

Full Stack Web Development

Advanced Topic

Job Connector Program

Outline

- Intro to Docker
- Docker image, volume and networks
- Dockerize your project
- Docker compose

What is Docker ?

Docker is a software platform that allows you to quickly build, test, and deploy applications. Docker packages software into standard units called containers that have everything the software needs to function including libraries, system tools, code, and runtime. Using Docker, you can quickly deploy and scale your app to any environment and have confidence that your code will run.



Intro to Docker – Docker vs VM

Docker

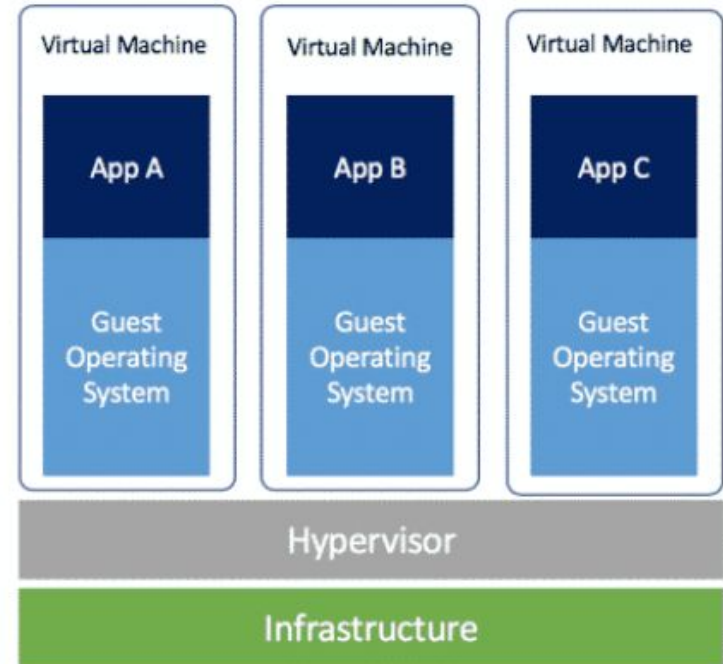
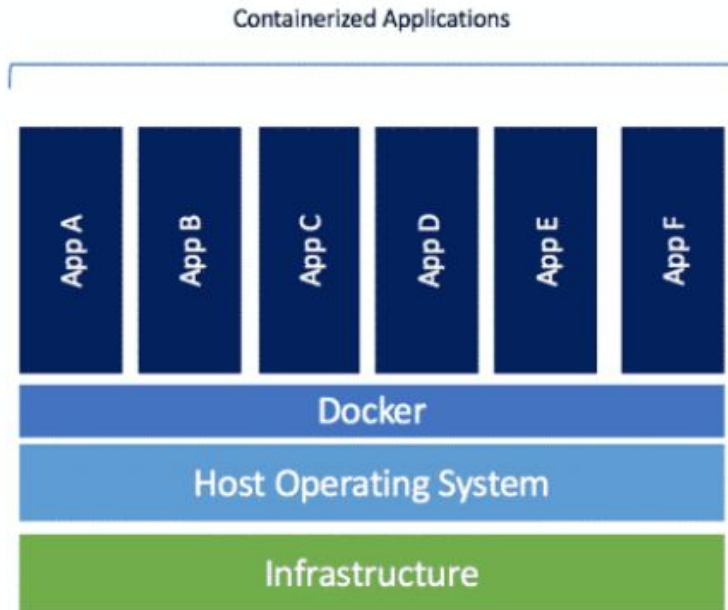
Allows developers to run applications that are owned in the form of containers on the operating system that is already installed on the server. So that on the server we no longer need to set up a VM, this is what causes Docker to be faster during the deployment process than a VM.

