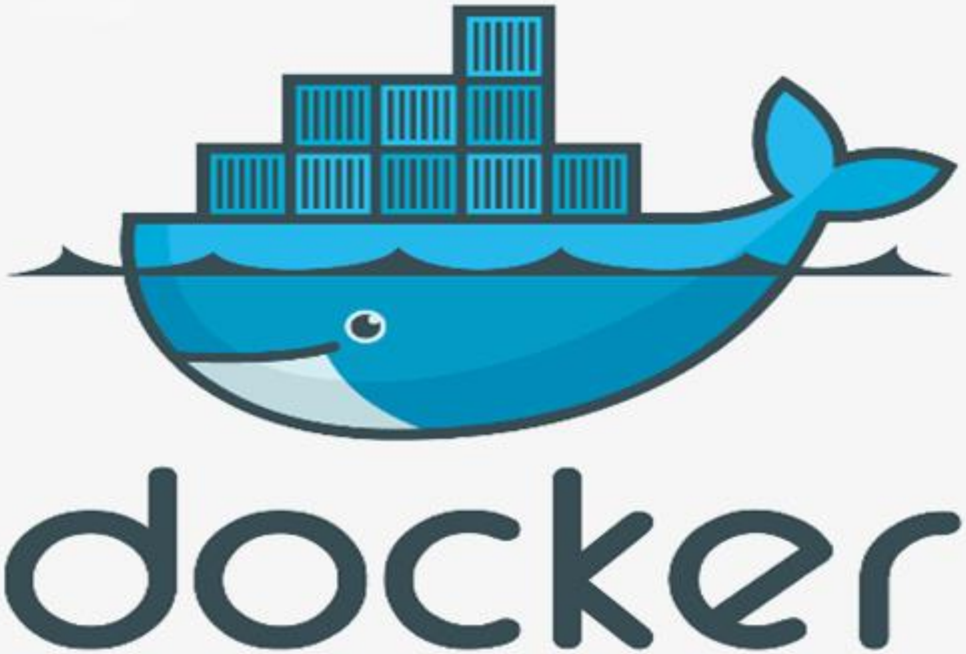
# DEVOPS – CI/CD DOCKER

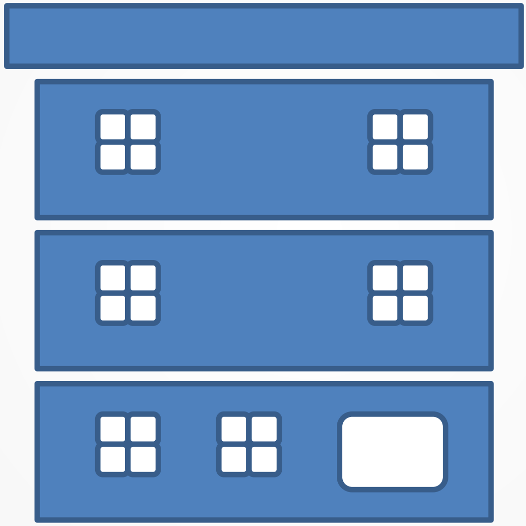
23-11-2020 to 27-11-2020

Venkatesh Reddy Madduri && ShankarPrasad


docker

# User Concerns

Hotel Server



User 1

App 1

User 2

App 2

Libraries

Kitchen

# User Concerns

It works fine on my machine

There is a Bug

Developer

Tester

phillipmartin.info

# User Concerns

What developers/tester care about :

- Portable runtime environment
- Missing dependencies, packages
- Run tests faster

What sys-admins care about:

- Cost & performance
- Efficient, consistent & repeatable
- Speed, reliability of CD & CI

