Introduction to Containerization & Orchestration

Docker

What is Containerization?

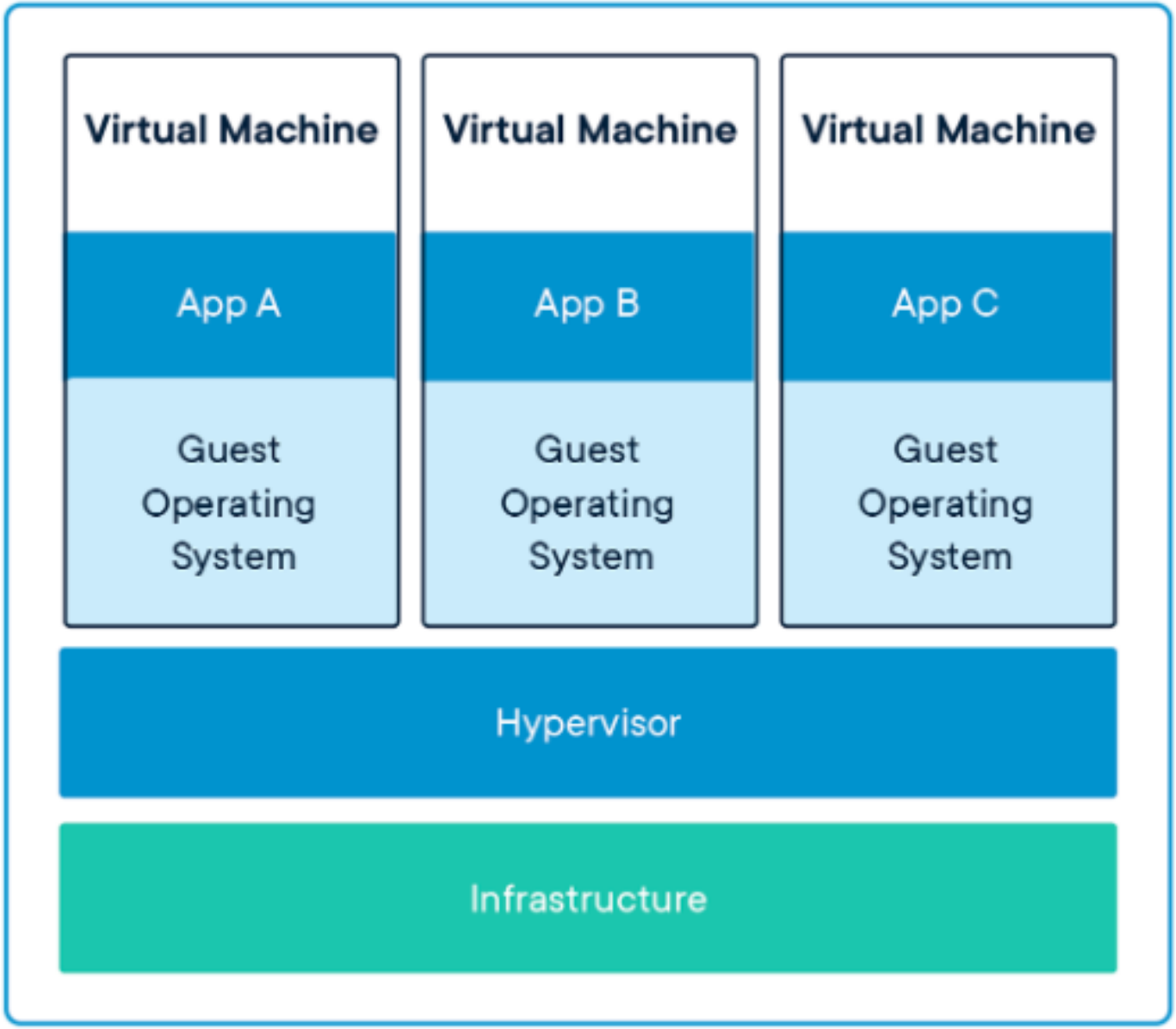
- Containerization is a lightweight form of virtualization that allows developers to package and run applications along with their dependencies in isolated environments called containers. Unlike traditional virtual machines, containers share the host system's operating system kernel but maintain isolated user spaces.