Virtual Machine

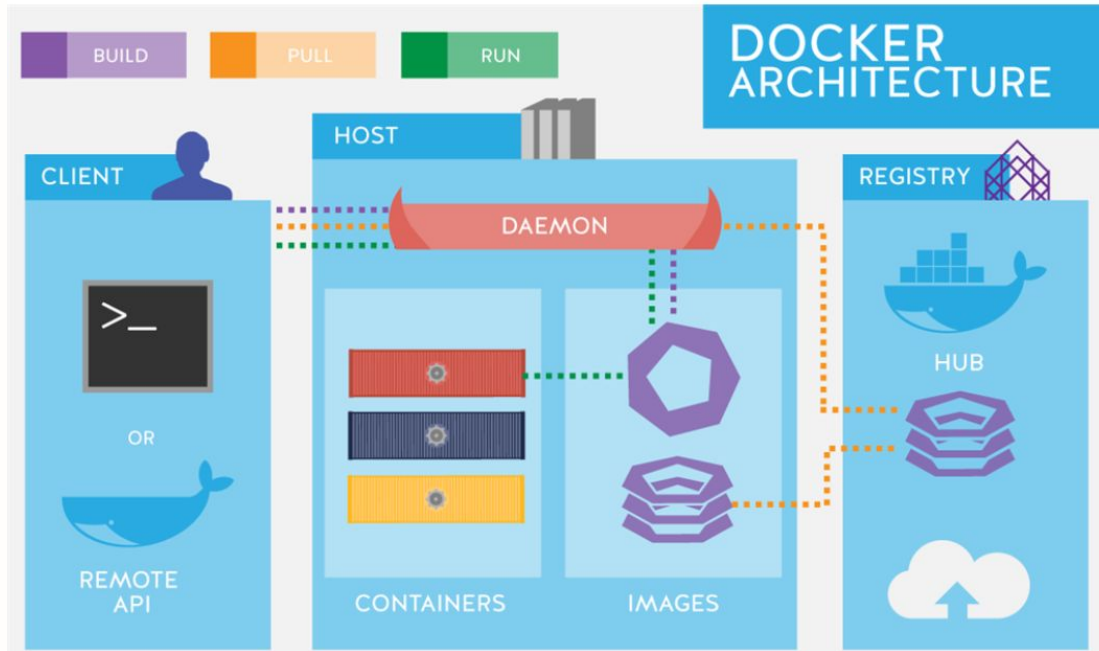
Every time we want to run an application on the server we have to prepare a virtual operating system first, and each application will usually be on a different virtual operating system on one computer. This is what causes it to take more time if we deploy the application to the VM.



Intro to Docker – Docker vs VM



Intro to Docker – Docker environment



Usually docker contains Docker Client, and Docker Server / Host.

Docker Client – contain terminal or command to execute Docker command

Docker Server – every request made by Docker Client, that would manage and executed using Docker Daemon.

There are several docker setup depends on operating system:

1. Windows: <https://docs.docker.com/desktop/windows/install/>
2. Linux: <https://docs.docker.com/engine/install/ubuntu/>
3. Mac: <https://docs.docker.com/desktop/mac/install/>



The Docker client provides a **command line interface (CLI)** that allows you to issue build, run, and stop application commands to a Docker daemon.

The main purpose of the Docker Client is to provide a means to direct the pull of images from a registry and to have it run on a Docker host.



Docker CLI – Common Commands

docker build

- Build an image from a Dockerfile

docker images

- List all images on a Docker host

docker run --name container_name images_name

- Run an image to create container

docker start container container_name

- Start existing container



Docker CLI – Common Commands

docker restart container container_name

- Restart existing container

docker rmi images_name

- Delete a local images

docker ps

- List all running

docker ps -a

- List all containers



Docker Daemon – In charge of managing images, either making changes or deleting. And also manage all the containers that are in docker.

Docker Host – The primary server that provides and runs the Docker Daemon.

Docker Registry – A place to store docker images, can be stored in the docker daemon for personal storage or stored in the Docker Hub so that it can be used by others.

Docker Objects

- Images Project templates that have been built into docker images.
- Networks Docker network that is used as a medium of communication between docker containers
- Containers Portable application that runs from docker images.

A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.

When you use the `docker pull` or `docker run` commands, the required images are pulled from your configured registry. When you use the `docker push` command, your image is pushed to your configured registry.

- **Docker Hub :** <https://hub.docker.com/>
- **Google Container Registry:** <https://cloud.google.com/container-registry>

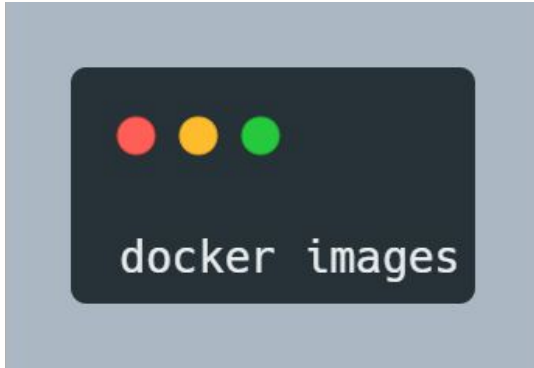
An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the ubuntu image, but installs the Apache web server and your application, as well as the configuration details needed to make your application run.

You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.