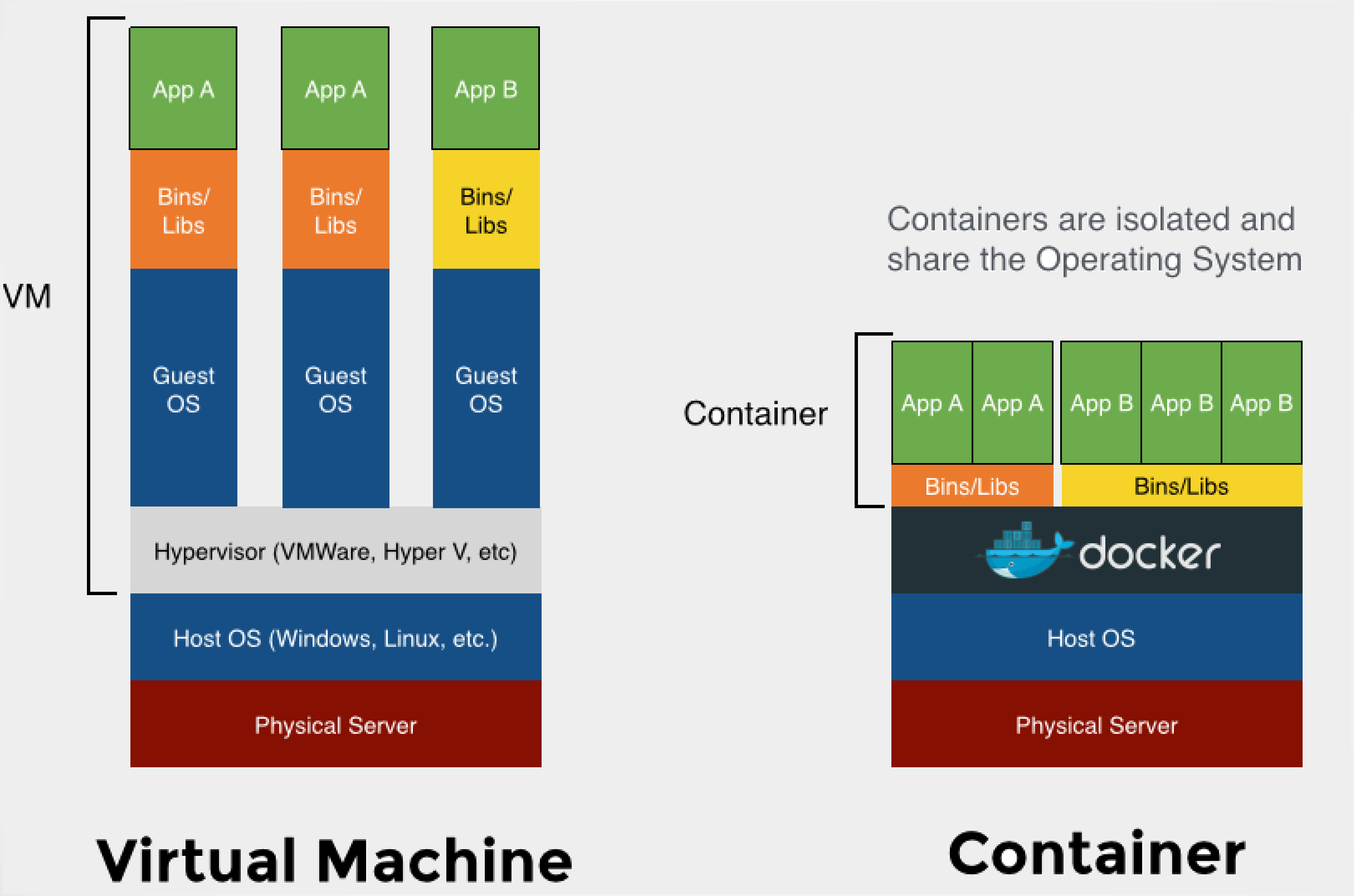
# What is docker?

- Docker is a platform for developing, shipping & running applications using an open-source container based technology
- OS level virtualization
- Run everywhere – physical or virtual or cloud
- Run anything – if it can run on host, it can run in the container

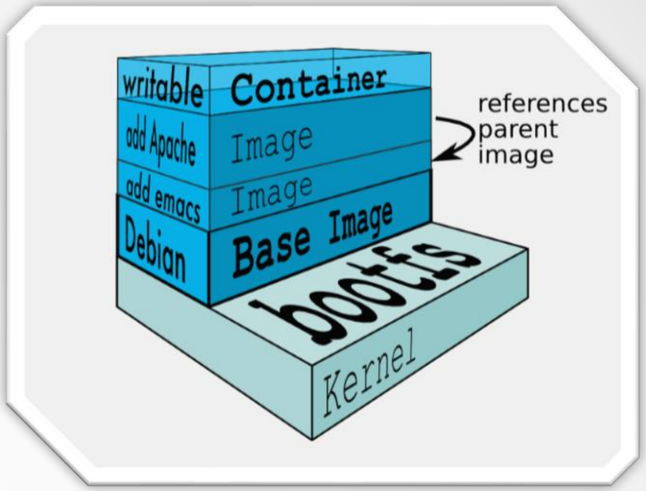
# Why Docker

- Scalable - lightweight
- Portable – Docker’d Apps can run anywhere
- Build any app in any language using any stack
- You don't have to pre-allocate any RAM
- Docker ensures your applications and resources are isolated and segregated
- Environment Standardization and Version Control

