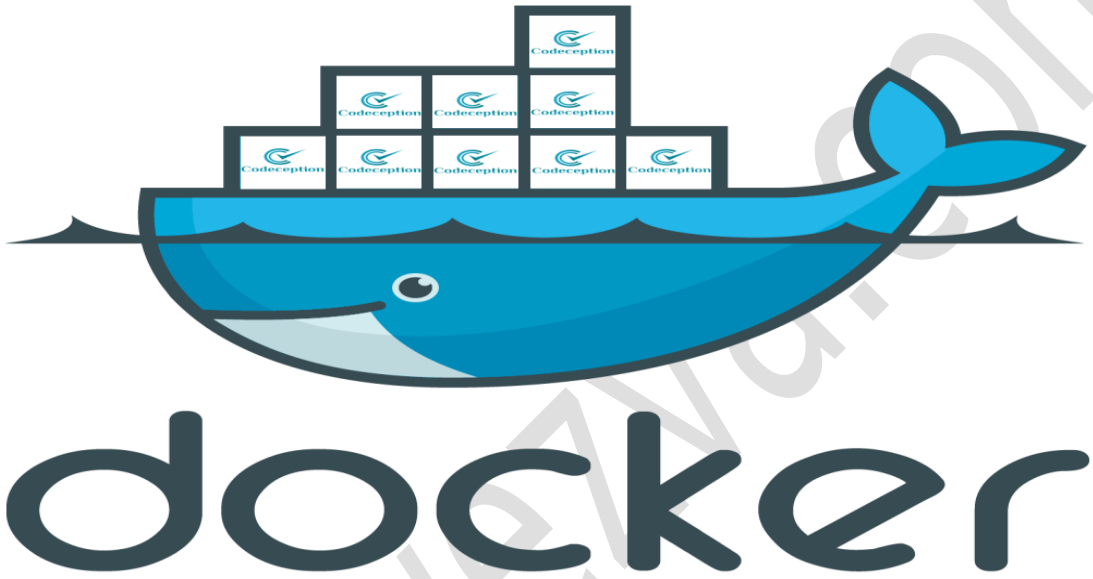
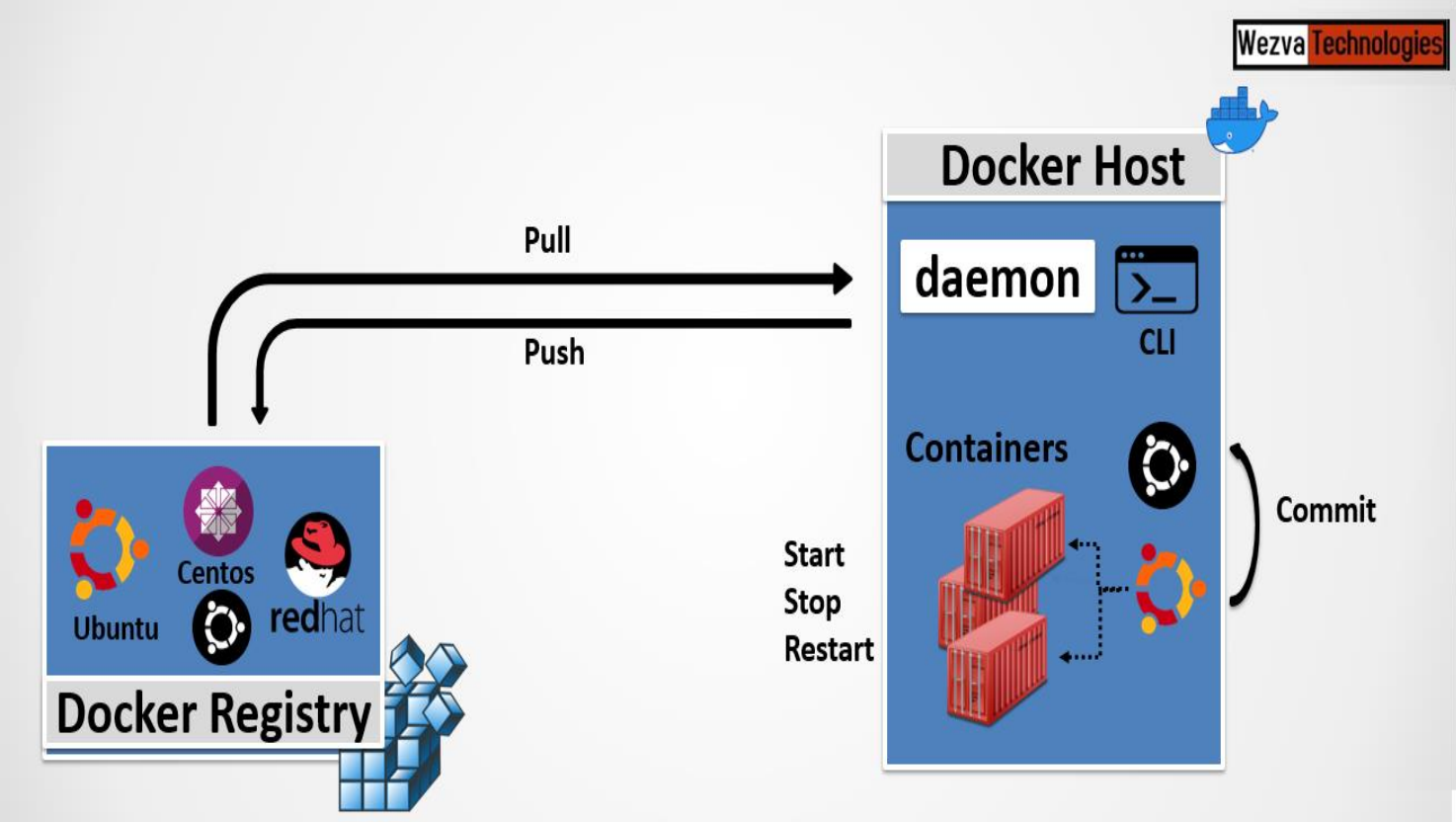
Docker Cheat Sheet