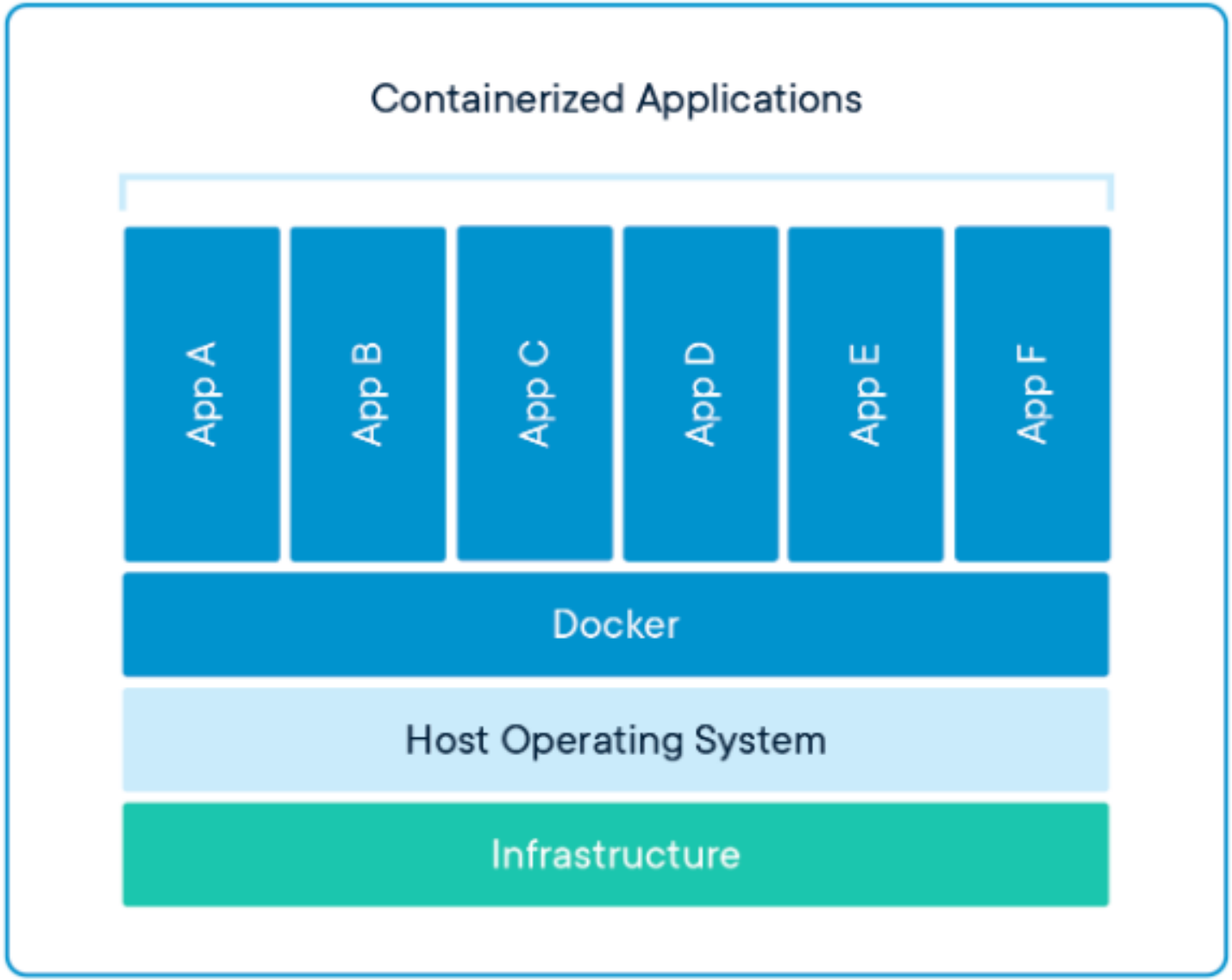
Key Concepts

- **Containers:**
  - Encapsulated environments that include the application, its dependencies, libraries, and configuration files.
  - They ensure that the application runs consistently regardless of the environment.
- **Docker:**
  - The most popular platform for creating, deploying, and managing containers.
  - It provides tools to package applications into containers, distribute them, and run them on any system with Docker installed.

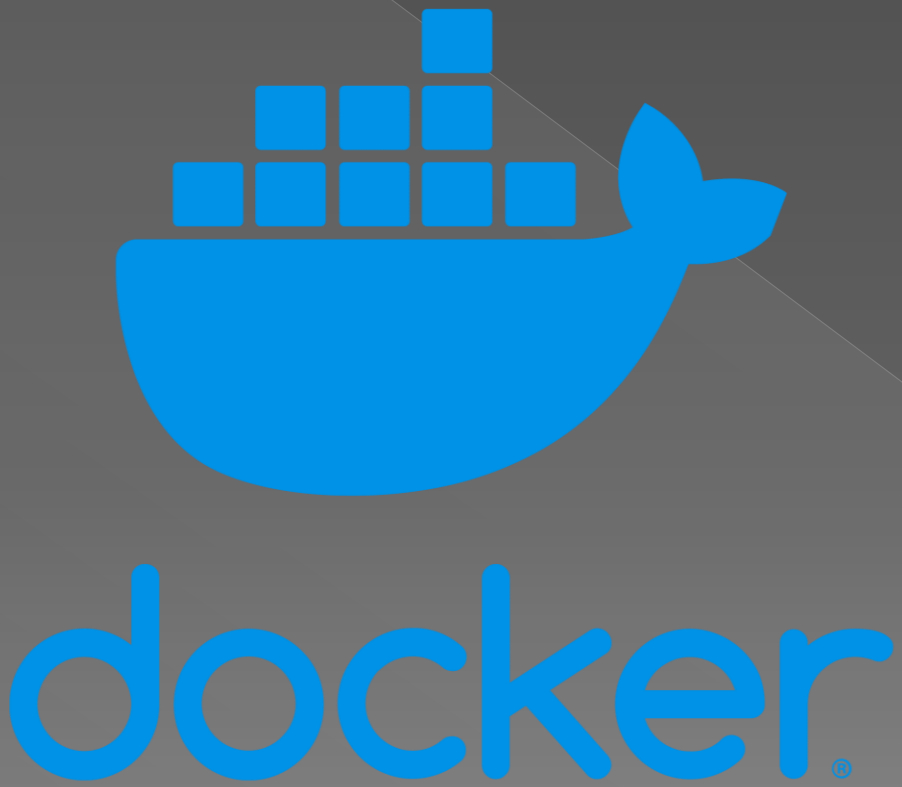
Benefits of Containerization:

- Portability
- Scalability
- Isolation
- Efficiency


Virtual Machine
App A
Guest Operating System
Virtual Machine
App B
Guest Operating System
Virtual Machine
App C
Guest Operating System
Hypervisor
Infrastructure


Containerized Applications
App A
App B
App C
App D
App E
App F
Docker
Host Operating System
Infrastructure

What is Docker



- **Docker** is an open-source platform that automates the deployment, scaling, and management of applications inside lightweight, portable containers.

How Docker Works?

- **Create a Dockerfile:** Developers write a Dockerfile with instructions on how to set up the environment and run the application.
- **Build an Image:** Use the Docker CLI to build an image from the Dockerfile using the command docker build.
- **Run a Container:** Use the built image to run a container using the command docker run. This starts an instance of the image and runs the application inside it.
- **Manage Containers:** Use Docker CLI commands to manage running containers, such as starting, stopping, and removing containers.

