic

17

Ora per poter interagire la webapp deve sapere l'indirizzo del container della logica, quindi con i seguenti comandi possiamo recuperare i container attivi e andare ad ispezionare quello che ci interessa:

```
docker container list
```

```
docker inspect <container_id>
```

Le informazioni relative all'ip sono indicate nella proprietà `NetworkSettings.IPAddress`

17

## Webapp+Logic

18

```
"NetworkSettings": {  
  ...  
  "Ports": {  
    "5000/tcp": [  
      {  
        "HostIp": "0.0.0.0",  
        "HostPort": "5050"  
      }  
    ]  
  },  
  ...  
  "Gateway": "172.17.0.1",  
  "..."  
  "IPAddress": "172.17.0.3",  
  ...  
}
```

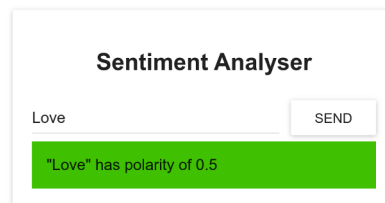
18

# Webapp+Logic

19

Ora possiamo avviare il container della Webapp con il seguente comando:

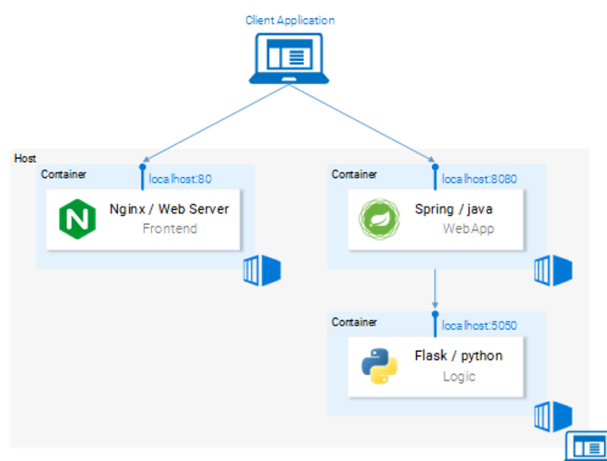
```
docker run -d -p 8080:8080 -e  
LOGIC_API_URL='http://172.17.0.3:5000' $DOCKER_USER_ID/webapp  
Apriamo il browser alla pagina: localhost
```



19

# Webapp+Logic

20



20

## docker-compose

21

- docker-compose è un strumento per la creazione, esecuzione e gestione dei servizi.
- Un servizio è inteso come una replica di uno o più container Docker.

21

## docker-compose

22

- I servizi e i set di servizi vengono definiti in file YAML e gestiti con la CLI docker-compose
- Con docker-compose quindi non ci si concentra più sul singolo container, ma permette di definire degli ambienti in cui valgono le interazioni tra servizi.

22

## docker-compose

23

- docker-compose permette di eseguire i seguenti compiti con semplici comandi:
  - Build immagini Docker
  - Eseguire applicazioni come servizi
  - Eseguire un intero sistema di servizi
  - Gestire lo stato di specifici servizi
  - Scalare i servizi
  - Visualizzare i logs di un intero set di servizi

23

## docker-compose

24

- Un file di docker-compose può descrivere quattro o cinque servizi unici che sono correlati, ma devono mantenere l'isolamento e possono scalare indipendentemente.
- Questo livello di interazione copre la maggior parte dei casi di uso quotidiano per la gestione del sistema.
- Per questo motivo, la maggior parte delle interazioni con Docker avverrà tramite docker-compose.

24

# Comporre i microservizi

25

- Dobbiamo scrivere il docker-compose file per comporre i microservizi appena creati:

```
version: "3"
services:
  frontend:
    image: frontend:latest
    environment:
      WEBAPP_URL: "http://172.16.238.11"
    ports:
      - "80:80"
    networks:
      sentiment:
        ipv4_address: 172.16.238.9
    deploy:
      replicas: 1
      update_config:
        parallelism: 2
        delay: 10s
      restart_policy:
        condition: always
```

25

# Comporre i microservizi

26

```
webapp:
  image: webapp:latest
  environment:
    LOGIC_API_URL: "http://172.16.238.10:5000"
  ports:
    - "8080:8080"
  networks:
    sentiment:
      ipv4_address: 172.16.238.11
  depends_on:
    - logic
  deploy:
    replicas: 1
    update_config:
      parallelism: 1
    restart_policy:
      condition: always
```

26

# Comporre i microservizi

27

```
logic:
  image: logic:latest
  ports:
    - "5000:5000"
  networks:
    sentiment:
      ipv4_address: 172.16.238.10
  deploy:
    replicas: 1
    update_config:
      parallelism: 1
    restart_policy:
      condition: always

networks:
  sentiment:
    ipam:
      driver: default
      config:
        - subnet: "172.16.238.0/24"
```

27