Docker and Containerization

A Quick view on why and Docker

Why do we need Docker?

Relevance in IT Industry

- **Current IT Industry Problem :**
- Continual investment and substantial manual effort to move code (Deploy) between environments.
- A typical modern system may include Javascript frameworks, NoSQL databases, message queues, REST APIs, and backends all written in a variety of programming languages
- This stack has to run partly or completely on top of a variety of hardware—from the developer's laptop and the in-house testing cluster to the production cloud provider.

Relevance in IT Industry

Developers focus on

building the application

and shipping it through testing and production

without worrying about differences in environment and dependencies.

Operations focus on the

core issues of running containers,

allocating resources,

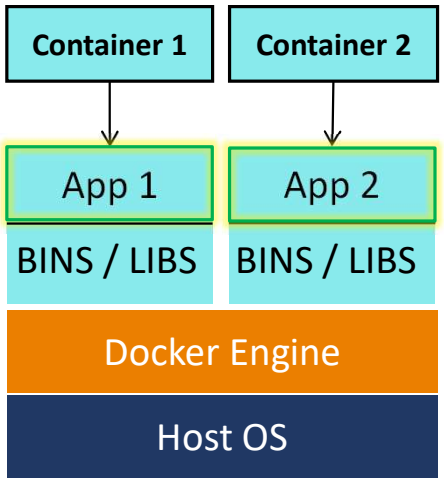
starting and stopping containers, and

migrating them between servers.

What is Docker ?

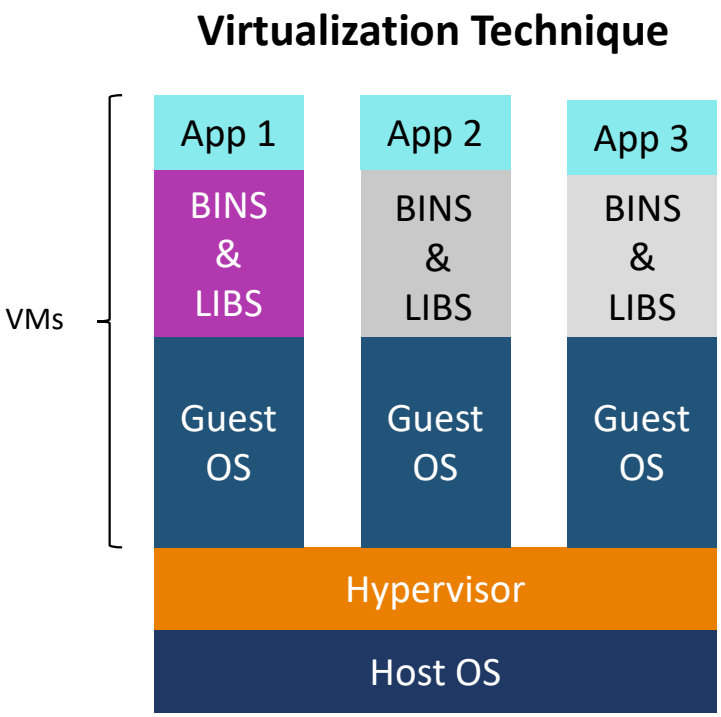
Docker and Containerization

Docker is a Containerization platform which packages your application and all its dependencies together in the form of Containers so as to ensure that your application works seamlessly in any environment be it Development or Test or Production.



Containers v/s Virtual Machines

Virtualization



Advantages

- Multiple OS In Same Machine
- Easy Maintenance & Recovery
- Lower Total Cost Of Ownership

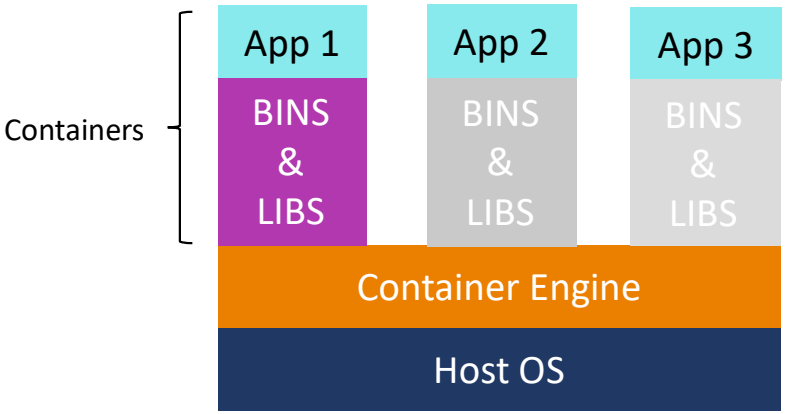
Disadvantages

- Multiple VMs Lead To Unstable Performance
- Hypervisors Are Not As Efficient As Host OS
- Long Boot-Up Process (Approx. 1 Minute)