Docker Terms

- **Docker Engine:**
  - The core component of Docker, consisting of a server & a REST API for interacting with the daemon, as well as a command-line interface (CLI) client.
- **Docker Images:**
  - Read-only templates that contain the application & all its dependencies, libraries, binaries, and other files necessary to run the application.
  - An image is built from a Dockerfile.

Docker Terms

- **Docker Containers:**
  - Instances of Docker images.
  - Containers are lightweight, portable, and isolated environments that run applications.
  - They share the host system's OS kernel but maintain their own filesystem, processes, and network interfaces.

Docker Terms

- **Dockerfile:**
  - A text file that contains a series of instructions on how to build a Docker image.
  - It defines what goes on in the environment inside your container.
- **Docker Hub:**
  - A cloud-based repository where Docker users and partners create, test, store, and distribute Docker images.
  - It is the default registry for Docker images.

Installing Docker Client

- Get the windows installer from the official docker website.
- Docker desktop: https://www.docker.com/products/docker-desktop/
- Double click and you can complete installer.

Activity 1

- Docker installation
- Check docker version
  - sudo docker version
  - sudo docker –v / sudo docker --version

Activity 2 commands

- Docker Containers
  - sudo docker container ls
- Docker Images:
  - sudo docker images
- Docker Volume
  - sudo docker volume ls

Activity 3: Pull images from Docker

- Step 1: pull image
  - sudo docker pull docker/getting-started
- Step 2: Check Image pulled
  - sudo docker images
- Step 3: Run the image in docker container with some specific PORT Number
  - sudo docker run -p 80:80 docker/getting-started
- Step 4: Exit the container
  - exit or ctrl+c
- Step 5: Check the status of Running container
  - sudo docker ps –a
- Step 6: Detach The container (run in detached mode)
  - sudo docker run -d -p 80:80 docker/getting-started
- step 7: Check the status of Container
  - sudo docker container ps –a
- check the output in the browser type localhost and you can see the output of getting started container

- Step 9: Stop Container
  - check all the running container and get the name of you container which you want to stop
  - docker container ls
  - docker stop container_name
- Step 10: Container Removal
  - docker ps –a
  - docker rm name_of_container
  - sudo docker rm -f name_of_container (use -f flag to remove container forcefully)

- Giving Own name to container
  - sudo docker run -d -p 80:80 --name my_getting_started_app docker/getting-started
  - Check running container
  - Stop the same
  - Remove the same
- Step 11: Remove Images
  - sudo docker images (get the image name which you want to remove)
  - sudo docker rmi name-of-image
  - sudo docker rmi -f name-of-image
- (after removal check again the image removed or not) – docker images

Activity 4: Pull the ready images from docker hub

- sudo docker pull ubuntu
- sudo docker pull mysql
- sudo docker pull mysql:5.7 (pull with version)
- check images
  - sudo docker images
- Remove 5.7 version image
  - sudo docker rmi mysql:5.7

- Let's run mysql image in docker container
- sudo docker run --name my_mysql_container -e MYSQL_ROOT_PASSWORD=123456 -d mysql
- sudo docker container ls (see running container)
- Connect with mysql
  - sudo docker exec -it my_mysql_container mysql -uroot –p
  - (enter your secret password and here we go you are connected with mysql)
  - create some queries and check.

Docker File

- For Creating the Image we need to understand Docker file
- its a file which doesn't contain any extention and its having some instructions written inside the same for creating/building a docker Image.
- These instruction steps define steps needed to create an environment where you can run your application.

