**DOCKER**

**DOCKER - Overview**

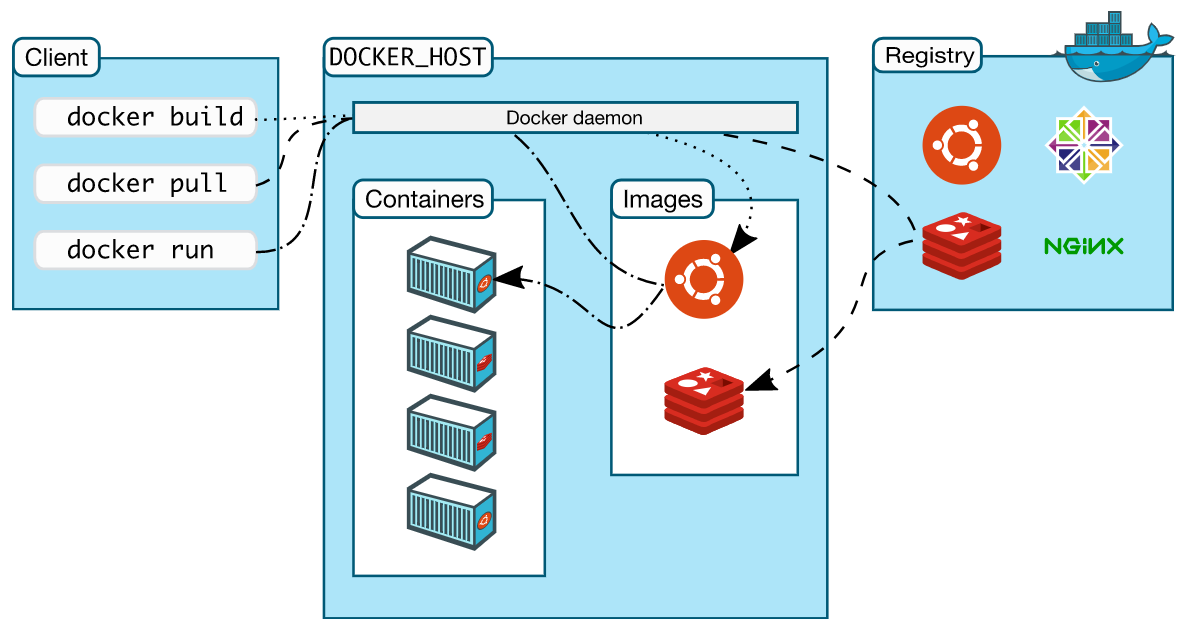
- Is an open platform for developing, shipping, and running applications on any OS

- Separates your applications from your infrastructure

- Manages your infrastructure in the same ways you manage your applications

- Is fast, lightweight, easy scalable, easy portable and easier to maintain

- Can significantly reduce the delay between writing code and running it on PROD



**DOCKER - Client**

* Primary way a user interacts with Docker
* Commands are send to the Docker host via e.g. API or CLI
* One client can communicate with many Docker daemons

**DOCKER - Daemon**

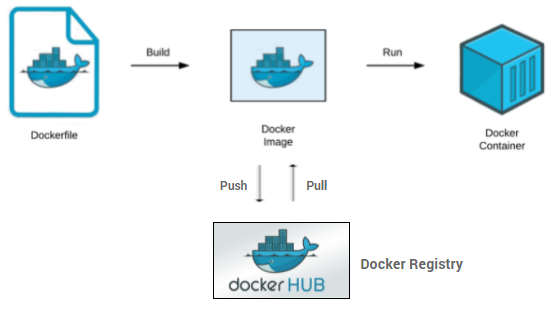
* Listens to API or CLI Requests
* Manages Docker objects
* Can communicate with other Docker daemons