Containerization

Note: Containerization Is Just Virtualization At The OS Level

Containerization Technique

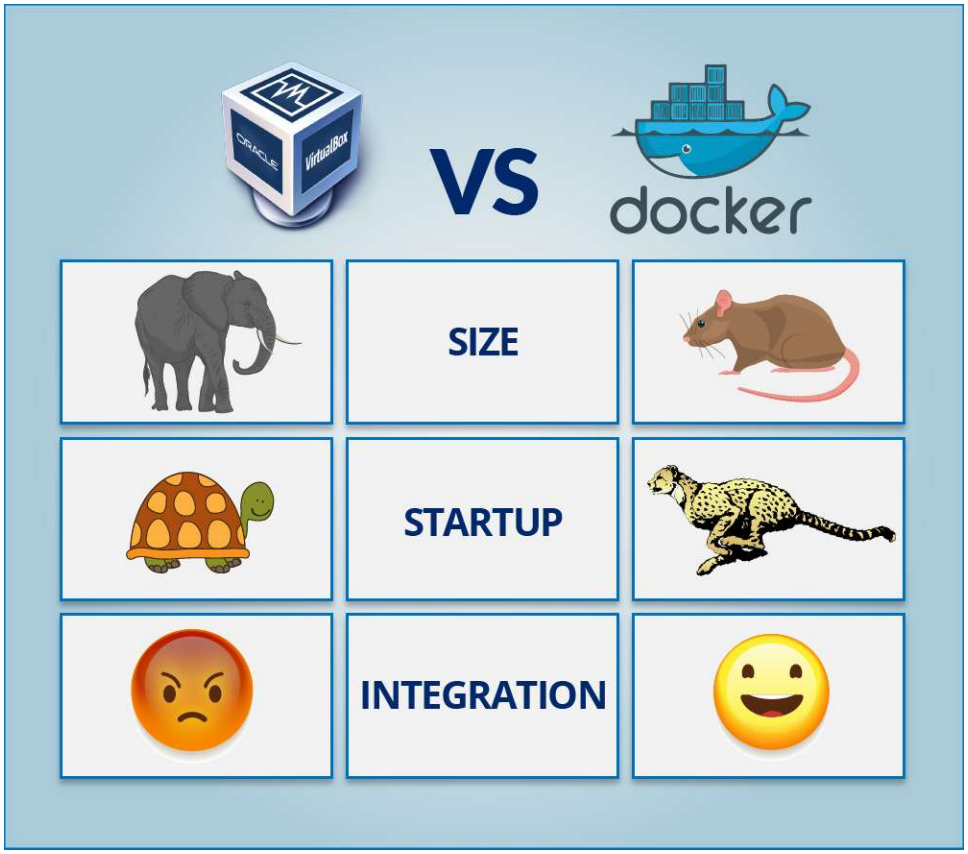


Advantages Over Virtualization

- Containers On Same OS Kernel Are Lighter & Smaller
- Better Resource Utilization Compared To VMs
- Short Boot-Up Process ($1/20^{th}$ of a second)

Benefits of Docker over VM’s

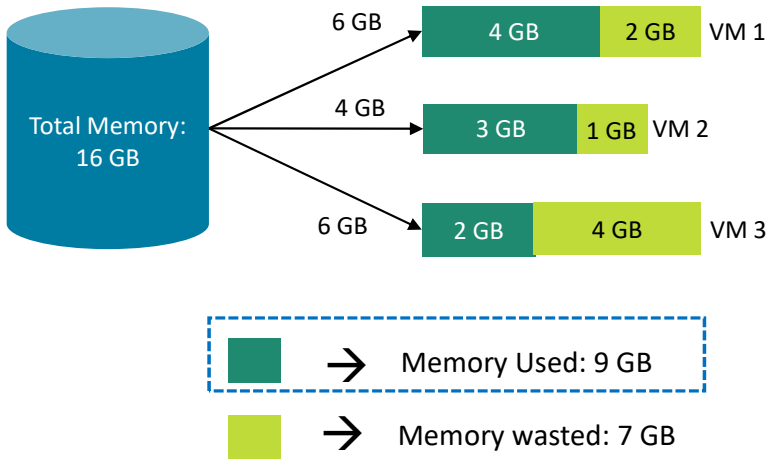
VMs or Containers?