Docker Images

After successfully install docker desktop into your PC lets try to check your docker images on local PC by typing



Since your docker is freshly installed, it wont show any images. Lets add some!

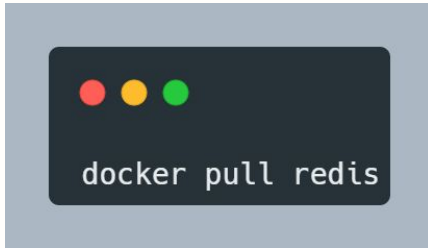
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
------------	-----	----------	---------	------



Docker Images

In this case, we will try to get an images from [docker hub](https://hub.docker.com/)

This time we will get redis images, write down the command in your terminal. After that lets run docker images once more. See the difference!



Docker Images

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
------------	-----	----------	---------	------

```
Using default tag: latest
latest: Pulling from library/redis
31b3f1ad4ce1: Pull complete
ff29a33e56fb: Pull complete
b230e0fd0bf5: Pull complete
9469c4ab3de7: Pull complete
6bd1cefcc7a5: Pull complete
610e362ffa50: Pull complete
Digest: sha256:b4e56cd71c74e379198b66b3db4dc415f42e8151a18da68d1b61f55fcc7af3e0
Status: Downloaded newer image for redis:latest
docker.io/library/redis:latest
```

You just successfully add redis image into your local docker server.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
redis	latest	9da089657551	7 days ago	117MB

A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

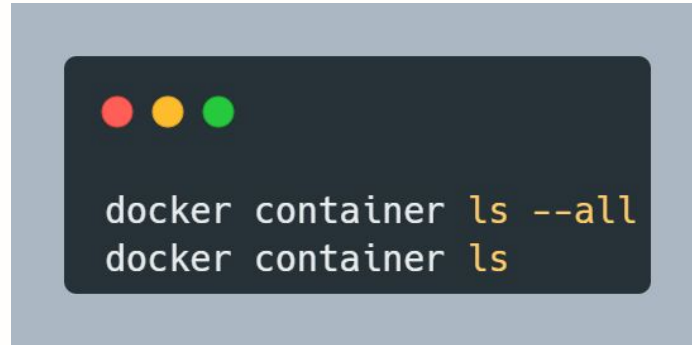
By default, a container is relatively well isolated from other containers and its host machine. You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.

A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.



Docker Container

Lets write down this command and run it on terminal. Since your docker is freshly installed it wont show anything



```
docker container ls --all
docker container ls
```

C:\Windows\system32\cmd.exe /c docker container ls							
CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES	




```
docker container create --name redis1 redis
```

In this case, we will try to create a container based on image we have. This is the create command for container, you can give a name to the container (in this case redis1) and the last part is your docker image which is redis. And you just successfully create a container.

Docker Container

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------



```
docker container ls --all
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
6232ad925681	redis	"docker-entrypoint.s..."	11 seconds ago	Created		redis1

Lets check our container list again and see the difference


After create the container, you can control the container (start, stop, delete). When it successfully start it will show the name of container.



```
docker container start redis1
```



```
docker container stop redis1
```



```
docker container create --name redis2 -p 8080:6379 redis
```

In order to access redis outside from the container, you can replace the port using custom port. In this case default port redis is 6379 in your container, but port 8080 would be used when you tried to use redis outside from the container.




```
docker container start redis2
```

Lets test your redis from outside the container. Run the docker container. Since we are using redis server, lets download redis client from [here](#), extract and locate redis directory and open in terminal. Set the port same as the one you expose before.



```
D:\installer\redis>redis-cli -p 8080
```



```
127.0.0.1:8080> set yourkey "we learn docker"
OK
127.0.0.1:8080> get yourkey
"we learn docker"
127.0.0.1:8080>
```

Try to run the redis. You just successfully accessing redis from docker container with custom port.

Dockerize Your Projects

```
index.js

const express = require("express");
const mysql = require("mysql2");

const app = express();

app.get("/", function (req, res) {
  res.send("hello world");
});

app.listen(3000);
console.log("listening on port 3000");
```

Create simple projects using nodejs and express and mysql. We would working on mysql later.

```
npm init --y
npm install express mysql2
```



Dockerize Your Projects

Create Dockerfile in your root of projects and write down this code.

This file contain several command that would be executed in docker to generate a docker image.

```
Dockerfile

FROM node:16

WORKDIR /app

COPY package.json .

RUN npm install

COPY . .