# Virtualization vs Containers



# Docker images



- Docker image is made up of file systems layered over each other
- A Docker base image is nothing but an OS user space minus the kernel
- Base is a boot filesystem, bootfs uses a *Union File System* & root filesystem stays in read-only mode
- UnionFS allows files and directories of separate file systems, to be transparently overlaid, forming a single coherent file system.
- Basically a tar file
- When a container is launched from an image, Docker mounts a read-write filesystem on top of any layers below

# Docker components

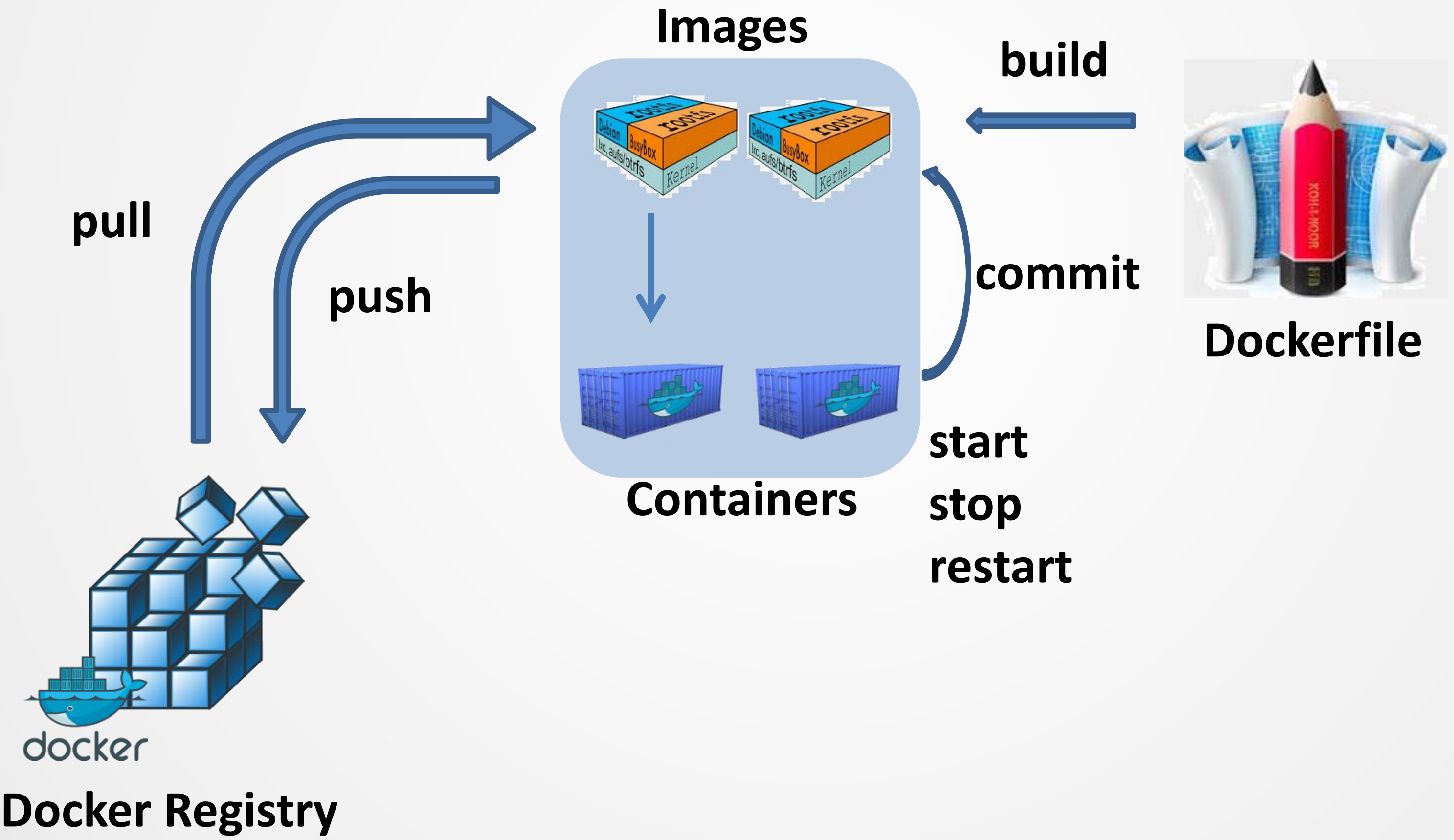
## Core components :