Resource Management – Memory

- **Size**
- Start-up
- Integration

In case of Virtual Machines

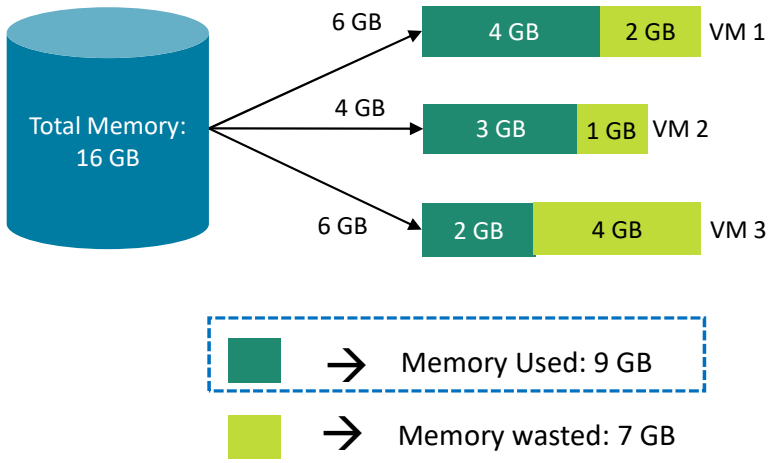


7 Gb of Memory is blocked and cannot be allotted to a new VM

Resource Management – Memory

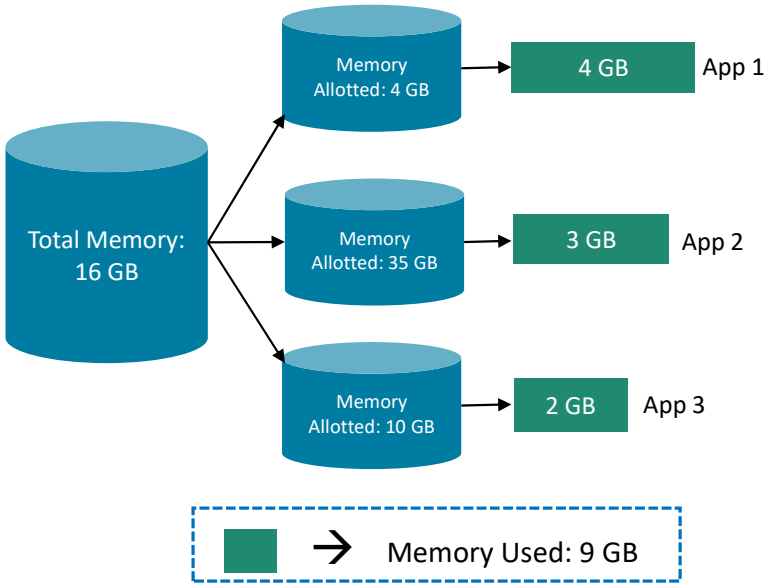
- **Size**
- Start-up
- Integration

In case of Virtual Machines



7 Gb of Memory is blocked and
cannot be allotted to a new VM

In case of Docker

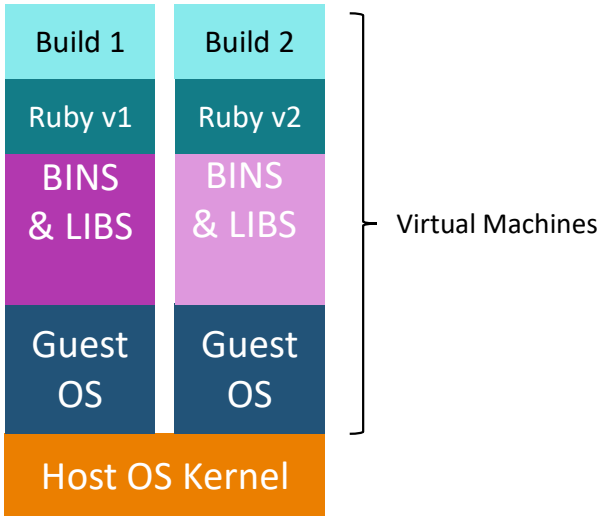


Only 9 GB memory utilized;