Introduction Containers allow the packaging of your application (and everything that you need to run it) in a "container image". Inside a container you can include a base operational system, libraries, files and folders, environment variables, volumes mount-points, and the application binaries.

A "container image" is a template for the execution of a container. It means that you can have multiple containers running from the same image, all sharing the same behavior, which promotes the scaling and distribution of the application. These images can be stored in a remote registry to ease the distribution.

Once a container is created, the execution is managed by the "Docker Engine" aka "Docker Daemon". You can interact with the the Docker Engine through the "docker" command. These three primary components of Docker (client, engine and registry) are diagrammed below:



Container related commands

Create & Run a Container in interactive mode

$ docker run --name <name> -it <Image> <FirstCMD>

* Download the Image
* create a new container (ID) using the Image
* start the container
* attach to the container

Start Container

$ docker start <ContainerID|Name>

Stop Container

$ docker stop <ContainerID|Name

Attach to a Running Container

$ docker attach <ContainerID|Name>

Delete Container

$ docker rm <ContainerID|Name>

Create & Run a Container in detached mode

$ docker run --name <name> -d <Image> <FirstCMD>

Print Logs of a Container

$ docker logs <ContainerID|Name>

$ docker logs -f <ContainerID|Name> # Live feed of the containers output

List Containers

$ docker ps #List of only Active containers

$ docker ps -a #List all containers

Execute a cmd/process inside a running Container without attaching to the terminal

$ docker exec <ContainerID|Name> <CMD>

$ docker exec -it <ContainerID|Name> /bin/bash #Executes and access bash

Inspecting the container

$ docker inspect <ContainerID|Name>

Inspecting the container's process

$ docker top <ContainerID|Name>

Copying file to a container from host

$ docker cp <hostfile> <ContainerID|Name>:<PathtoCopyInsideContainer>

$ docker cp <ContainerID|Name>:<PathtoCopyInsideContainer> <hostfile>

Pass Environment variable inside container

$ docker run -e VAR=VALUE -d <ImageName> <FirstCMD>

Change working dir inside container for first CMD

$ docker run -w <path> -d <ImageName> <FirstCMD>

Change default user inside container for running first CMD

$ docker run -u <user> -d <ImageName> <FirstCMD>

Binding Ports:

Docker containers can connect to the outside world without further configuration, but the outside world cannot connect to Docker containers by default.

- **In Docker, the containers themselves can have applications running on ports.**

- When you run a container, if you want to access the application in the container via a port number, you need to map the port number of the container to the port number of the Docker host.

Create a Container & map port to host machine

$ docker run --name <name> -d -p <HostPort>:<ContainerPort> <Image> <FirstCMD>

Create a Container & map all the ports to host machine

$ docker run --name <name> -d -P <Image> <FirstCMD>

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

Create a Container with Volume

$ docker run --name <name> -it -v <ContainerDir> <Image>

Create a Container with Volume & map to host machine

$ docker run --name <name> -it -v <HostDir>:<ContainerDir> <Image>

Create a Container and access volumes from another Container

$ docker run --name <name> -it --volumes-from <ContainerID> --privileged=true <Image>

Images related commands

Download Image

$ docker pull <Image>

Upload Image

$ docker push <Image>

List Images

$ docker images

Delete Image

$ docker rmi <ImageID|Name>

Building Images Interactively

Docker also gives you the capability to create your own Docker images.