- **Docker Daemon**
  - Docker engine, runs on the host machine
- **Docker Client**
  - CLI used to interact with the daemon

## Workflow components :

- **Docker Image**
  - Templates which holds the environment & your applications
- **Docker Container**
  - Run-time instances created from images. Start, Stop, Run, Delete
- **Docker Registry**
  - Public & Private repositories used to store images
- **Dockerfile**
  - Automates image construction

# Docker system



Images

build

pull

push

commit

Dockerfile

Containers

start
stop
restart

Docker Registry

# Installing Docker CE (Community Edition)

## Windows/OSX

Boot2Docker is a tiny VM which ships with

- VirtualBox
- Docker client

http://boot2docker.io/

## CentOS

Set up the Docker CE repository

% **yum install -y yum-utils**

% **yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo**

Installing the Docker package

% **yum -y install docker-ce (** use **docker** as the package for **AWS AMI )**

Starting the Docker daemon

% **service start docker or systemctl start docker**

# First steps with Docker

- Ensuring Docker is ready

```
$ docker info
```

- Docker run command

```
$ docker run -itd  ubuntu //bin/bash
```

**-i** = keeps STDIN open from the container

**-t** = assings a pseudo tty to the container

```
% hostname
% cat /etc/hosts
% ps –aux
% apt-get install vim ( dpkg –l |grep vim)
% exit
```

- Container naming

```
$ docker run –name first_container  -it ubuntu //bin/bash
```

- Starting a stopped container

```
$ docker start first_container
```

- Attaching a container

```
$ docker attach first_container
```

- Creating daemonized containers

```
  $ docker run --name second_new -d ubuntu //bin/sh -c "while
true; do echo hello world; sleep 5; done“
```

- Fetch logs of a container

```
$ docker logs –f second_new
```

- Stopping a container

```
$ docker stop training_new
```

- List containers

```
$ docker ps
```

**-a** = lists all containers

**-q** = shows only container ID

- Deleting a container

```
$ docker rm training_new
```

- Inspecting the container's processes

```
$ docker top training_new
```

```
$ docker inspect <containerid>
$ docker inspect --format '{{ .NetworkSettings.IPAddress }}’ <containerid>
$ docker inspect --format='{{ .State.Running }}‘ <containerid>
```

- Run a command in a running container without attaching to it

```
$ docker exec <containerID>
 % docker exec training_new hostname
 % docker exec -d training_new ls
```

- Display a live stream of container(s) resource usage statistics

```
 $ docker stats [containerIDs]   apt-get update or apt-get install –y apache2
```

- Get real time events from the server

```
     $ docker events --filter
[container|event|image|label|type|volume|network|daemon
     % docker events --filter event=attach --filter event=die
     % docker events --since ‘1h’
```

- Exposing Our Container With Port Redirects

  - All ports are private by default

  - When you docker run -p <port> ..., that port becomes public

  - When you docker run -P ... (without port number), all ports declared with EXPOSE become public

```
$ docker run -p <hostport>:<containerport> nginx:latest
% docker run –d -p 80:80 nginx:latest
```

# Data Volumes :

A volume is a specially designated directory within one or more containers that bypasses the Union File System

- Volumes can be shared and reused between containers.
- A container doesn't have to be running to share its volumes.
- Changes to a volume are made directly.
- Changes to a volume will not be included when you update an image.
- Volumes persist until no containers use them.