7 GB can be allotted to a new Container

Deployment

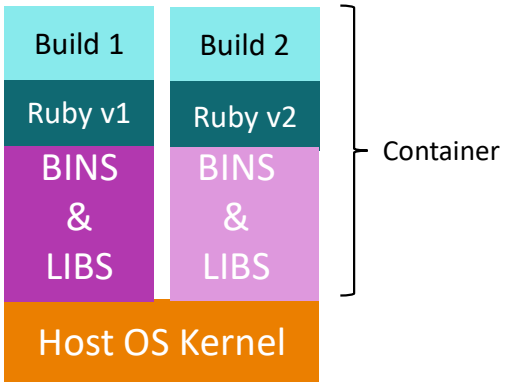
- Size
- **Start-up**
- Integration

In case of Virtual Machines



New Builds → Multiple OS → Separate Libraries → Heavy → **More Time**

In case of Docker



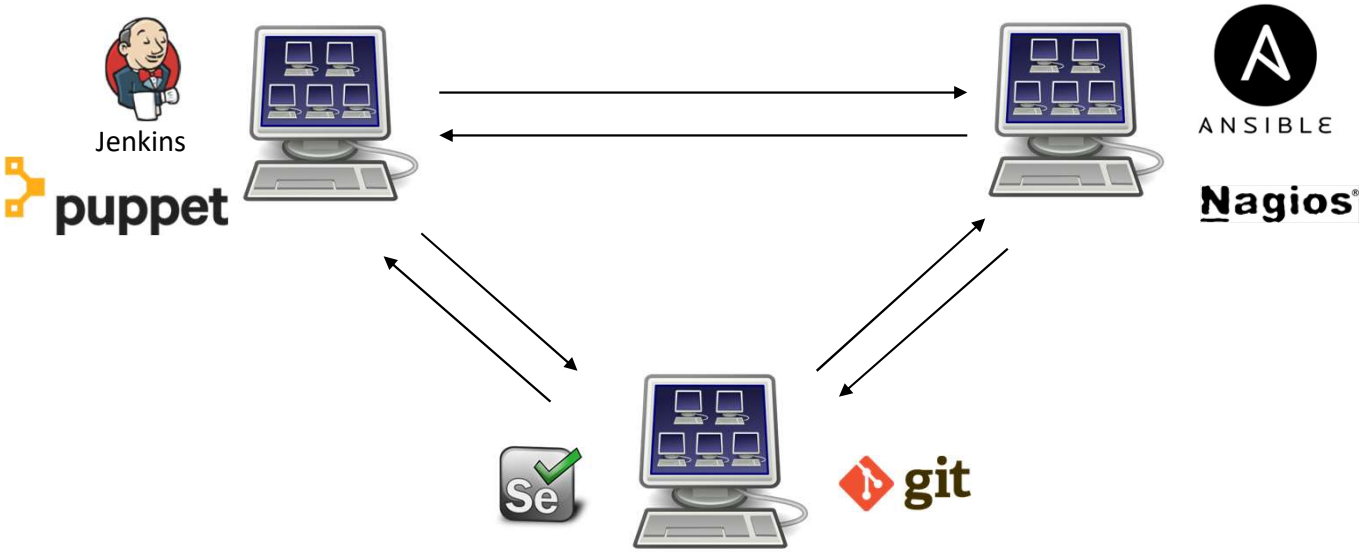
New Builds → Same OS → Separate Libraries → Lightweight → **Less Time**

IT Management – Integrations

- Size
- Start-up
- **Integration**

Integration In Virtual Machines Is Possible, But:

