Contanisering av applikationer (Christopher Ek)

PS: You need to sign up for Github CoPilot, it makes coding 5x faster I promise

Uppgift 1

A. True

B. False

C. False

D. True

E. False

F. False

Container: #contanisering-av-applikationer-christopher-ek

Uppgift 2

`Dockerfile` är en YAML fil som används för att definera hur en docker image ska sättas ihop (defines how the docker image should be assembled/forged in the fire) medans `docker-compose.yml` är till för att bestäma hur docker container ska köras. t.ex vilka portar som ska vara aktiva, vilka nätverk de ska tillhöra och namn samt image/build, det finns lite mer paramterar här som environment, depends_on, labels, deploy och sen har vi resource management (jag tar upp det här lite extra eftersom jag inte riktigt visade det under min presentation)

```
# Code is from docker docs
version: "3.9"
services:
  redis:
    image: redis:alpine
    deploy:
```

```
      resources: # here we have resource management
        limits:
          cpus: '0.50' # we set 50% of cpu resources of a single core (cost,
energy, heat)
          memory: 50M
        reservations: # here we have cpu time reserverd (like scheduling)
(priority)
          cpus: '0.25'
          memory: 20M
```

Dockerfile is for one container, docker-compose.yml is for multiple containers.

Uppgift 3

A. to create a presistance volume, you would have to use a seperate volume that is outside the container. with `Dockerfile` one would use the command `VOLUME /usr/src/app` this would only tell where inside the docker container the volume would be inside the container, we use

`-v "$(pwd)/www":/usr/share/nginx/html (get current workdir on host, and cd to www, make that mount point)` if you don't tell where the mount point is docker would choose something, honestly I don't really know where, didnt have time to figure that one out. We can alkso use :ro to make it read only. `also docker run -v host_data:container_data` I know there is also something for mountain volumes from other containers but not really sure, more investigation is required, please hold...

Volumes can be created with `docker volume create/inspect/rm volume_name* docker-compose.yml` we use the tag `volumes:`

```
# Code from Docker docs
version: "3.9" # verison of the docker compose file.

services: # this defines all the containerst that the docker compose file contains

  frontend: # container name

    image: node:lts # container image

    volumes: # persistance volume code from stackoverflow
      - type: volume # what type
        source: myapp # source, like below myapp
        target: /src/app # target inside of docker container
        read_only: true # make it read only

    volumes: # ofc you cant have two volumes tags, but this is only for show.
      - /path/on/host:/path/inside/container:ro # host_data:container_data make it
read only

# defines a volume name on the host computer,
# can be found easily in docker desktop
```

```
volumes:
  myapp: # volume name
```

B. Well i think my A answer kinda killed two birds with one stone, i kinda show how to implement volumes in docker compose and dockerfile above.

Uppgift 4

```
FROM python # default
FROM python:3.10-slim # changed
```

So what is the diffrence between these two?

Not specificing any versio, like `FROM python` will use the latest version.

If you only specifiy the first digit of the version like `FROM python:3` you only get the latest of version of python 3.X. `FROM python:3.10` is static and will always be python 3.10

Well the python image is of cource built ontop of some linux kernel and most like even a distro like alpine, and `FROM python:3.10-slim` focusing on the slim part, that just says that we wanna use the slim version of alpine to built python ontop, we dont dont need unnecessary bloat.

Uppgift 5

A. This one is a doosie..

1. First of all segmentation is very imporant, that why two application that shares nothing should never be on the same network, and should be aware of each other. It's always best to use routers and request handlers to manage traffic on the network, like with traefik or some custom code. This could be accomplished with dockers, networking protocols. So we assign apps to certain networks using the `docker-compose` tag networks:

2. There is something called the CIA Traid, (Confidentiality, Integrity, Availability)

Confidentiality: Protection of data from unauth access and misuse. This step is very important when it comes to data security. We can use The Bell LaPadula Principle to protect data from unauth access and misuse.

