सी-डैक
CDAC
Gist

# GIST Docker Deployment

**Presentation**

docker

# Introduction

Docker is a software platform that allows you to build, test, and deploy applications quickly.

Running Docker on AWS provides developers and admins a highly reliable, low-cost way to build, ship, and run distributed applications at any scale.

# Why Docker?

Containers are nothing but small VMs

- Enables Efficient use of system resources
- Enables faster software delivery cycles
- Enables portable applications

# Docker Won’t

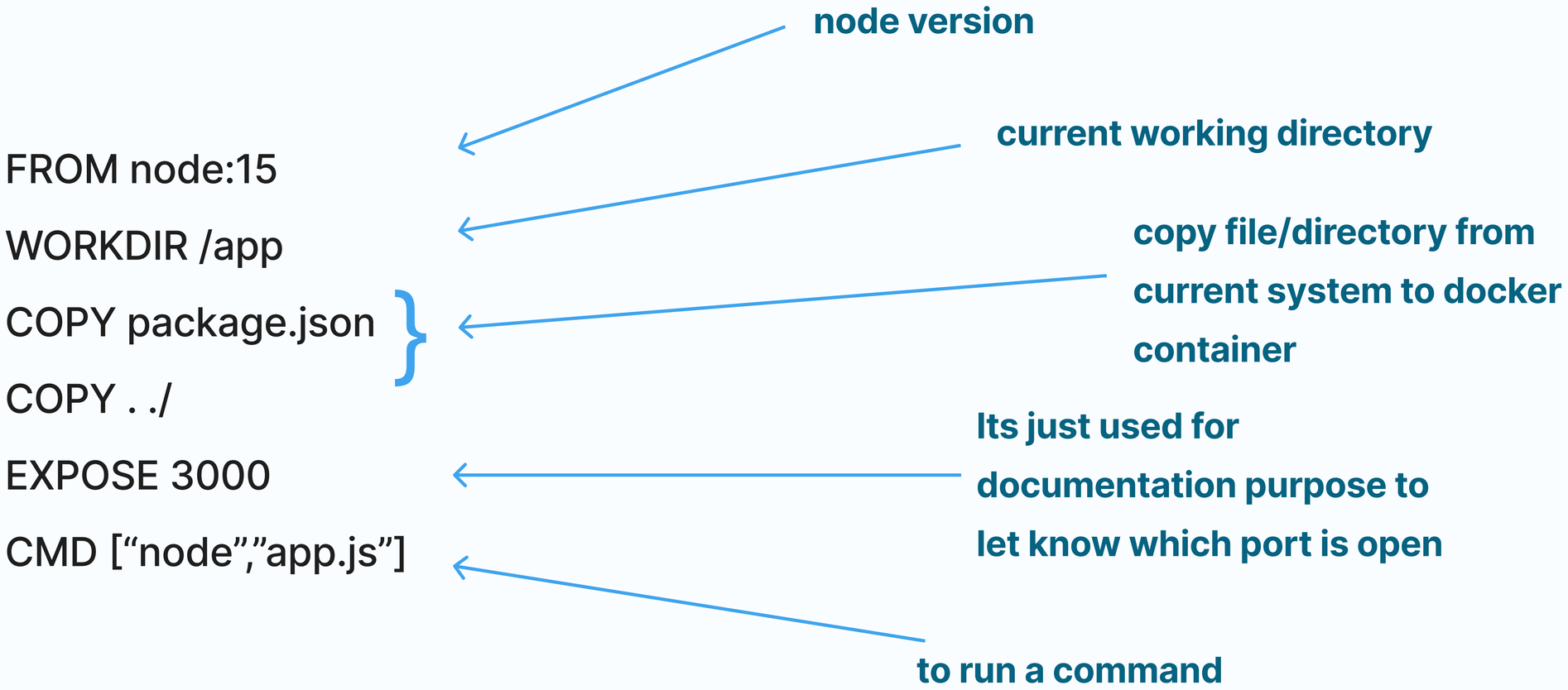
Solve the security issue ( it has to be application specific and we have actually mentioned it further in the presentation)

# Creating a Dockerfile

Dockerfile is a step by step process for performing the necessary step to build the application on docker container.Dockerfile conatins step to create a image for your application.

A Docker Image is a read-only file with a bunch of instructions. When these instructions are executed, it creates a Docker container.

# Example:


FROM node:15
WORKDIR /app
COPY package.json
COPY . ./
EXPOSE 3000
CMD [“node”,”app.js”]
node version
current working directory
copy file/directory from current system to docker container
Its just used for documentation purpose to let know which port is open
to run a command

# Building a docker Image

```
> docker build
```

on running this command it will create find the Dockerfile in the current directory and start executing all steps one by one .

FROM node:15 will actually fetch the offical node image from docker.io/library/node:15

After that it will create a workdir /app and it makes it the present working directory for us
Keep in mind that Docker actually cache the steps when we first build the image and the next time we build the image it will only execute the steps that are changed.