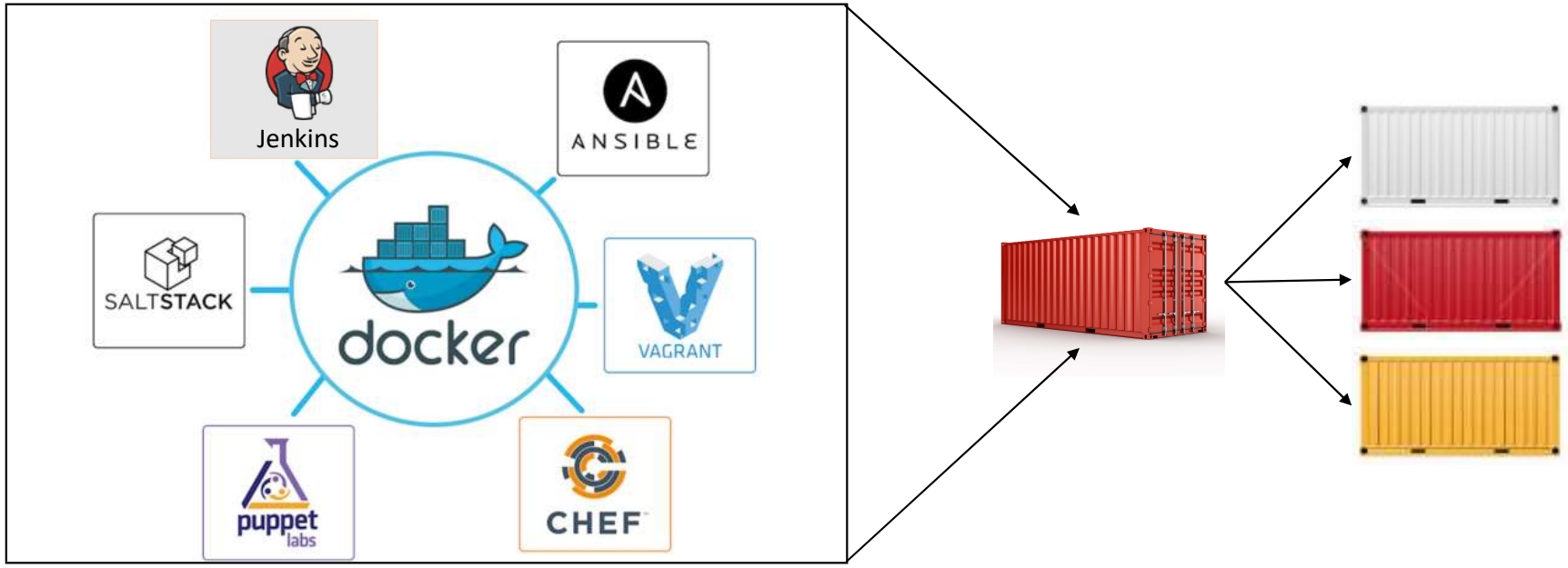
- Costly Due To Infrastructure Requirements
- Not Easily Scalable