Like displayed here Model (IMGUR)

TLDR: Can read down, can't read up. Which means you can only access the data on in the layer below you. This is just like access levels. run applications as restricted users. This can be achived with `docker.cli --user 1000:1000 (runs as given user)` With Docker-compose we can just use the command

```
  user: ${USER}
```

3. certificates, rsa, digital signatures (public-keys & private-keys asymmetric) are used to verify the authenticity of data. So lets say we have a docker image called "Flask-3.10" and we want to run it as a flask

app, we need to make sure that the image is signed by a trusted authority, the creators of the flask app, takes their private key and signs the image, so now you can check with their public key if the image is signed by the person that you think it is. so the Flask-3.10 will only have 1 public key. It's like the opposite of encryption but not decryption. We can do this with

```
# Generate a key pair
docker trust key generate Christopher

# Adding me as a trusted authority
docker trust signer add --ky cert.pem Christopher christopherek/school-api:latest

# Signs the image and pushes to dockerhub
docker trust sign christopherek/school-api:latest
```

B. I think i covered that above, but I can do a rundown again.

1. Network. This will limit the reach of the infected container. Which limits the access scope. this helps inside app to determeine access scope, and also all applications on that server is isolated. I watched the Docker-con 17 video and it didnt really lifke isolation, but i dont think he means like this. This is very good i would say.

2. Access levels, we dont have to use root in all containers we can create users instead with restricted access instead. This makes it so the container dosent have as much reach, since its only a user not admin or sudo.

3. Certificates, this is used to verify the authenticity of data. We use this to simply verify that the container isnt from a shady actor, it could still be infected but its less likely, which helps the internal and external equal.

Uppgift 6

Out of Memory Exception

Here we can just use `-m or --memory=500` for cli and for `docker-compose`

```
resources: # here we have resource management
  limits:
    memory: 500M # 'here we can limit the memory
```

To not get OOM we can use the command `--oom-kill-disable` this will make it so we don't kill the kernel when the container is trying to access more memory then it has. keep in mind that memory limits must be set or the container will fail eventually.

We could keep OOM-kill on, and just set the tag `restart: always` or `restart: on-failure` to keep the scope down. OOM is an error so this should work.

Here I would also maybe implement a the dockerfile ref `HEALTHCHECK` to keep track of the container, so it dosent impact other containers to much.

Uppgift 7

Can image on dockerhub be trusted...

A. No.

B. Because everyone can upload something to docker hub, and therefor we can't always trust the data. If a docker-image is from a verified source, you see a blue checkmark that says `Verified Publisher` but you also have official docker image, which are labeled `Official Image`, this can be trusted to come from the correct source, but that source could still be infected. Official images can be trusted, but could still be infected from third party.

Uppgift 8

sudo or no sudo A.

Consequence

1. Full data loss
2. Personal Information leak
3. Full scale data breach

B. Well we downgrade the container user, to something less privileged. Will remove scale down the potentaial data loss. The database server might not need to root, because of someone gets `RCE` with `full shell-access` it cant delete folders that are owned by some other user/group policy, so you can't delete the database, unless you contact the database via the API for example.

Uppgift 9

| *docker-compose inspect?*

`docker-compose` is used to define and control multiple containers. One docker-compose file could be used to create a full-stack web application. and the other one for a full-stack phone application.

With docker-compose we can simply define memory, ports, networks, resources, and volumes etc. We define how they should be run but not how the images could be assembled.

Uppgift 10

| *isolation*

Well to be fair i think i've covered that in the previous questions but, its good to keep things isolated to control the scope for all containers. Limit networks, remove exposed ports use the same volume but for different use-cases.