- Volumes declared from cmd-line

```
$ docker run –it –v /usr/data ubuntu
```

- Sharing Volumes across containers [ Data containers ]

- This is done using the --volumes-from flag for docker run

```
$ docker run --privileged=true –it –volumes-from test1 ubuntu
$ docker run --privileged=true –it –volumes-from test1 –volumes-from test2 ubuntu
```

- Sharing a directory between the host and a container

```
$ docker run –it –name test1 -v /home/user/Docker:/data ubuntu
```

- Rename a container

```
$ docker rename <src> <dest>
```

- Show current available images

```
$ docker images
```

- Search for images

```
$ docker search ubuntu
```

- Download images

```
$ docker pull debian:jessie
```

# Building Images Interactively

- Create a new container and make some changes

```
$ docker run --name training_the_container -i -t ubuntu
    % apt-get install vim
    % exit
```

- Inspect the changes

```
$ docker diff training_the_container
```

- Commit & run your image

```
$ docker commit training_the_container
$ docker commit training_the_container myfirstImage
$ docker run -it <newImageId>
```

- Tagging images

```
$ docker tag <newImageId> myfirstImage
```

# Dockerfile instructions

- **RUN :** Run the command when the container is being built
- **CMD :**

Specifies the command to run when a container is launched, if values are specified during launch it will override the Dockerfile value

```
% CMD ["echo", "Hi"]
```

```
$ docker build -rm -t="training/dockerfiles" .
$ docker run -it training/dockerfiles
```

- **ENTRYPOINT :**

Same as RUN, arguments we specify on the docker run command line will be passed as arguments to the command specified in the ENTRYPOINT

```
% ENTRYPOINT ["echo", "Hi"]
```

- **WORKDIR :**

Provides a way to set the working directory for the container and the ENTRYPOINT and/or CMD to be executed when a container is launched from the image.

```
% WORKDIR /usr/bin
```

- You can override the working directory at runtime with the -w flag

```
$ docker run -it -w /var training/dockerfiles
```

# Building Docker Images

- Create a test dir & a Dockerfile

```
% mkdir Test; cd test && vim Dockerfile
```

```
FROM ubuntu
RUN apt-get -y install vim
```

- Build the image

```
$ docker build -t myFirstImage .
```

- Running the built image

```
$ docker run -it myFirstImage
```

- List all the layers composing an image

```
$ docker history myFirstImage
```

- **ENV :**

set environment variables during the image build process

```
% ENV ORACLE_HOME /var
```

- **USER :**

specifies a user that the image should be run as

```
% USER nobody
```

- You can override this at runtime by specifying the -u flag with

```
$ docker run -it -u nobody training/dockerfile
```

- **VOLUME (data volumes) :**

```
% VOLUME ["/data" ]
```

- **COPY :**

Adds files and directories from our build environment into our image

```
% COPY readme /data1
```

- this will add the readme file from the build dir to /data in the image

- **ADD:**

Similar to COPY, whereas it can extract archives

```
% ADD latest.tar.gz /var/www/wordpress
```

- **MAINTAINER :**

Tells you who wrote the Dockerfile

```
% MAINTAINER training
```

- **EXPOSE :**

Tells Docker what ports are to be published in this image

```
% EXPOSE 8080
```

# Sample Dockerfile to setup Apache2

```
FROM ubuntu:12.04
MAINTAINER training
RUN apt-get update && apt-get install -y apache2 && apt-get clean && rm -rf /var/lib/apt/lists/*
ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2
EXPOSE 80
CMD ["/usr/sbin/apache2", "-D", "FOREGROUND"]
```

```
$ docker build -t apacheimg -f ./Dockerfileapache .

$ docker run -d -p 80:80 -v /var/www:/var/www apacheimg
```

# Sample Dockerfile to setup Mongodb

```
############################################################
# Dockerfile to build MongoDB container images
# Based on Ubuntu
############################################################
# Set the base image to Ubuntu
FROM ubuntu

# File Author / Maintainer
MAINTAINER Example training

# Update the repository sources list
RUN apt-get update
################## BEGIN INSTALLATION ######################
# Install MongoDB Following the Instructions at MongoDB Docs
# Ref: http://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/
# Add the package verification key
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10

# Add MongoDB to the repository sources list
RUN echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen' | tee /etc/apt/sources.list.d/mongodb.list

# Update the repository sources list once more
RUN apt-get update

# Install MongoDB package (.deb)
RUN apt-get install -y mongodb-10gen

# Create the default data directory
RUN mkdir -p /data/db
##################### INSTALLATION END #####################
# Expose the default port
EXPOSE 27017

# Default port to execute the entrypoint (MongoDB)
CMD ["/usr/bin/mongod", "--config", "/etc/mongodb.conf"]
```