# Docker Basic Commands

```
> docker image ls
> docker ps
> docker run -d node-app-image
> docker run -d --name node-app node-app-image
> docker rm -image_name
> docker run -p 3000:3000 -d --name node-app node-app-image
```

## Remove a container using the CLI

1. Get the ID of the container by using the `docker ps` command.

```
$ docker ps
```

2. Use the `docker stop` command to stop the container.

```
# Swap out <the-container-id> with the ID from docker ps
$ docker stop <the-container-id>
```

3. Once the container has stopped, you can remove it by using the `docker rm` command.

```
$ docker rm <the-container-id>
```

> **Note**
>
> You can stop and remove a container in a single command by adding the "force" flag to the `docker rm` command. For example:
> `docker rm -f <the-container-id>`

# Displaying Docker Images

To see the list of Docker images on the system, you can issue the following command.

```
docker images
```

This command is used to display all the images currently installed on the system.

## Syntax

```
docker images
```

**TAG –** This is used to logically tag images.

**Image ID –** This is used to uniquely identify the image.

**Created –** The number of days since the image was created.

**Virtual Size –** The size of the image.

# Removing Docker Images

The Docker images on the system can be removed via the docker rmi command. Let's look at this command in more detail.

```
docker rmi
```

This command is used to remove docker images

## Syntax

```
docker rmi Image ID
```

## Options

**Image ID** – This is the ID of the image which needs to be removed

# docker images -q

This command us used to return only the Image ID’s of the images.

## Syntax

```
docker images
```

## Options

**q** – It tells the Docker command to return the Image ID’s only.

## Return Value

The output will show only the Image ID’s of the images on the Docker host.

## Example

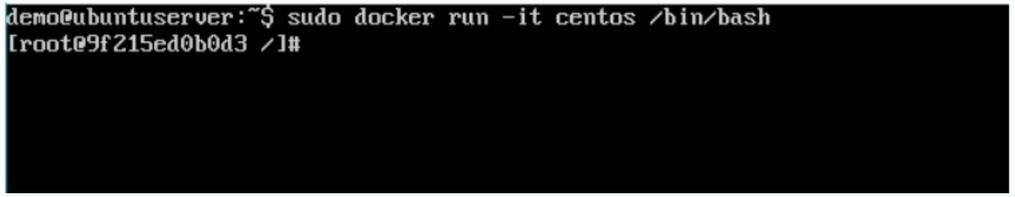
```
sudo docker images -q
```

# Running a Container

Running of containers is managed with the Docker run command. To run a container in an interactive mode, first launch the Docker container.

```
sudo docker run -it centos /bin/hash
```

Then hit Crtl+p and you will return to your OS shell.




You will then be running in the instance of the CentOS system on the Ubuntu server.

## Listing of Containers

One can list all of the containers on the machine via the docker ps command.
This command is used to return the currently running containers.

```
docker ps
```

Containers are instances of Docker images that can be run using the Docker run command. The basic purpose of Docker is to run containers. Let’s discuss how to work with containers.

# docker ps -a

This command is used to list all of the containers on the system

## Syntax

```
docker ps -a
```

## Options

**a –** It tells the docker ps command to list all of the containers on the system.

## Return Value

The output will show all containers.

## Example

```
sudo docker ps -a
```

# docker history

With this command, you can see all the commands that were run with an image via a container.

## Syntax

```
docker history ImageID
```

## Options

**ImageID** – This is the Image ID for which you want to see all the commands that were run against it

# Building the Docker File

We created our Docker File in the last chapter. It's now time to build the Docker File. The Docker File can be built with the following command –

```
docker build
```

Let's learn more about this command.

## docker build

This method allows the users to build their own Docker images.

