# Docker notes

A **container** is an isolated process on your host machine.

- A containers needs to fulfill four major requirments
    - Can only be ran on one host, two computers cannot run a single container
    - They are groups of processes. Containers must have a root process
    - They need to be isolated
    - The have to fulfill common features, features can change over time.
- ▼ It isolates itself via "kernel namespaces and cgroups"
    - Chroot
        - chroot is a Linux command to change the root directory of the current running process and its children. The area is often referred to as a jail.
        - https://miro.medium.com/max/700/1*QuHou1T2lBoPgi8-dB7kBQ.png
        - Making a jail does not make something secure. This is because relative paths can still refer to files outside of the new root.
        - You can exit a jail by running the following script

        ```
        #include <sys/stat.h>
        #include <unistd.h>

        int main(void)
        {
            mkdir(".out", 0755);
            chroot(".out");
            chdir("../../../../../");
            chroot(".");
            return execl("/bin/bash", "-i", NULL);
        }
        ```

        - This script will create a new directory, make it the new root directory, change to a higher directory and then change the root directory to the current one, which then spawns a shell.
        - Chroot is no longer used by container runtimes, and was replaced by `pivot_root(2)`
        - `pivot_root(2)` puts old mounts into a separate directory on calling. Old mounts can be unmounted afterwards to make the filesystem invisible to broken out processes
    - Namespaces
        - A namespace is a way to wrap certain global system resources in an abstraction layer, making it appear like the process within a namespace has its own isolated insantance of the resource.
        - It cannot view other processes running on the host, it cannot connect to its internet or intranet
    - API
        - The namespace API of the linux kernel consists of three main system calls:
            - The `clone(2)` API function creates a new child process and can dictate if it shares its memory space, the table of file descriptors, and the table of signal handlers.
            - The `unshare(2)` function allows a process to disassociate parts of the exection context which are currently being shared with outhers.

- The `setns(2)` function reassociates the calling thread with the provided namespace file descriptor. This function can be used to join existing namespaces

A **Dockerfile** is a text-based script of instructions which creates a container image

To build the container image use `docker build -t name .`

To run a container use `docker run -dp 3000:3000 getting-started`

- `-d` tells us to detatch the image (run it in the background)
- `-p 3000:3000` tells it to match port 3000 of the host to port 3000 of the image

`docker ps` lists the IDs of the running containers

`docker stop <ID>` in order to stop the container process

In order to replace the container, you must first remove the old one with `docker rm <ID>`

The prior two commands can be combined with `docker rm -f <ID>`

When trying to push a container to dockerhub with the command `docker push <DockerID>/getting-started:tagname` , you will get the following error:

```
The push refers to repository [docker.io/sm9l/getting-started]
An image does not exist locally with the tag: sm9l/getting-started
```

This is because the tag is different from the local one, fix this with the following command

`docker tag getting-started YOUR-USER-NAME/getting-started`

Push once again `docker push YOUR-USER-NAME/getting-started` , if you don't add the `:tagname` docker will just use the default `:latest`

Each container has its own "**scratch space**", to create/upload/remove files, even if they are using the same image.

In order to execute a bash command on container start use

`docker run -d ubuntu bash -c "bash command string"`

In order to execute a command in the container use

`docker exec <containerID> "command string"`

**Volumes** provide a way to connect specific filesytem pahs of the container back to the host machine

**Mounting** is when you create a volume and attach it to the directory the data is stored in, which can then persist the data

You can create a volume with the name "todo-db" via the following command

`docker volume create todo-db`

To start the todo app with volumes use the command `docker run -dp 3000:3000 -v todo-db:/etc/todos getting-started` where `todo-db` is name of the volume and `/etc/todos` is the directory where our database is stored

Now every time you run that command to start the container, the data will persist

If you want to see where the data is being stored locally use `docker volume inspect <volumeName>`

**Bind mounts** are similar to regular mounts except you can control the exact mountpoint on the host

This is useful when the source code running in the container is on the host and can be worked on in real-time. The nodemon package allows for the program to update without having to relaunch

| | Named Volumes | Bind Mounts |
|---|---|---|
| Host location | Docker chooses | You control |
| Mount Example (using -v) | my-volume:/usr/local/data | /path/to/data:/usr/local/data |
| Populates new volume with container contents | Yes | No |
| Supports Volume Drivers | Yes | No |

The "docker development workflow" includes mounting the source code into the container, install all of the needed dependencies, and enabling nodemon.

### Multi container apps

Each container should do one thing and do it well

This is because:

- different aspects of the project will scale at different rates
- isolation allows you to update versions easily
- you will use different databases when testing locally vs on production, thus you dont have to ship it if it is independant
- Containers only start one process, running multiple will require a process manager which adds complexity

As containers are in isolation we need to use networking in order to allow them to communicate

Create a network by using the command `docker network create <name>`

To start a MySQL container and attach it to the network do the following

```
docker run -d \
     --network todo-app --network-alias mysql \
     -v todo-mysql-data:/var/lib/mysql \
     -e MYSQL_ROOT_PASSWORD=secret \
     -e MYSQL_DATABASE=todos \
     mysql:5.7
```

Each container has its own ip

The nicolaka/netshoot container has alot of tools for debugging network issues, run it with `docker run -it --network todo-app nicolaka/netshoot`

To find a container's IP in the same network, enter the netshoot container and do `dig <network-alias/hostname>`, you often do not need to know the IP, as the network-alias maps to the IP in the same way you use a domain name.