Uppgift 11

| *layers?*

```
# Question
# Vad hade du kunnat göra för att optimera och förbättra denna Dockerfile mer?
Beskriv vilka förändringar du skulle göra och motivera varför.
```

```
FROM python:3.10 # use exact version to avoid issues
# Remember to update this for every build
LABEL version="23.2"
WORKDIR /
COPY app/* .

RUN pip install fastapi
RUN pip install uvicorn
RUN pip install pydantic
RUN pip install -r requirements.txt

RUN apt-get update
LABEL description=""
RUN apt-get install nano vim

ARG PORT=8000
ENV PORT=$PORT
EXPOSE $PORT
LABEL maintainers="The Knights who say Ni!"

WORKDIR /scripts
CMD /scripts/start-server.sh
```

1. Change `python to python:3.10-slim` to use exact version, this ensures that our container will always work, and wont be affected by updates. Also added slim for less resources. I have answerd this in the previous question.

```
# old
FROM python

# new
FROM python:3.10-slim
```

2. Well since all statements that interact with the container will create a new layer. So thats why we wanna set the ARG and Label at the top of the Dockerfile. So when we runt the first statement that interact with the container, those ARG and labels will be cached in the layer.

```
# Args
ARG SERVER_FILE="main.py"
ARG MAINTAINER="Christopher Ek"
ARG AUTHOR="Christopher Ek"
ARG PORT_LISTEN="8000"

# env
ENV FILE=$SERVER_FILE


# Labels
LABEL author=$AUTHOR
```

```
LABEL maintainer=$MAINTAINER
LABEL version="0.1"
LABEL desc="Lorem ipsum"

WORKDIR /app

COPY * . # copy all files in the directory

# ^ here we create layer one,
# so this will be cached if it was successful.
# everything ^above^ will be cached in layer 01
```

1. Copy app/* . is not really correct, alot of margin of error.. Instead we would do this.

```
COPY ${pwd}/app/* .
```

4. I would also set the workdir to /app, it makes more sense.

```
WORKDIR /app
```

6. Use ARG to define label version

```
# ARG
ARG VERSION="unkown"
# LABEL
LABEL=$VERSION
```

7. Add libaries to requirements.txt and also use pip freeze and .lock to lock the file.

```
# Old
RUN pip install fastapi
RUN pip install uvicorn
RUN pip install pydantic
RUN pip install -r requirements.txt

# new
COPY requirements.lock .
RUN pip install -r requirements.lock
```

8. Multi-stage build

The purpose using multi-stage builds is to safe on resources, like having stage 1 be compiling and stage 2 be running, and only copy the exe file. It creates a very nice boundary between production and development,

since production is always the exe running and development has all files. This is good segmentation, and lowers the risk of accidental damage to the production build.

Remove EXPOSE & PORT, let docker-compose do the work.

Don't judge me to hard im writing from scratch concept

```
### -------- ###
# -- MULTI --- #
# -- STAGE --- #
# -- BUILD --- #
### -------- ###
# Author: Christopher Ek
# Version: 1.12.1


### ------------------------------  STAGE 1  ------------------------------------
-- ###
FROM python:3.10-slim as builder

ARG MAINTAINER="Christopher Ek"
ARG Version="23.2"
ARG PORT="5000"
# Make is permanent sort of.
ENV PORT=$PORT

LABEL Description=""
LABEL MAINTAINER="${MAINTAINER}"
LABEL Version="${Version}"

WORKDIR /app
# Copy and download dependency using go mod.
COPY requirements.txt .
# Layer 01 ^
RUN pip install -r requirements.txt
# Layer 02 ^ also remove req

# When that is cached, we copy the rest.
COPY . .

# Create the exe file
RUN pip install pyinstaller
RUN pyinstaller --onefile main.py
### ------------------------------  STAGE 2  ------------------------------ ###
# mini image, to only run the exe file
FROM scratch

# Copy from builder or stage 1
COPY --from=builder ["/app/dist/main.exe", "."] # Copy the executable to the root
of the image.

# get some updates.
RUN apt-get update
```

```
# no nano here,
# I would remove this line,
# because we dont need any editor.
# on the container that just runs the exe file.
RUN apt-get install nvim

# run the application
ENTRYPOINT [ "./main" ] # Run the executable.
```

docker-compose file

```
# Instead of using EXPOSE in the Dockerfile, we can use docker-compose.
services:
  exe:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: exe
    ports:
      - 8000:8000 # host:container
```