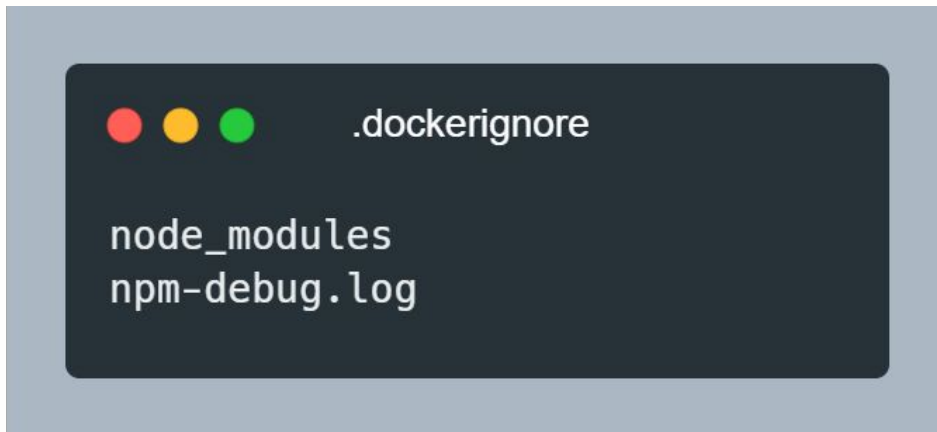
EXPOSE 3000

VOLUME [ "/app/node_modules" ]

CMD ["npm", "run", "dev"]
```

Dockerize Your Projects

Create .dockerignore file in your projects. Add this line of code into the file




```
.dockerignore

node_modules
npm-debug.log
```



Dockerize Your Projects

In the root of project (directory that has your Dockerfile) run the following command to build the Docker image. The `-t` flag lets you tag your image so it's easier to find later using the `docker images` command:



```
docker build -t node-simple-app:v1 .
```



Dockerize Your Projects

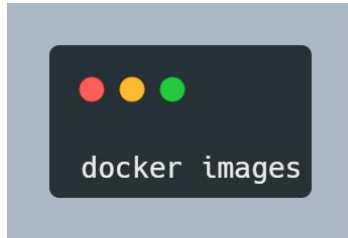
After executing the script, your terminal will execute the Dockerfile in your projects. You can read the log to make sure everything is run properly.

```
[+] Building 27.3s (11/11) FINISHED
=> [internal] load build definition from Docker 0.0s
=> => transferring dockerfile: 349B 0.0s
=> [internal] load .dockerignore 0.0s
=> => transferring context: 2B 0.0s
=> [internal] load metadata for docker.io/library/node:12-alpine3 4.6s
=> [auth] library/node:pull token for registry- 0.0s
=> [internal] load build context 0.9s
=> => transferring context: 2.06MB 0.9s
=> [1/5] FROM docker.io/library/node:12-alpine 16.0s
=> => resolve docker.io/library/node:12-alpine3 0.0s
=> => sha256:f9ec946db8888dba2b 1.43kB / 1.43kB 0.0s
=> => sha256:d44ce17eb41705155b 1.16kB / 1.16kB 0.0s
=> => sha256:09ab2aaab90b1cbc65 6.58kB / 6.58kB 0.0s
=> => sha256:8663204ce13b2961da 2.82MB / 2.82MB 4.1s
=> => sha256:4d92f3a2be1fe61 24.91MB / 24.91MB 14.5s
=> => sha256:55223db1bc3509c21b 2.36MB / 2.36MB 3.5s
=> => sha256:8978dce81359bfbad7476a 451B / 451B 3.9s
=> => extracting sha256:8663204ce13b2961da55026 0.2s
=> => extracting sha256:4d92f3a2be1fe6170a5de4d 1.1s
=> => extracting sha256:55223db1bc3509c21bb4284 0.1s
=> [2/5] WORKDIR /app 1.8s
=> [3/5] COPY package.json /app 0.0s
=> [4/5] RUN npm i && npm cache clean --force 4.4s
=> [5/5] COPY . /app 0.1s
=> exporting to image 0.1s
=> => exporting layers 0.1s
=> => writing image sha256:4cecf156a72f30d2d07a 0.0s
=> => naming to docker.io/library/simple-app 0.0s
```

Dockerize Your Projects

Check your docker images, and you will see the projects already generate images in your docker. You just successfully create images from nodejs projects.

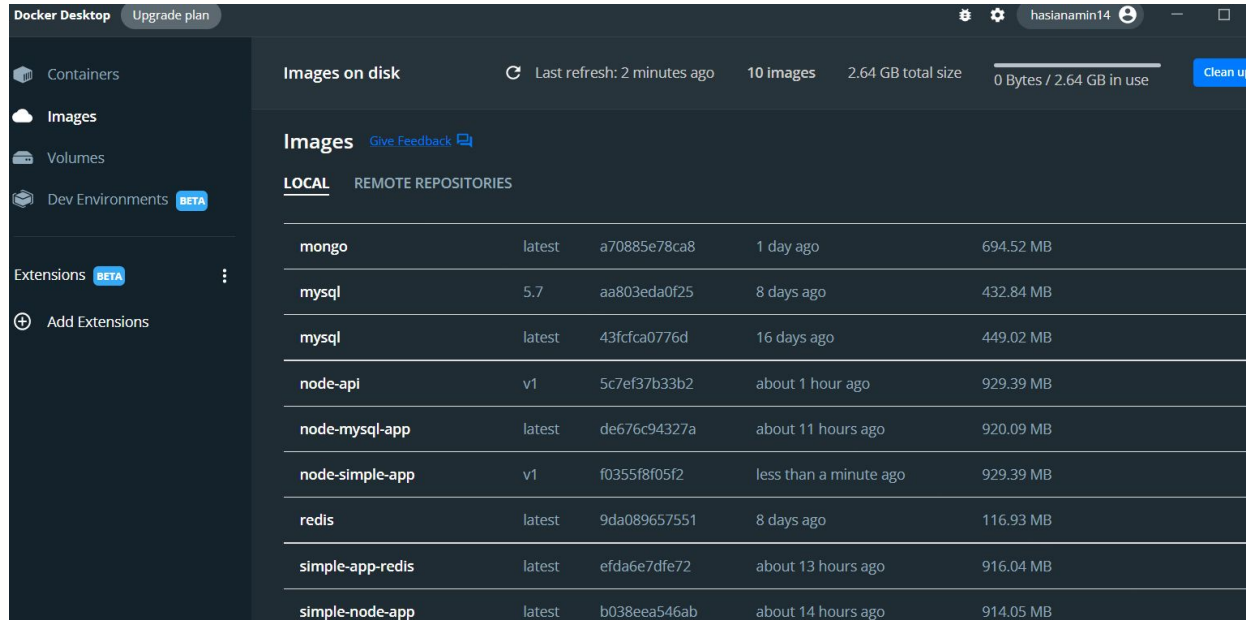
Run docker logs to check if your node-simple-app container running properly.