- Build the Image using the new Dockerfile for Mongodb

```
$ docker build  -t mongodbimg -f ./DockerfileMongo  .
$ docker run -d -p 28001:27017 mongodbimg
```

- Installing Jenkins with Docker

```
$ docker run -p 8080:8080 --name=jenkins-master -d --env JAVA_OPTS="-Xmx8192m"  jenkins
```

- Deleting images

```
$ docker rmi training/dockerfiles
$ docker rmi `docker images -a -q`
```

# Deploying a registry server

- Start your registry:

```
$ docker run –d –p 5000:5000 –restart=always –name registry registry:2
```

- Tag a image in the registry

```
$  docker tag ubuntu localhost:5000/Ubuntu
```

- Push image to registry

```
$ docker push localhost:5000/Ubuntu
```

- Pull image from registry

```
$ docker pull localhost:5000/Ubuntu
```

- Stop the registry

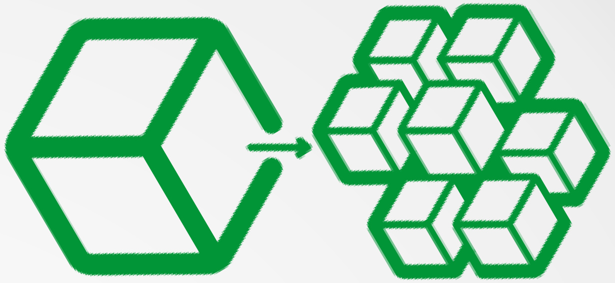
```
$ docker stop registry
$ docker rm –v registry
```

# Monolithic Applications

In a monolithic application, the core problem is this: scaling monolithic is difficult. The resultant application ends up having a very large code base and poses challenges in regard to maintainability, deployment, and modifications

- Monolithic Applications are **huge**, difficult to manage all the components like UI, database, message queue server, load balancers, web servers, storage
- **Frequent downtime** as even a single module failure brings the system down due to the cascading effect
- In order to do an **Technology adoption** or upgrade a technology stack, it would require the whole application to be upgraded, tested, and deployed
- Server costs go high as its **more expensive** to buy bigger capacity hardware
- Horizontal Scaling increases **operational costs**
- **High-risk in deployments** as deploying an entire solution or application in one go poses a high risk as all modules are going to be deployed even for a single change in one of the modules
- **Higher testing time** needed as to deploy the complete application, we will have to test the functionality of the entire application

# Microservice Architecture



- **Microservices** architecture is an approach to develop a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms
- Each component is continuously developed and separately maintained, and the application is then simply the sum of its constituent components

Benefits:

✓ **Developer independence**: Small teams work in parallel and can iterate faster than large team