**DOCKER - Registries**

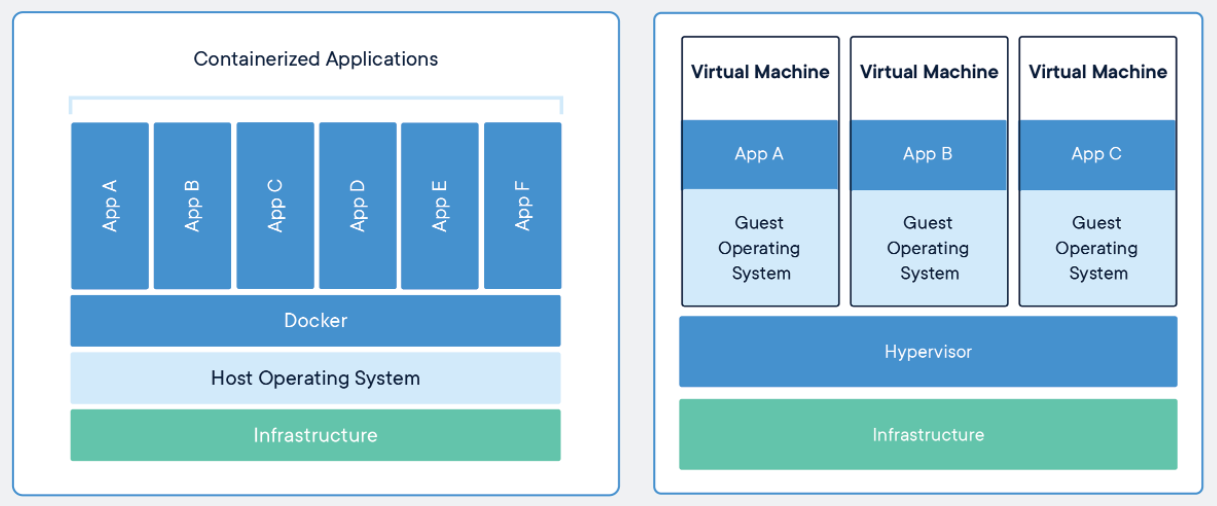
* Stores and provides Docker Images (DockerHub / AWS ECR)
* Images can be public or private
* Think of GitHub for Docker images (pull / push)

**DOCKER - Objects**

* Images
* ReadOnly templates for Container creation
* Can be moved / deleted / pulled / pushed / published via API or CLI
* Can be created and configured from a Dockerfile
* Containers
* Runnable instance of an Image
* Can be created / started / stopped / moved / deleted via API or CLI
* Many containers can be executed and configured via a DockerCompose file
* Volumes
* Contains the data of a docker container lives
* Can be used for persistence of that data
* Can be mirrored to a folder of a client (e.g a directory on your computer )
* Networks (out of scope)
* Plugins (out of scope) and more …



**DOCKER Container VS Classic VMs**



| **Component** | **Containers** | **Virtual Machines** |
| --- | --- | --- |
| **Emulation Layer** | Uses container engine to isolate multiple processes in a shared OS kernel | Uses hypervisor to manage guest operating systems |
| **Speed** | Starts in seconds | Starts in minutes |
| **Size** | Sized in megabytes | Sized in gigabytes |
| **Minimum Cost** | Tens of dollars | Hundreds of dollars |
| **Scalability** | Easy to scale with speed and small size | Supports some scaling but will reach compute resource limits quickly |
| **Replaceability** | Easy | Difficult |
| **Resource Isolation** | Moderate | Strong |
| **Monolithic Applications** | Inefficient for monolithic applications | Supports monolithic applications |
| **Complexity for Dev Teams** | Low | High |
| **Dev Team Interaction with Infrastructure** | Low | High |

**DOCKER - Tutorial - Part 1**

Content: Setting up a node.js app within an Docker Container

1. **Pre-requirements**

* Docker installation
* Node.js and npm must be installed
* Basic Node.js and npm knowledge
* IDE, e.g. VS Code

1. **Setup Express App**

* Create project folder named node-docker-app  
  *$ mkdir node-docker-app*
* Cd into the new directory

*$ cd node-docker-app*

* Open VSCode IDE and open the folder
* Init the project

*$ npm init -y*

* Install Express

*$ npm install express*

* Create index.js

const express = require("express");

const app = express();

const port = process.env.port || 2000;

app.get("/", (req, res) => {

res.send("<h2>Hi Quinscape!");

})

app.listen(port, () => console.log(`listen on port ${port}`));

* Start express app  
  *$ node index.js*
* Navigate in browser to localhost:2000 and STOP the app afterwards

1. **Setup docker image and run a it within a Container**

* Navigate to <https://hub.docker.com> and search for "node"
* Click on node.js which is going to be our base image, look through instructions
* Create a "Dockerfile" in the project root directory and copy content

# Selects node.js with version as base image from public Docker repository.

FROM node:15

# Defines (and creates) a working directory within our container.

# Any commands are running from this directory, it is the default path.

WORKDIR /app

# Copy package.json into the working directory of the container.

COPY package.json .