```
REPOSITORY          TAG          IMAGE ID          CREATED           SIZE
simple-node-app      latest      b038eea546ab     23 seconds ago   914MB
node-app            latest      4ad1ec968f3d     17 minutes ago   95.1MB
simple-app-2         latest      60eec3046bae     24 minutes ago   95.1MB
simple-app           latest      4cecf156a72f     52 minutes ago   95.1MB
mongo               latest      a70885e78ca8     16 hours ago     695MB
redis               latest      9da089657551     7 days ago       117MB
mysql               latest      43fcfca0776d     2 weeks ago      449MB
```

Dockerize Your Projects

Or you can check images from docker desktop from sidebar Images



Dockerize Your Projects

Create a new container based on image you just create. Run the container and check the logs to see if it works properly.



```
docker container create --name node-simple-app node-simple-app:v1  
docker container start node-simple-app
```



```
docker logs node-simple-app
```


Docker volumes are a widely used and useful tool for ensuring data persistence while working in containers. Docker volumes are file systems mounted on Docker containers to preserve data generated by the running container.

- The data doesn't persist when that container no longer exists, and it can be difficult to get the data out of the container if another process needs it.
- A container's writable layer is tightly coupled to the host machine where the container is running. The data cannot be easily moveable somewhere else.
- Writing into a container's writable layer requires a storage driver to manage the filesystem.


Docker Volume

```
$ docker volume create [OPTIONS] [VOLUME]
```

Command	Description
<code>docker volume create</code>	Create a volume
<code>docker volume inspect</code>	Display detailed information on one or more volumes
<code>docker volume ls</code>	List volumes
<code>docker volume prune</code>	Remove all unused local volumes
<code>docker volume rm</code>	Remove one or more volumes

Docker Volume

Lets pull mysql image from docker hub: https://hub.docker.com/_/mysql and create a new volume and name it as mysql and check if that volume is created successfully.



```
docker pull mysql
```



```
docker volume create mysql  
docker volume ls
```

Docker Volume

Create a new container based on mysql image, in this code we named it as mysql_server. There is -v argument, this used to define that our container would pointing into volume that has a name mysql.