Create Image Manually

$ docker commit <ContainerID|Name> <ImageName>

Dockerfile

A Docker File is a simple text file with list of instructions on how to build your images automatically.

Step 1 – Create a file called Docker File and edit it using vim. Please note that the name of the file has to be "Dockerfile" with "D" as capital.

Step 2 – Build your Docker File using the following instructions.

Instructions	Arguments
FROM	Sets the Base image for subsequent instructions

MAINTAINER	Sets the author field of the generated images
RUN	Executes commands inside the container
CMD	Default first command to execute
ENV	Sets an environment variable in the Image
ARG	Similar to ENV but sets the variable only on the temp container
ENTRYPOINT	Similar to CMD but doesn't overwrite use cmds, instead adds the user cmds as Arguments to the default cmd
COPY	Copies new files or directories into the filesystem of the container
ADD	Copies new files, directories or remote file URLs into the filesyste m of the container
USER	Sets the user name or UID to use when running an image
WORKDIR	Sets the working directory for any RUN, CMD, ENTRYPOINT, COPY, and ADD commands
EXPOSE	Informs Docker that the container listens on the specified network port at runtime.
VOLUME	Creates a mount point and marks it as holding externally mounted volumes from native host or other containers

Step 3 - Build Image using Dockerfile

```
$ docker build -t <ImageName>  .

# pick a different file other than default Dockerfile

$ docker build -t <ImageName> -f <Dockerfilepath> .
```

Example: Check class notes for more examples

```
MAINAINER mailme@wezva.com ADAM
FROM Ubuntu
USER root
RUN apt update
RUN apt install -y vim
```

```
WORKDIR /tmp
RUN touch Dummy
ENV MYNAME ADAM
ENV JAVA_HOME /opt/jdk1.8
COPY copyme /tmp/copyme
EXPOSE 80
CMD ["/bin/bash"]
```

Docker Tags or Image Tags

- An image name is made up of slash-separated name components, optionally prefixed by a registry hostname & a tag name.

  [Registry:Port]/[RepositoryName]/[ImageName]:[TagName]

- Docker tags convey useful information about a specific image version/variant
- Whenever an image is tagged without an explicit tag, it's given the "latest" tag by default

Docker Registry?

A Docker registry is a storage and distribution system for named Docker images. The same image might have multiple different versions, identified by their tags.

A Docker registry is organized into Docker repositories, where a repository holds all the versions of a specific image. The registry allows Docker users to pull images locally, as well as push new images to the registry (given adequate access permissions when applicable).

By default, the Docker engine interacts with DockerHub, Docker's public registry instance. However, it is possible to run on-premise the open-source Docker registry/distribution, as well as a commercially supported version called Docker Trusted Registry.

By default, Docker uses https to communicate with the Registry.

DockerHub

Docker Hub is the Docker's own registry offering which provides public & private repositories.

Other companies host paid online Docker registries for public use. Cloud providers like AWS and Google, who also offer container-hosting services, market the high availability of their registries.

▫ Amazon Elastic Container Registry (ECR) integrates with AWS Identity and Access Management (IAM) service for authentication. It supports only private repositories and does not provide automated image building.

▫ Google Container Registry (GCR) authentication is based on Google's Cloud Storage service permissions. It supports only private repositories and provides automated image builds via integration with Google Cloud Source Repositories, GitHub, and Bitbucket.

▫ Azure Container Registry (ACR) supports multi-region registries and authenticates with Active Directory. It supports only private repositories and does not provide automated image building.

▫ Private Docker Registry supports OAuth, LDAP and Active Directory authentication. It offers both private and public repositories, free up to 3 repositories (private or public)

Push Images to Public Repositories:

1. Login to Docker Registry

$ docker login [Registry server]

2. Tag your image

$ docker tag <YourImage>:[Tag] [Registry server]/[ImageName]:[Tag]

3. Push your image to Docker Hub

$ docker push [Registry server]/[ImageName]:[Tag]

Creating Private Docker Registry:

Download the Docker Private Registry Image & Start you registry

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

Tag your image

```
$ docker tag <YourImage>:[Tag] localhost:5000/[ImageName]:[Tag]
```

Push your image to Docker Registry

```
$ docker push localhost:5000/[ImageName]:[Tag]
```

Pull specific Image

```
$ docker pull localhost:5000/[ImageName]:[Tag]
```

Docker Network?

- Bridge: Default network created by docker. Every container gets an IP address & containers can access each other in the machine using this IP
- Host: This removes the network isolation between host & containers to use the host network directly. Used for standalone containers and not multiple web containers. This option is available only on Swarm services.
- Overlay: are best when you need containers running on different Docker hosts to communicate, or when multiple applications work together using swarm services.
- Macvlan networks are best when you are migrating from a VM setup or need your containers to look like physical hosts on your network, each with a unique MAC address.