# Installs express and the other dependencies from the package.json.

RUN npm install

# Copies any file from the local node-docker-app folder to the working # dir of the container (including again the package.json).

# This is an optimization. As the steps/layers/commands are cached and

# are not executed again, when nothing has changed. Depencies do only # change sometimes, whereas the code in the /app dir usually changes often.

COPY . ./

# Exposes port 2000 without effect (this line is just for documentation)

EXPOSE 2000

# Runs the application entry point index.js (runtime) when the # container starts.

CMD ["node", "index.js"]

* Build the extended Docker image (from the node.js base image)  
  *$ docker build .*
* List all Docker images  
  *$ docker image ls*
* Remove Docker image again, as it has no name  
  *$ docker image rm [IMAGE-ID]  
  $ docker image ls*
* Build the extended Docker image with a name (from the node.js base image)  
  *$ docker build -t node-docker-app-image .  
  $ docker image ls*
* Run the node-docker-app-image in a container named node-docker-app in detached mode (without opening a port) and list all running containers  
  *$ docker run -d --name node-app node-docker-app-image  
  $ docker ps*
* Refresh localhost:2000 in the browser (does not work) as no ports have been opened (EXPOSE 2000, was only for documentation)
* Stop the Docker container again  
  *$ docker stop [IMAGE-ID]  
  $ docker ps*
* List all container (not only the running ones)  
  *$ docker ps -a*
* Remove the docker container. The -f allows the removal of a running container.  
  *$ docker rm node-app -f  
  $ docker ps -a*
* Rerun the container with TCP port mapping (Maps TCP port 2000 in the container to port 3000 on the Docker host)   
  *$ docker run -p 3000:2000 -d --name node-app node-docker-app-image*
* Show in localhost:2000 (does not work) and localhost:3000 in the browser.
* Stop and start an existing container  
  *$ docker stop node-app  
  $ docker start node-app*

**DOCKER - Tutorial - Part 2**

Content: Setting up a dockerignore file, and auto-update the Container application automatically when the code has changed locally on the Docker host system in the DEV environment. Additionally a short introduction into environment variable handling is appended at the end of this session.

1. **Pre-requirements**

**-** Completion of Tutorial Part 1

1. **Setting up a .dockerignore file**

**-** Stop and remove container, remove source image, rebuild image, and re-run container

*$ docker rm node-app -f  
$ docker image rm node-docker-app-image  
$ docker build -t node-docker-app-image .  
$ docker run -p 3000:2000 -d --name node-app node-docker-app-image*

**-** Bash into the docker container to look-up what has been copied into the /app folder due to the COPY command of the Dockerfile. Exit afterwards.  
*$ docker exec -it node-app bash  
$ exit*

**-** Create a .dockerignore file in the project’s root folder

node\_modules

.dockerignore

.git

.gitignore

**-** Delete the running container, build a new image and run it in a new Docker container to verify that the .dockerignore file works.   
Note, that the node\_modules folder within the Docker Container is still existing. The reason therefore is that the RUN npm install command ofthe Dockerfile creates the node\_modules folder. The node\_modules exclusion in the .dockerignore file has the effect that COPY . ./ (which is executed after the RUN npm install command) no longer copies the node\_modules folder from the local into the container.  
*$ docker rm node-app -f  
$ docker build -t node-docker-app-image .   
$ docker run -p 3000:2000 -d --name node-app node-docker-app-image  
$ docker exec -it node-app bash  
$ exit*

1. **Setting up auto-deploy on code changes by use of Docker volumes and codemon for DEV.**

**-** Change code locally and check in browser whether the change has an effect

app.get("/", (req, res) => {

res.send("<h2>Hi Quinscape! Docker rules!</h2>");

})

**-** Bash into the container and lookup whether the index.js has changed there  
*$ docker exec -it node-app bash  
$ cat index.js  
$ exit*

**-** Stopping the container, rebuilding the image and rerunning would of course update the code change within the container, but this is cumbersome.  
*$ docker rm node-app -f  
$ docker build -t node-docker-app-image .   
$ docker run -p 3000:2000 -d --name node-app node-docker-app-image  
$ docker rm node-app -f*