```
docker container create --name mysql_server -v mysql:/var/lib/mysql mysql
```

Create a new container based on mysql image, in this code we named it as mysql_server. There is -v argument, this used to define that our container would pointing into volume that has a name mysql. After that run the mysql_server container.



```
docker container create --name mysql_server -v mysql:/var/lib/mysql mysql
docker container start mysql_server
```

Try accessing mysql from running container. By default we already have a root user in that mysql with no password. Lets create a new database.




```
docker exec -it mysql_server mysql -u root
```



```
create database testing_db;
```


Docker Volume

Lets check database that we just created before. After that, try to exit from mysql terminal. At this point, we will try to stop and remove our container with name mysql_server.



```
show databases;
```

```
mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| performance_schema |
| sys |
| testing_db |
+-----+
5 rows in set (0.00 sec)
```



```
docker container stop mysql_server
docker container rm mysql_server
```

After you remove the container try to create a new one container again but define the volume same as the last one. Start the container and check the list of database in this new container.



```
docker container create --name mysql_server2 -v mysql:/var/lib/mysql mysql
```



```
docker container start mysql_server2  
docker exec -it mysql_server2 mysql -u
```


Docker Volume

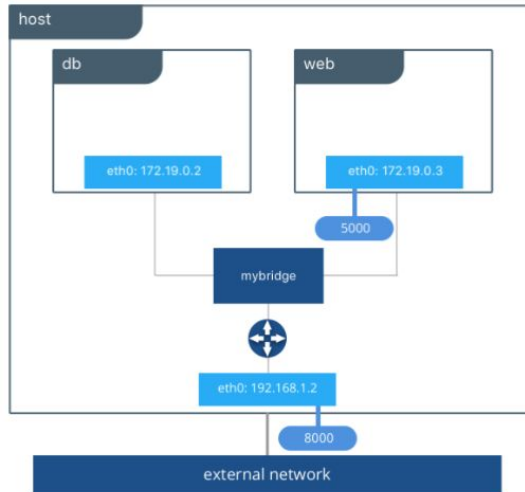
As you can see, it would keep the database same as mysql_server container before. Its caused by detach volume into this new container.



```
mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql          |
| performance_schema |
| sys            |
| testing_db     |
+-----+
5 rows in set (0.00 sec)
```

Docker Network

Networking is about communication among processes, and Docker's networking is no different. Docker networking is primarily used to establish communication between Docker containers and the outside world via the host machine where the Docker daemon is running.



\$ docker network COMMAND

Command	Description
<code>docker network connect</code>	Connect a container to a network
<code>docker network create</code>	Create a network
<code>docker network disconnect</code>	Disconnect a container from a network
<code>docker network inspect</code>	Display detailed information on one or more networks
<code>docker network ls</code>	List networks
<code>docker network prune</code>	Remove all unused networks
<code>docker network rm</code>	Remove one or more networks

In order to manage networks, you can use subcommands to create, inspect, list, remove, prune, connect, and disconnect networks.



Docker Network

Lets modify a bit our index.js

In this case, we will working on mysql into our projects. Write down the code to define the connection.

```
index.js

const express = require("express");
const mysql = require("mysql2");

const mysqlConfig = {
  host: "mysql_server",
  user: "testing",
  password: "secret",
  database: "test_db",
};

const app = express();

let con = null;

app.get("/connect", function (req, res) {
  con = mysql.createConnection(mysqlConfig);
  con.connect(function (err) {
    if (err) throw err;
    res.send("connected");
  });
});
```

Docker Network

Add some route to our projects. Lets add route to create a table name it as numbers table.

```
index.js

app.post("/number-table", function (req, res) {
  db.connect(function (err) {
    if (err) throw err;
    const sql = `
      CREATE TABLE IF NOT EXISTS numbers (
        id INT AUTO_INCREMENT PRIMARY KEY,
        number INT NOT NULL,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
      ) ENGINE=INNODB;
    `;
    db.query(sql, function (err, result) {
      if (err) throw err;
      res.send("numbers table created");
    });
  });
});
```

Docker Network

And create route to insert and retrieve data from mysql database. And dont forget to set port to listen our projects.