DockerFile terms

- **FROM**
  - Specifies the base image to use for the subsequent instructions.
  - Every Dockerfile must start with a FROM instruction.
  - FROM ubuntu:20.04
- **RUN**
  - Executes commands in a new layer on top of the current image and commits the results.
  - Commonly used to install software packages.
  - RUN apt-get update && apt-get install -y python3

DockerFile terms

- **CMD**
  - Provides defaults for an executing container.
  - There can only be one CMD instruction in a Dockerfile. If multiple CMD instructions are provided, only the last one will be used.
  - CMD ["python3", "app.py"]
- **LABEL**
  - Adds metadata to the image. Can be used for description, versioning, authorship, and other useful information.
  - LABEL maintainer="Sonam Soni <sonam@gmail.com>"

DockerFile terms

- **EXPOSE**
  - Informs Docker that the container listens on the specified network ports at runtime.
  - EXPOSE 8080
- **ENV**
  - Sets environment variables inside the container.
  - ENV APP_HOME /usr/src/app
- **ADD**
  - Copies files, directories, or remote file URLs from the source to the destination path in the image.
  - ADD config.tar.gz /etc/config/

DockerFile terms

- **COPY**
  - Similar to ADD, but only supports copying local files and directories.
  - COPY . /app
- **ENTRYPOINT**
  - Configures a container that will run as an executable.
  - ENTRYPOINT ["python3", "app.py"]
- **VOLUME**
  - Creates a mount point with the specified path and marks it as holding externally mounted volumes from the host or other containers.
  - VOLUME /data

DockerFile terms

- **WORKDIR**
  - Sets the working directory for any RUN, CMD, ENTRYPOINT, COPY, and ADD instructions that follow it in the Dockerfile.
  - WORKDIR /app
- **USER**
  - Sets the user name or UID to use when running the image and for any RUN, CMD, and ENTRYPOINT instructions that follow it.
  - USER appuser
- **ARG**
  - Defines a variable that users can pass at build-time to the builder with the docker build command.
  - ARG build_version
  - RUN echo "Building version $build_version"

Let’s Create one Website & Dockerfile

- Create one folder named mindsprint
- Create index.html with some sample code
- Create dockerfile without any extension in that.
- Code Structure
  - mindsprint /
    - Dockerfile
    - index.html

Docker file Code

```
# Use the official Nginx image from Docker Hub
FROM nginx:alpine

# Copy the HTML file to the default Nginx public directory
COPY index.html /usr/share/nginx/html

# Expose port 80 to the host
EXPOSE 80
```

Build an Image and run as Container

- docker build -t mindsprint
- docker run -d -p 8080:80 mindsprint
- http://localhost:8080 check your deployed website

Activity

- Create one another website for Banglore-tourism
- Deploy it on live server using apache server.
- Create docker file
- Build image
- Check the created images
- Run it as container and check localhost output

Dockerfile for apache

```
# Use the official Apache image from Docker Hub
FROM httpd:alpine

# Copy the HTML file to the default Apache public directory
COPY index.html /usr/local/apache2/htdocs/

# Expose port 80 to the host
EXPOSE 80
```

More than 1 file

- #Copy the entire content of the local directory to the default Apache public directory
- COPY . /usr/local/apache2/htdocs/
- (. Represent an entire folder to copy into htdocs folder)

Backend App

- Create one NodeJs application
  - App.js
  - Package.json
  - Install express package (npm install)

App.js file

```
const fs = require('fs');
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
    res.send('Hello, world!.');
});

app.listen(port, () => {
    console.log(`App running on http://localhost:${port}`);
});
```

Package.json file

```
{
  "name": "app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  ▷ Debug
  "scripts": {
    "start": "node app.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.19.2"
  }
}
```

DockerFile

```
# Use the official Node.js image
FROM node:alpine

# Create and set the working directory
WORKDIR /app

# Copy package.json and package-lock.json
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy the application files
COPY . .

# Expose port 3000
EXPOSE 3000

# Start the application
CMD ["npm", "start"]
```

Deployment

- Create an image
- Deploy it on a container with port 3000
- docker build -t docker-node-app .
- docker run -d -p 3000:3000 -v docker-node-app

Create Account to DockerHub

- Open hub.docker.com
- Create an account
- let’s connect from docker
- sudo docker login
  - prompt with username: enter your docker username
  - password: type your password (The password which you type is not visible for security reasons) just press enter
  - If all good you will connected with your docker hub.