**-** To auto-update the code in the Docker Container a Host Volume must be setup  
*$ docker rm node-app -f  
$ docker run -v ${pwd}:/app -p 3000:2000 -d --name node-app node-docker-app-image  
  
Windows-CmdShell: %cd%   
Windows-PowerShell: ${pwd}  
Mac/Linux-Shell: $(pwd)*

**-** Check in the browser if the code changes are reflected already.

**-** To regard code changes the node process must be restarted (automatically), which can be achieved by codemon.  
$ *npm install nodemon --save-dev*

**-** As for now only DEV should react to code changes, we also need to update the package.json scripts. Note, that especially for Windows the -L might be required, to resolve certain errors. (Out of scope, google the error message for deeper insights).

"scripts": {

"start": "node index.js",

"dev": "nodemon -L index.js"

},

**-** The Dockerfile need to be adopted accordingly, so replace   
CMD ["node", "index.js"] by CMD ["npm", "run", "dev"]

**-** Stop, remove the Docker Container, rebuild the Docker Image and re-run the Container with the Host Volume again, so that the recent changes can take effect.

*$ docker rm node-app -f*

*$ docker build -t node-docker-app-image .*

*$ docker run -v ${pwd}:/app -p 3000:2000 -d --name node-app node-docker-app-image*

**-** Change the response in the index.js and check in the Browser whether the changes are reflected immediately.

**-** Let’s do some tests with volumes. Delete the local node\_modules folder, as we do not really need it locally. And then test whether the app is still runnable, therefore redeploy (docker run) the application and check in the Browser whether the app still works. Stop and remove the container afterwards.  
*$ docker run -v ${pwd}:/app -p 3000:2000 -d --name node-app node-docker-app-image  
$ docker ps  
$ docker ps -a  
$ docker logs node-app*

**-** The reason for this is that when the Dockerbuild file command COPY . ./ is executed the complete /app folder in the Docker Container is replaced with the local volume, where we just had deleted the node\_modules folder.  
To solve this, we add an additional more specific (longer path) anonymous volume, which prevents the copy command to remove the node\_modules folder in the Docker Container  
*$ docker rm node-app -f  
$ docker run -v ${pwd}:/app -v /app/node\_modules -p 3000:2000 -d --name node-app node-docker-app-image*

**-** Is the Host directory bi-directional mounted? Let’s test it. So first bash into the Container.  
*$ docker exec -it node-app bash*

**-** Create a test1.txt file in the IDE and see if it appears in the container, as expected. Remove test1.txt afterwards  
*$ ls  
$ rm test1.txt*

**-** Now create a test2.txt file within the container and see if it appears in the IDE, as expected. Remove it afterwards.  
*$ ls  
$ touch test2.txt  
$ rm test2.txt  
$ exit*

**-** The latter is not really intended in most situations, it might even be a security issue. To solve this, make the mounted Docker Host Volume readOnly.  
*$ docker rm node-app -f*  
*$ docker run -v ${pwd}:/app:ro -v /app/node\_modules -p 3000:2000 -d --name node-app node-docker-app-image*

**-** Repeat the test and create a file within the /app folder of the Docker Container.

**-** Lists the created volumes  
*$ docker volume ls*

**-** Removes all local Volumes not used by at least one Container  
*$ docker volume prune*

**-** Also (beside the Container itself) deletes the volume of a (running) container, due to the appended ‘v’.  
*$ docker rm node-app -fv*

1. **Making use of Environment variables**

**-** Create an .env file in the project’s root folder with the following content.

PORT=4000

**-** In the index.js change const port = process.env.port || 2000; to const port = process.env.PORT || 2000;

**-** Change the port also in the Dockerbuild file for the docs (although this does have no effect). So replace EXPOSE 2000 by EXPOSE $PORT

**-** Use the env file when the Docker Container is started again  
*$ docker run -v ${pwd}:/app:ro -v /app/node\_modules --env-file ./.env -p 3000:4000 -d --name node-app node-docker-app-image*

**-** Bash into the container and check whether the environment variable has been set  
*$ docker exec -it node-app bash  
$ printenv*

*$ exit*

**-** Note that the following command would have the same effect and that you still can overwrite the env variables this way, but this is clunky when you want to set many environment variables  
*$ docker run -v ${pwd}:/app:ro -v /app/node\_modules --env PORT=5000 -p 3000:5000 -d --name node-app node-docker-app-image*

**-** Also note, that running such a long command to start a single container, is in a real-world-project, where you tend often to manage half a dozen containers, not very convenient. That brings us to Docker Compose, and Part 3 of this tutorial series.