SQL connection settings can be stored in the env vars

**This is dangerous and discouraged as env vars can be leaked**

To start the app container with sql connection do the following

```
docker run -dp 3000:3000 \
  -w /app -v "$(pwd):/app" \
  --network todo-app \
  -e MYSQL_HOST=mysql \
  -e MYSQL_USER=root \
  -e MYSQL_PASSWORD=secret \
  -e MYSQL_DB=todos \
  node:12-alpine \
  sh -c "yarn install && yarn run dev"
```

Any additions made to the todo list can be seen in the sql db

### Docker Compose

The Docker Compose tool enables you to define and share multi-container apps. This happens via using a YAML file, which is kept at the root of the project rope.

`docker-compose.yml` will look like this

```
version: "3.7" #this should be the latest version

 services: #this is a list of services or containers we want to run as part of our app
   app: #this is the name of the app and will auto become a network alias
     image: node:12-alpine #this is the image we are working on
     command: sh -c "yarn install && yarn run dev" #this is the command to be ran
     ports: #this defines the ports
       - 3000:3000 #this is called "shirt syntax" there is also long syntax
     working_dir: /app #this defines the working directory we are using
     volumes: #this defines the volume we are mapping too
       - ./:/app
     environment: #this sets the env variables for the mysql connection settings
       MYSQL_HOST: mysql
       MYSQL_USER: root
       MYSQL_PASSWORD: secret
       MYSQL_DB: todos
```

This is a cleaner way of writing the command we used before

```
docker run -dp 3000:3000 \
  -w /app -v "$(pwd):/app" \
  --network todo-app \
  -e MYSQL_HOST=mysql \
  -e MYSQL_USER=root \
  -e MYSQL_PASSWORD=secret \
  -e MYSQL_DB=todos \
  node:12-alpine \
  sh -c "yarn install && yarn run dev"
```

```
docker run -d \
  --network todo-app --network-alias mysql \
  -v todo-mysql-data:/var/lib/mysql \
  -e MYSQL_ROOT_PASSWORD=secret \
  -e MYSQL_DATABASE=todos \
  mysql:5.7
```

Likewise, instead of using the messy command to start the mysql service, we can add the following snippet to `docker-compose.yml`

```


   mysql: #the name "mysql" goes in the app "services" list
     image: mysql:5.7
     volumes:
       - todo-mysql-data:/var/lib/mysql
     environment:
       MYSQL_ROOT_PASSWORD: secret
       MYSQL_DATABASE: todos

 volumes: #with docker compose, volumes have to be defined at the top level
   todo-mysql-data:
```

The final yaml file should look like this

```
version: "3.7"

services:
  app:
    image: node:12-alpine
    command: sh -c "yarn install && yarn run dev"
    ports:
      - 3000:3000
    working_dir: /app
    volumes:
      - ./:/app
    environment:
      MYSQL_HOST: mysql
      MYSQL_USER: root
      MYSQL_PASSWORD: secret
      MYSQL_DB: todos

  mysql:
    image: mysql:5.7
    volumes:
      - todo-mysql-data:/var/lib/mysql
    environment:
      MYSQL_ROOT_PASSWORD: secret
      MYSQL_DATABASE: todos

volumes:
  todo-mysql-data:
```

To run the application stack, use the command `docker-compose up` , you can also use the `-d` flag to run it in the background

To view the logs or see errors use `docker-compose logs` and use `-f` to see it update live or "following" the logs. You can also add the name of a specific service to see those exclusive logs

In order to tear down the app, use `docker-compose down`, this will **not** remove the volumes. To remove volumes add `--volumes`

## Image-building best practices

To scan it for security vulnerabilities use `docker scan <imageName>`

To see the layers in the image use, `docker image histoy <imageName>`

a file with the name `.dockerignore` will allow you to selectivley blacklist files to not copy, it is recommended that you but dependancies in, eg `node_modules`

You can use caching in Dockerfile to speed up build times dramatically, assuming you are making basic changes (the code).

## Explaining the Dockerfile (specifically for a node project)

1) The first line is the **parser directive**, this is optional however recommended. `# syntax=docker/dockerfile:1` , the `docker/dockerfile:1` points to the latest release of version 1 syntax

2) The second line is the base image we want to use, eg `FROM node:12.18.1`, base images are like toolkits

3) The third line is the node environment `ENV NODE_ENV=production`, setting it to production increases performance

4) Next we are going to do `WORKDIR /app`, this is the equivilent of `mkdir app && cd app`, all following commands will be done in the folder

5) `COPY ["package.json", "package-lock.json*", "./"]` this allows us to carry the dependency files *into* the image.

6) `RUN npm install --production` allows us to execute the command on the command line to install the dependencies, *inside* the image.

7) copy your source code into the image using `COPY . .`, the first dot refers to the directory the

8) Finally run a command in the container using `CMD [ "node", "server.js" ]`