```
index.js


app.post("/number", function (req, res) {
  const number = Math.round(Math.random() * 100);
  db.connect(function (err) {
    if (err) throw err;
    const sql = `INSERT INTO numbers (number) VALUES
    (${number})`;
    db.query(sql, function (err, result) {
      if (err) throw err;
      res.send(`${number} inserted into table`);
    });
  });
});

app.get("/number", function (req, res) {
  db.connect(function (err) {
    if (err) throw err;
    const sql = `SELECT * FROM numbers`;
    db.query(sql, function (err, result, fields) {
      if (err) throw err;
      res.send(JSON.stringify(result));
    });
  });
});

app.listen(3000);
console.log("listening on port 3000");
```

Docker Network

Lets create a new network in our docker. Write down this command to create and check if that network created successfully.

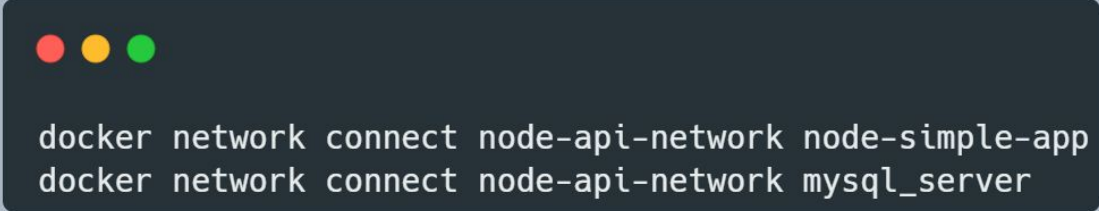


```
docker network create node-api-network  
dcoker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
2c108776517e	be-sample_default	bridge	local
c3e7fb539144	bridge	bridge	local
795bdc01fc06	host	host	local
001479a01cd3	network-simple	bridge	local
dff78aaa1bf7	node-api-network	bridge	local
62c7e8e1f459	none	null	local
899324ac7e9f	stable-network	bridge	local


Add container name that you would like to connect into network you just made before.

Lets connect node-simple-app and mysql_server into the same network.



```
docker network connect node-api-network node-simple-app  
docker network connect node-api-network mysql_server
```


Lets check if the container we just add is successfully connect with the network. Check each container using inspect argument.



```
docker container inspect mysql_server
docker container inspect node-simple-app
```

```
},
"node-api-network": {
  "IPAMConfig": {},
  "Links": null,
  "Aliases": [
    "532ad33b8fc7"
  ],
  "NetworkID": "",
  "EndpointID": "",
  "Gateway": "",
  "IPAddress": "",
  "IPPrefixLen": 0,
  "IPv6Gateway": "",
  "GlobalIPv6Address": "",
  "GlobalIPv6PrefixLen": 0,
  "MacAddress": "",
  "DriverOpts": {}
}
```

Compose is a tool for defining and running multi-container Docker applications. 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.

Compose works in all environments: production, staging, development, testing, as well as CI workflows.



Using Compose is basically a three-step process:

- Define your app's environment with a Dockerfile so it can be reproduced anywhere.
- Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.
- Run docker compose up and the Docker compose command starts and runs your entire app. You can alternatively run docker-compose up using the docker-compose binary.



Compose has commands for managing the whole lifecycle of your application:

- Start, stop, and rebuild services
- View the status of running services
- Stream the log output of running services
- Run a one-off command on a service



Docker Compose – Compose Specification

The Compose file is a YAML file defining services, networks, and volumes for a Docker application. The latest and recommended version of the Compose file format is defined by the Compose Specification.

This document specifies the Compose file format used to define multi-containers applications. Distribution of this document is unlimited.



The Compose file is a YAML file defining:

- Version (DEPRECATED)
- Services (REQUIRED)
- Networks, volumes, configs and secrets.

The default path for a Compose file is `compose.yaml` (preferred) or `compose.yml` in working directory. Compose implementations SHOULD also support `docker-compose.yaml` and `docker-compose.yml` for backward compatibility. If both files exist, Compose implementations MUST prefer canonical `compose.yaml` one.