**-** Finally remove the running Docker Container  
*$ docker rm node-app -fv*

**DOCKER - HandsOn - Part 3**

Content: Sets up docker compose supporting DEV and PROD environment, adds document based Mongo DB Container and connects to the Mongo DB in the index.js.

1. **Pre-requirements**

**-** Completion of Tutorial Part 1 & 2.

1. **Docker Compose**

**-** Create a docker-compose.yml the project’s root folder.

version: "3.9"

services:

node-app:

build: .

ports:

- "3000:4000"

volumes:

- .:/app:ro

- /app/node\_modules

env\_file:

- ./.env

**-** Note that the yml above substitutes the *$ docker run -v ${pwd}:/app:ro -v /app/node\_modules --env-file ./.env -p 3000:4000 -d --name node-app node-docker-app-image* command.

**-** Build the image and execute the docker-compose.yml  
*$ docker compose up -d*

**-** Stop all running composed Docker Container and remove all (**-v**) anonymous and named Volumes of Containers started though the docker-compose command  
*$ docker-compose down -v*

**-** Important to know is that a second *$ docker compose up -d* will not rebuild the image(s). It takes the already existing one(s). If the image, due to recent changes e.g. in Dockerfile, needs to be built again, append --build to the ‘up’ command.  
*$ docker-compose up --build  
$ docker-compose down -v*

**3. Using Docker Compose for DEV and PROD environments**

**-** Stop all running composed Docker Container and remove anonymous and named Volumes (-v)  
*$ docker-compose down -v*

**-** Change content of docker-compose.yml to

version: "3.9"

services:

node-app:

build: .

ports:

- "3000:4000"

# Note, that there should be different env files for different services.

env\_file:

- ./.env

**-** Note, that **volumes:** part in the docker-compose.yml has been removed, as this should only take effect on DEV.

**-** Create a docker-compose.dev.yml in the project’s root directory.

version: "3.9"

services:

node-app:

volumes:

- .:/app:ro

- /app/node\_modules

environment:

- NODE\_ENV=development

# Overwrites the run command 'CMD ["npm", "run", "dev"]' in the

# Dockerfile.

command: npm run dev

**-** Create docker-compose.prod.yml in the project’s root directory. Note, that we do not need the volumes here, as on PROD codemon is not active (see package.json scripts section), as local code changes shall NOT be released directly/automatically via Volume mounting.

version: "3.9"

services:

node-app:

environment:

- NODE\_ENV=production

# Overwrites the run command 'CMD ["npm", "run", "dev"]' in the

# Dockerfile.

command: npm start

**-** Update the .dockerignore file accordingly by adding:

docker-compose\*

**-** Run Docker Compose for DEV:  
*$ docker-compose -f docker-compose.yml -f docker-compose.dev.yml up -d*

*$ docker ps*

**-** Stop Docker Compose for DEV and delete anonymous and named volumes:  
*$ docker-compose -f docker-compose.yml -f docker-compose.dev.yml down -v  
$ docker ps*

**-** Run Docker Compose for PROD:  
*$ docker-compose -f docker-compose.yml -f docker-compose.prod.yml up -d  
$ docker ps*

**-** Note, on PROD code changes do not apply as nodemon is not installed on PROD. In order to reflect code changes on PROD the app needs to build again.  
*$ docker-compose -f docker-compose.yml -f docker-compose.prod.yml down -v*  
*$ docker-compose -f docker-compose.yml -f docker-compose.prod.yml up -d --build  
$ docker ps*

**-** Bash in PROD /app/node\_modules. On PROD the DEV dependencies (like nodemon) are not required and should therefore be not installed.  
*$ docker exec -it node-docker-app-3\_node-app\_1 bash*

*$ cd node\_modules*

*$ ls | grep node*

*$ ls*

*$ exit  
$ docker-compose -f docker-compose.yml -f docker-compose.prod.yml down -v*

**-** To avoid the npm install execution for DEV dependencies on PROD the Dockerfile needs to be extended by a bash script. So replace the first snippet below by the second snippet.