## Syntax

```
docker build  -t ImageName:TagName dir
```

## Rebuilding the Image

```
-docker build -t node-app-image .
```

## Options

**-t –** is to mention a tag to the image

**ImageName** – This is the name you want to give to your image.

**TagName** – This is the tag you want to give to your image.

**Dir** – The directory where the Docker File is present.

# Expose your Image Port

## Port Expose Command

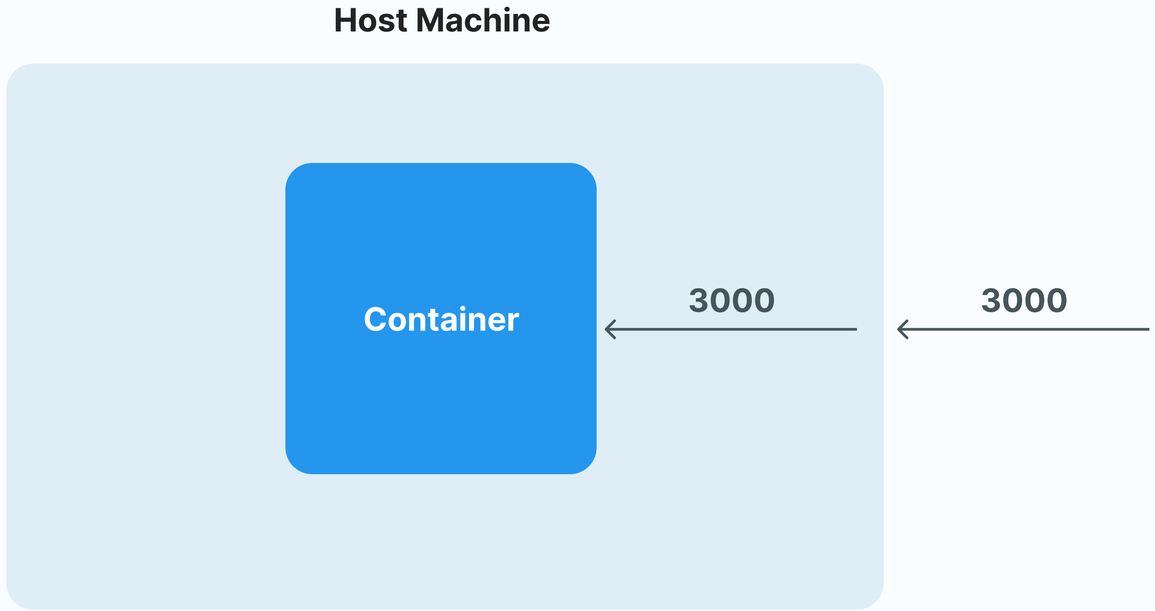
```
docker run -p 3000:3000 -d --name node-app node-app-image
```

## --name

this command assigns a name to the container

## Getting Inside Docker Container

```
docker exec -ti node-app bash
```



# Bind Volumes

This actually binds the Current Specified Directory to the docker directory so any changes that happens inside this directory will reflect in the docker contianer specified directory

```
docker run -v path-of-folder-in-local-machine:path-of-folder-on-Docker-container ^
-p 3000:3000 ^
-d --name node-app node-app-image
```

```
docker run -v %cd%:/app -p 3000:3000 -d --name  node-app node-app-image
```

BIND MOUNT IS ONLY FOR DEVELOPMENT PROCESS

# Creation Volumes to persist Data

```
docker run -v %cd%:/app -v /app/node_modules -p 3000:3000 -d --name  node-app node-app-image
```

## Read Only Container Volume

```
docker run -v %cd%:/app:ro -p 3000:3000 -d --name  node-app node-app-image
```

## To view created volumes we can use command

```
docker volume ls
```

## Deleting a volume

```
docker volume rm IMAGE-ID
```

```
docker volume prune
```

# Docker Compose File

Services are just the images that we want to build or run
we can run multiple services inside one compose file

This way you can run multiple containers with one file

```
version: "3"
services:
  node-app:
    build: .
    ports:
      - "3000:3000"
    volumes:
      - .:/app:ro
      - /app/node_modules
    environment:
      - PORT=3000
    # env_file:
    #   - ./.env
```

NOTE:with yml file spacing/identaiton matters

```
-docker-compose up -d
```

## Stopping containter with docker-compose down

```
-docker-compose down -v
```

## Re-Building Images with Docker-compose File

```
-docker-compose up -d --build
```

# A SHARED COMPOSE FILE BETWEEN ALL DOCKER_COMPOSE FILES

docker-compose.yml file

```
version: '3'
services:
  node-app:
    build: .
    ports:
      - "3000:3000"
    environment:
      - PORT=3000
```

```
version: '3'
services:
  node-app:
    volumes:
      - ./:/app
      - /app/node_modules
    environment:
      - NODE_ENV=development
    command: npm run dev
```

>docker-compose.prod.yml

```
version: '3'
services:
  node-app:
    environment:
      - NODE_ENV=production
    command: node index.js
```

```
docker-compose -f docker-compose.yml -f docker-compose.dev.yml up -d
```

This are the settings that are gonna be shared between all compose files

# Docker-compose for Production

Docker compose for production should be updated code that have
be lastly changed in development server that we want to serve
so we should run --build flag to rebuild with latest updated code

```
docker-compose -f docker-compose.yml -f docker-compose.dev.yml up -d --build
```

## Stopping Container and removing volume associated with

```
docker-compose -f docker-compose.yml -f docker-compose.prod.yml down -v
```

# Script to check environment variable and run command

**Updating Docker file to RUN for development Mode and Production Mode only By PASSING NODE_ENV enviornment variable and checking with a small script**

```
FROM node:15
WORKDIR /app
COPY package.json .
RUN npm install

RUN if [ "$NODE_ENV" = "development" ]; \
        then npm install; \
        else npm install --only=produnction; \
        fi

COPY . ./
ENV PORT 3000
EXPOSE $PORT
CMD ["node","index.js"]
```

# Updating compose file for Deployment and Production mode

```
>docker-compose.yml
version: '3'
services:
  node-app:
    build:
      context: .
      args:
        NODE_ENV: development
    ports:
      - "3000:3000"
    environment:
```

```
>docker-compose.prod.yml
version: '3'
services:
  node-app:
    build:
      context: .
      args:
        NODE_ENV: production
    environment:
      - NODE_ENV=production
    command: node index.js
```

```
>docker-compose.dev.yml
version: '3'
services:
  node-app:
    volumes:
      - ./:/app
      - /app/node_modules
    environment:
      - NODE_ENV=development
    command: npm run dev
```