Docker Compose – Compose File

Lets go back to our projects, this time we will implement starting container node app and mysql using docker compose. Lets create file name as docker-compose.yaml and put this code into your file.

```
docker-compose.yaml

version: "3.8"

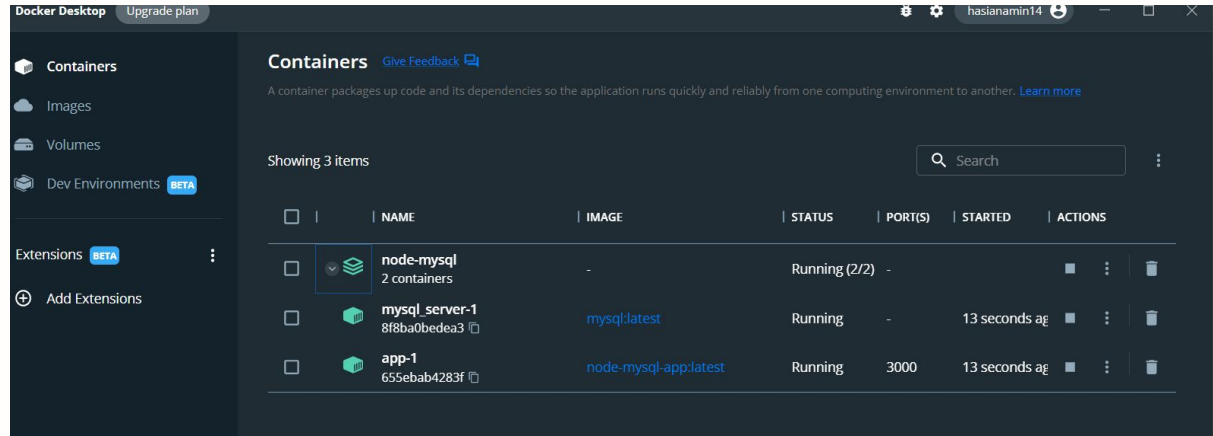
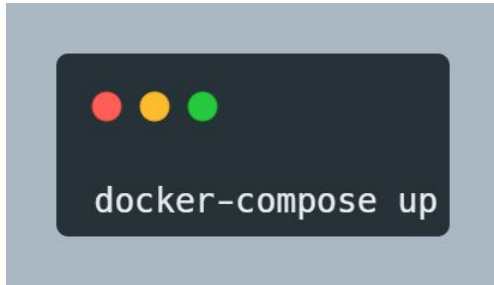
networks:
  node-api-network:
    name: node-api-network

services:
  app:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "3000:3000"
    volumes:
      - ./app
      - /app/node_modules
    depends_on:
      - mysql_server
    networks:
      - node-api-network

  mysql_server:
    image: mysql
    environment:
      - MYSQL_DATABASE=test_db
      - MYSQL_USER=testing
      - MYSQL_PASSWORD=secret
      - MYSQL_ROOT_PASSWORD=secret
    networks:
      - node-api-network
```

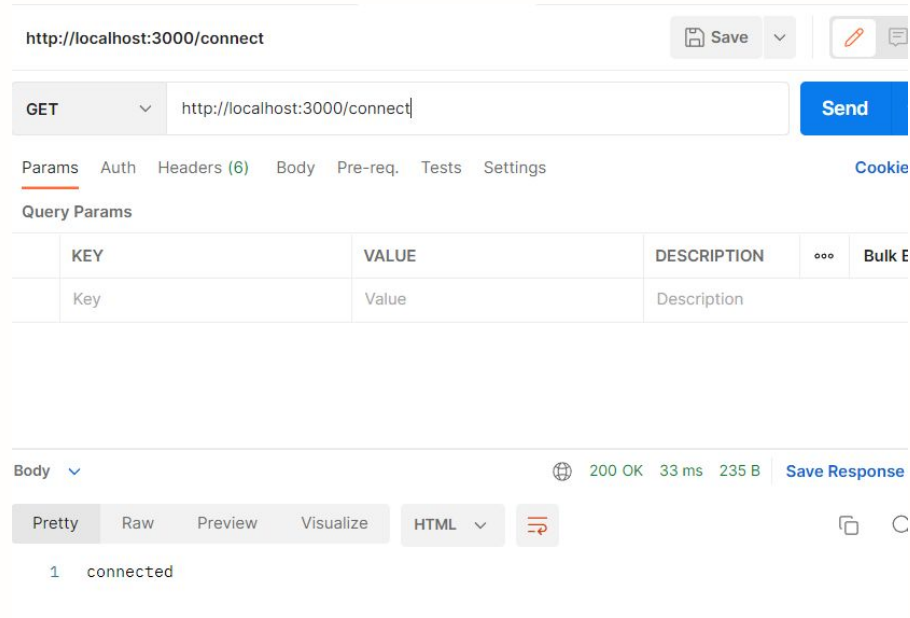
Docker Compose – Compose File

Lets run your container, use docker-compose command to take an action for docker-compose.yaml file. You can check through docker desktop for running container. But this time, there is more than one container running and wrapped by one container



Docker Compose – Compose File

Lets try and test our projects through postman!



http://localhost:3000/connect

GET http://localhost:3000/connect

Params Auth Headers (6) Body Pre-req Tests Settings

Query Params

KEY	VALUE	DESCRIPTION	...	Bulk E
Key	Value	Description		

Body 200 OK 33 ms 235 B Save Response

Pretty Raw Preview Visualize HTML

1 connected

Thank You!