# Installs express and the other dependencies from the package.json.

RUN npm install

# Install express and the all other dependencies from the package.json

# (including the dev-dependencies) for development, but only install the

# none dev-dependencies for production.

ARG NODE\_ENV

RUN if [ "$NODE\_ENV" = "development" ]; \

then npm install; \

else npm install --only=production; \

fi

**-** Additionally we need to overwrite the build: . part of the docker-compose.yml.

**-** So in the docker-compose.dev.yml insert as first child of the node-app: node the following block.

build:

context: .

args:

NODE\_ENV: development

**-** And within the docker-compose.prod.yaml insert as first child of the node-app: this block.

build:

context: .

args:

NODE\_ENV: production

**-** Rerun Docker Compose PROD and check whether the dev dependency codemon is now missing in the node\_moduls folder as expected. Note, that the *--build* arg is required as we changed the **Dockerfile**.  
*$* *docker-compose -f docker-compose.yml -f docker-compose.prod.yml up -d --build*  
$ *docker exec -it node-docker-app-3\_node-app\_1 bash*

*$ cd node\_modules*

*$ ls | grep node*

*$ exit  
$ docker-compose -f docker-compose.yml -f docker-compose.prod.yml down -v*

**3. Using Docker Compose to setup a Mongo DB Server**

**-** Add mongoose as a dependency to the project.  
*$ npm install mongoose*

**-** Visit the link below, to get quick infos about the Docker Mongo Image, provided by DockerHub. <https://hub.docker.com/_/mongo>

**-** Add Mongo as child to the services: node in the docker.compose.yml.

mongo:

image: mongo

environment:

* MONGO\_INITDB\_ROOT\_USERNAME: admin

MONGO\_INITDB\_ROOT\_PASSWORD: root

**-** Append the depends\_on as last child of the node\_app: node.

depends\_on:

- mongo

**-** In order to ensure that the recent changes take effect properly it seems that the **--build** appended to the docker-compose up command is not sufficient here. So if you run into trouble here simply delete the affected images and the use the **--build** flag with docker-compose up. (Note that you might need to delete the related containers first.  
*$ docker image rm node-docker-app-3\_mongo\_1  
$ node-docker-app-3\_node-app\_1*

**-** Now start the Dev Environment and Bash into (newly added) the Mongo Container with the **--build** flag.  
*$ docker-compose -f docker-compose.yml -f docker-compose.dev.yml up -d --build*

*$ docker ls*

*$ docker exec -it node-docker-app-3\_mongo\_1 bash*

*# mongo -u "admin" -p "root"*

*> db*

*> use mydb*

*> show dbs*

*> db.books.insert({"name": "Bernd-Dieter Pete"})*

*> show dbs*

*> db.books.find()*

*> exit*

*# exit*

**-** Stop the Containers with the -v argument, restart them and bash again into the mongo Container.  
*$ docker-compose -f docker-compose.yml -f docker-compose.dev.yml down -v  
$ docker-compose -f docker-compose.yml -f docker-compose.dev.yml up -d  
$ docker exec -it* node-docker-app-3\_mongo\_1 *mongo -u "admin" -p "root"  
> db  
> use mydb  
> show dbs  
> db.books.find()  
> exit  
# exit  
$ docker volume ls  
$ docker-compose -f docker-compose.yml -f docker-compose.dev.yml down -v  
$ docker volume ls*

**-** We lost our database as we have deleted and recreated the container. To ensure that this volume does not get deleted, we need to create a named volume **AND must no longer use the -v argument**, as besides the container itself it also deletes all contained anonymous and named volumes (and custom networks). To create a named volume add the lines below as child to the mongo: node in the docker-compose.yml.

volumes:

- mongo-db:/data/db

**-** The volume’s name is **mongo-db**. And the path within the container to the database data is (container working directory) **/data/db**. This path can be found in the image documentation on <https://hub.docker.com/_/mongo>

**-** Additionally we need to declare this volume as a root element in the docker-compose.yml.

volumes:

mongo-db:

**-** Now test it, as shown above.  
*$ docker volume ls  
$ docker-compose -f docker-compose.yml -f docker-compose.dev.yml up -d  
$ docker volume ls  
$ docker exec -it node-docker-app-3\_mongo\_1 mongo -u "admin" -p "root"  
> db*

*> use mydb*

*> show dbs  
> db.books.insert({"name": "Bernd-Dieter Pete"})  
> db.books.find()  
> exit  
$ docker volume ps*

*$ docker-compose -f docker-compose.yml -f docker-compose.dev.yml down*

*$ docker volume ps*

*$ docker-compose -f docker-compose.yml -f docker-compose.dev.yml up -d  
$ docker exec -it node-docker-app-3\_mongo\_1 mongo -u "admin" -p "root"  
> db*

*> use mydb*

*> show dbs  
> db.books.find()  
> exit*

**- CAUTION**: You can **NOT** use the **-v** argument nor you can use the command *$docker volume prune* as both will **DELETE** your named and anonymous volumes.  
Unfortunately there seems not to be a convenient way to delete volumes that are no longer in use with a batch command. What you can do is delete the volumes one by one. Alternatively you could start all your containers with volumes you want to keep, and then call the *$ docker volume prune* command. Volumes of running containers are not touched by the prune command.  
*$ docker volume ls  
$ docker volume rm [VOLUME NAME]*

*$ docker volume prune*

**4. Connecting to the Mongo Database**

**-** Find out the IP-Address of the running Mongo DB container.  
*$ docker inspect node-docker-app-3\_mongo\_1*

**-** Now you could add the following code snippet to the **index.js**.

const mongoose = require("mongoose");

mongoose

.connect("mongodb://admin:root@[IP-Address]:27017/?authSource=admin")

.then(() => console.log("Successfully connected to Mongo DB."))

.catch((e) => console.log(e));

**-** Actually this is **NOT** a good idea, as each time the container starts a new IP-Address is assigned to that container. Fortunately Docker also creates a custom network with a DNS.  
*$ docker network ls*

**-** So better use this code snippet.

*const mongoose = require("mongoose");*

*const connectWithRetry = () => {*

*mongoose*

*.connect("mongodb://admin:root@mongo:27017/?authSource=admin")*

*.then(() => console.log("Successfully connected to Mongo DB."))*

*.catch((e) => {*

*console.log(e);*

*setTimeout(connectWithRetry, 5000);*

*});*

*}*

*connectWithRetry();*

**-** Via DNS the **service:** name **mongo** can be resolved to the current IP Address of that running container. **Also note**, that although there is a depends\_on: mongo within the node-app service in the docker-compose.yml, it is not guaranteed that the mongo server is up and running prior to the node-app container. It is only guaranteed that the mongo container is started first. So there is a recursive retry mechanism after 5 seconds for cases where the start up process of the mongo container takes longer than the start up process of the node-app container.

**-** Now stop and remove the containers and then restart them but with the **-d** argument so that we can see the express server output. Alternatively you could start the containers with the **-d**. To inspect the container console output you could use the *docker logs node-docker-app-3\_node-app\_1* command*.*  
*$ docker-compose -f docker-compose.yml -f docker-compose.dev.yml down*

*$ docker-compose -f docker-compose.yml -f docker-compose.dev.yml up  
$ docker-compose -f docker-compose.yml -f docker-compose.dev.yml down  
$ docker-compose -f docker-compose.yml -f docker-compose.dev.yml up -d  
$ docker logs node-docker-app-3\_node-app\_1*

**-** To demonstrate the DNS works also from within a Docker Container you could also bash into the express container and ping the mongo container or vice versa.  
*$ docker-compose -f docker-compose.yml -f docker-compose.dev.yml down*

*$ docker-compose -f docker-compose.yml -f docker-compose.dev.yml up -d  
$ docker logs node-docker-app-3\_node-app\_1*

*$ docker exec -it node-docker-app-3\_node-app\_1 bash  
$ ping mongo*

**SOURCES**

[https://docs.docker.com](https://docs.docker.com/get-started/overview/)

<https://www.docker.com/resources/what-container#/package_software>

<https://www.youtube.com/watch?v=9zUHg7xjIqQ&t=12811s&ab_channel=freeCodeCamp.org>

<https://www.capitalone.com/tech/cloud/containers-vs-vms/>

<https://spin.atomicobject.com/2019/07/11/docker-volumes-explained/>