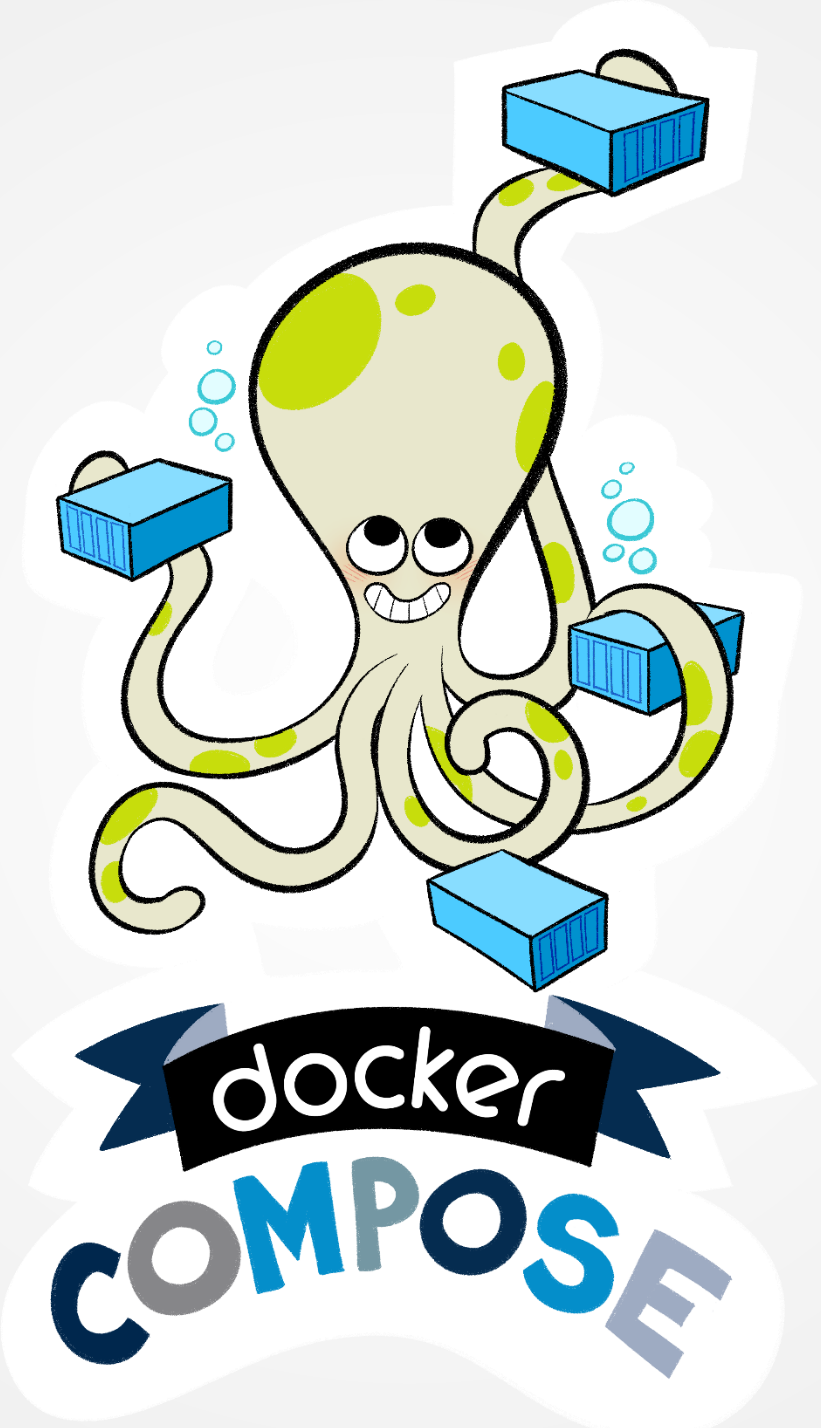
✓ **Isolation and resilience**: If a component dies, you spin up another while and the rest of the application continues to function

✓ **Scalability**: Smaller components take up fewer resources and can be scaled to meet increasing demand of that component only.

✓ **Lifecycle automation**: Individual components are easier to fit into continuous delivery pipelines and complex deployment scenarios not possible with monoliths

# Microservice solution using Docker

- Compose the application using Docker
- Break the application components into individual containers
- Split the data that’s shared between services into volumes
- Separate responsibilities so that each containers runs only one component/executable
- Store the changeable data (configurations, logs) as Volumes so that they are mounted on various containers


docker
COMPOSE

# What is Docker Compose ?

- Compose is a tool for defining and running multi-container Docker applications
- With Compose, you use a Compose file to configure your application's services. Then, using a single command, you create and start all the services from your configuration

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

Using Compose is basically a three-step process:

1. Define your app's environment with a Dockerfile so it can be reproduced anywhere.
2. Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.
3. Lastly, run docker-compose up and Compose will start and run your entire app

## Install Compose on Linux systems:

[https://docs.docker.com/compose/install/#install-compose](https://docs.docker.com/compose/install/#install-compose)

```
curl -L https://github.com/docker/compose/releases/download/1.18.0/docker-compose-`uname -s`-`uname -m` -o /usr/local/bin/docker-compose

sudo curl -L "https://github.com/docker/compose/releases/download/1.27.4/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

- Docker compose file is like an configuration file, where you define all the different stuffs we perform on command line into a file.
- The file is basically an Yaml file (.yml)
- Default file is docker-compose.yml

Examples:

https://github.com/scmlearningcentre/docker.git

```
$ docker-compose -f <composefile> <options>
$ docker-compose up -d <service>
$ docker-compose ps
$ docker-compose images
$ docker-compose logs –f <service>
$ docker-compose stop <service>
$ docker-compose rm <service>
$ docker-compose build <service>
$ docker-compose up --scale <service>=<Num>
```

Thank you