Push Image to docker hub

- Create Tag to your Docker Images
  - sudo docker tag imagename sonamsoni/myimage
- Push Image to your Docker account
  - sudo docker push sonamsoni/myimage
- Once its uploaded check docker hub to see the image uploaded

Docker Volumes

- Volumes are the storage, where we can persist our data generated by docker containers.
- Using that we can share data between containers and also between container and host.
- Docker Volumes are stored outside the container's filesystem so if you want to see your data after your container stops or remove you can check the same.

- Let's create one Volume
  - sudo docker volume create my_volume
- Let's run image in one container with volume
  - sudo docker run -d --name my_container -v my_volume:/data imagename

- container created with Name: my_container
- attached with volume: my_volume
- stores the containers data inside: /data directory
- Get List of Volumes
  - sudo docker volume ls
- Inspect a perticular volume
  - sudo docker inspect my_volume (my_volume is the volume name)

Container commands

- See exited container:
  - docker ps -a --filter status=exited
- Remove all exited containers
  - docker container prune
- Above command prompt with confirmation
- To bypass that confirmation use
  - docker container prune -f

Task 1

- **Run a Hello World Container:**
- Task: Pull and run the official Docker "Hello World" container.
- Steps:
  - Use docker pull hello-world to pull the image.
  - Use docker run hello-world to run the container.
  - Observe the output to understand the basic Docker container lifecycle.

Task 2:Build and Run a Customized Nginx Server:

- **Build and Run a Customized Nginx Server:**
- Task: Create a Dockerfile to build a custom Nginx server image.
- Steps:
  - Write a Dockerfile to install Nginx and copy a custom index.html file into the container.
  - Build the Docker image using docker build -t my-nginx ..
  - Run the container using docker run -d -p 80:80 my-nginx.
  - Access http://localhost to see your custom webpage served by Nginx.

Persist Data with Docker Volumes:

- Task:
  - Use Docker volumes to persist data between container runs.Steps:Create a Docker volume using docker volume create mydata.
- Run a container and mount the volume to /data inside the container:
  - docker run -d -v mydata:/data --name my-container nginx.
- Create files inside the container's /data directory and stop the container.
- Start a new container with the same volume:
  - docker run -d -v mydata:/data --name my-container nginx.
- Verify that the files created earlier are still present.

Network Docker Containers:

- Task:
  - Create and use a custom Docker network to connect multiple containers.
- Steps:
  - Create a Docker network using
  - docker network create my-network.
- Run a container named container1 & container2 and connect it to my-network:
  - docker run -d --name container1 --network my-network nginx
  - docker run -d --name container2 --network my-network nginx

- Verify Connectivity:
- Access container2 from container1 using its container name as the hostname:
  - docker exec -it container1 bash
  - (this cmmand insert you in container 1)
  - Execute below commands to install ping in container1
    - apt update
    - apt install iputils-ping
  - ping container2 (type this command)
- This command verifies that container1 can reach container2 over the my-network.
- CleanUp:
  - docker container stop container1 container2
  - docker network rm my-network

Jenkins run as container

- Execute jenkins image and try to connect with jenkins dashboard
- docker run -d -p 8080:8080 --name jenkinsservice jenkins/jenkins
- docker exec -it jenkinsservice bash

```
C:\Users\NEW>docker exec -it jenkinsservice bash
jenkins@9b1dde2e81b1:/$ cat /var/jenkins_home/secrets/initialAdminPassword
012e899f03904961bef1c7bf530686ad
jenkins@9b1dde2e81b1:/$ exit
exit
```

Activity

- Check in browser localhost:8080
- Paste password and continue with all pluging installation