Docker and Devops

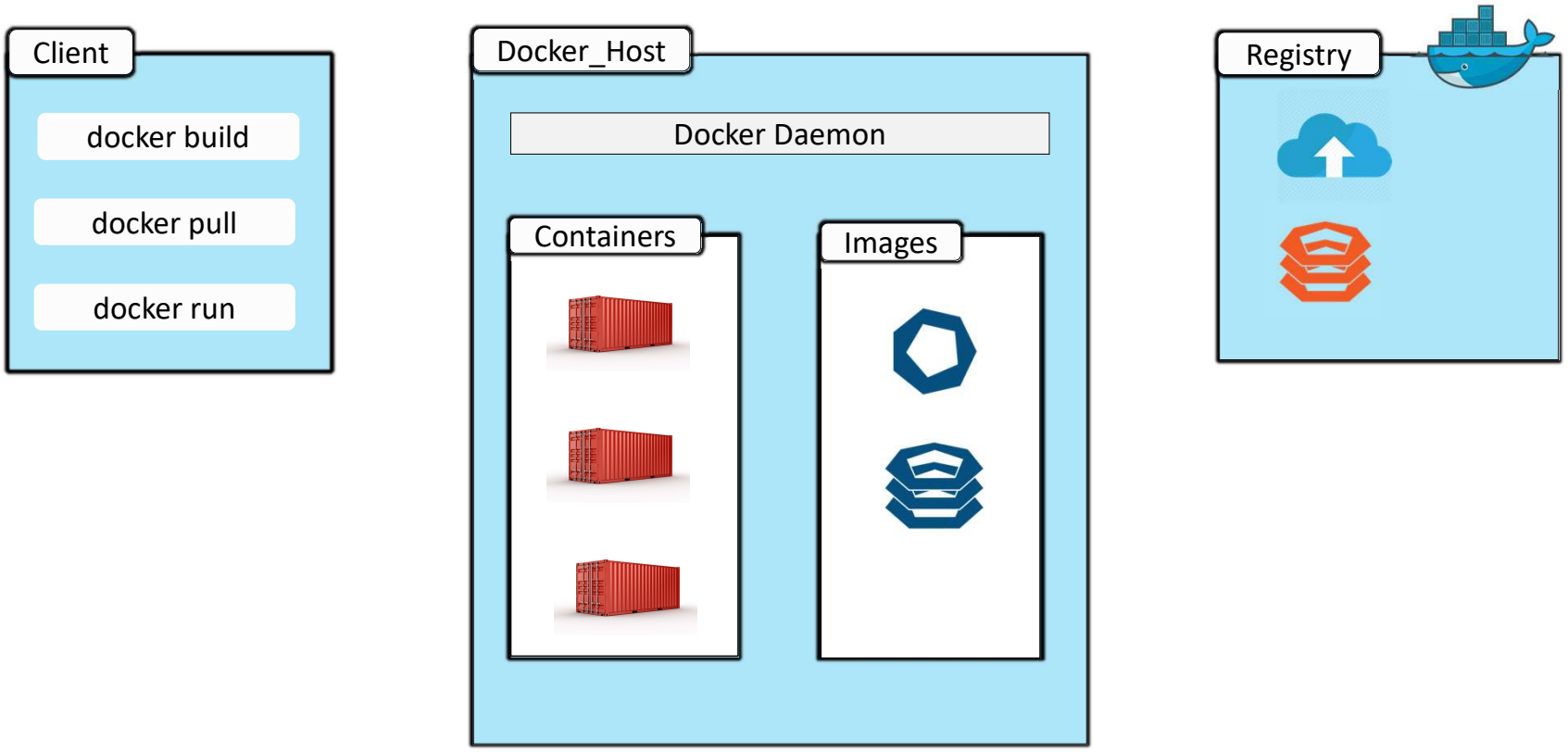
- Size
- Start-up
- **Integration**

Integration in Docker is Faster, Cheap & Easily Scalable



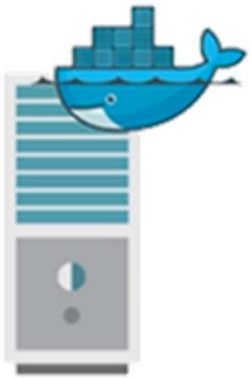
Architecture

Docker Architecture



Docker Architecture – Docker Daemon

- The Docker daemon runs on a host machine.
- The user uses the Docker client to interact with the daemon.

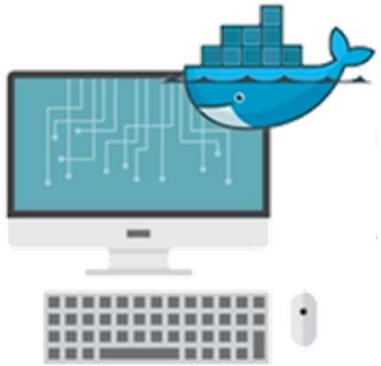


Why Use Docker Daemon?

- Responsible for creating, running, and monitoring containers
- Building and storing images

Docker Architecture – The Client

- The Docker client is the primary user interface to Docker.
- It accepts commands and configuration flags and communicates with a Docker daemon via HTTP.
- One client can even communicate with multiple unrelated daemons.



Why Use Docker Daemon?

- Since all communication has to be done over HTTP - it is easy connect to remote Docker
- The API used for communication with daemon allows developers to write programs that interface directly with the daemon, without using Docker

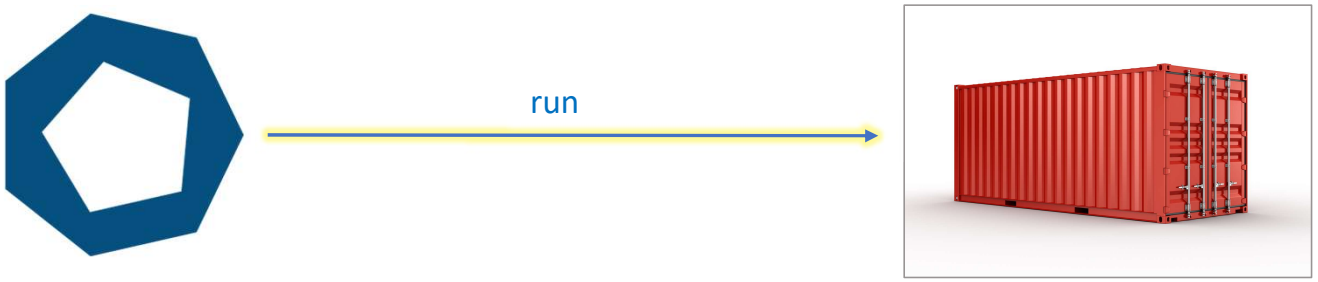
Docker Architecture – Docker Registry

- Docker Registry is a storage component for Docker Images
- We can store the Images in either Public / Private repositories
- Docker Hub is Docker's very own cloud repository

Why Use Docker Registries?

- Control where your images are being stored
- Integrate image storage with your in-house development workflow

Docker Architecture – Images/Containers



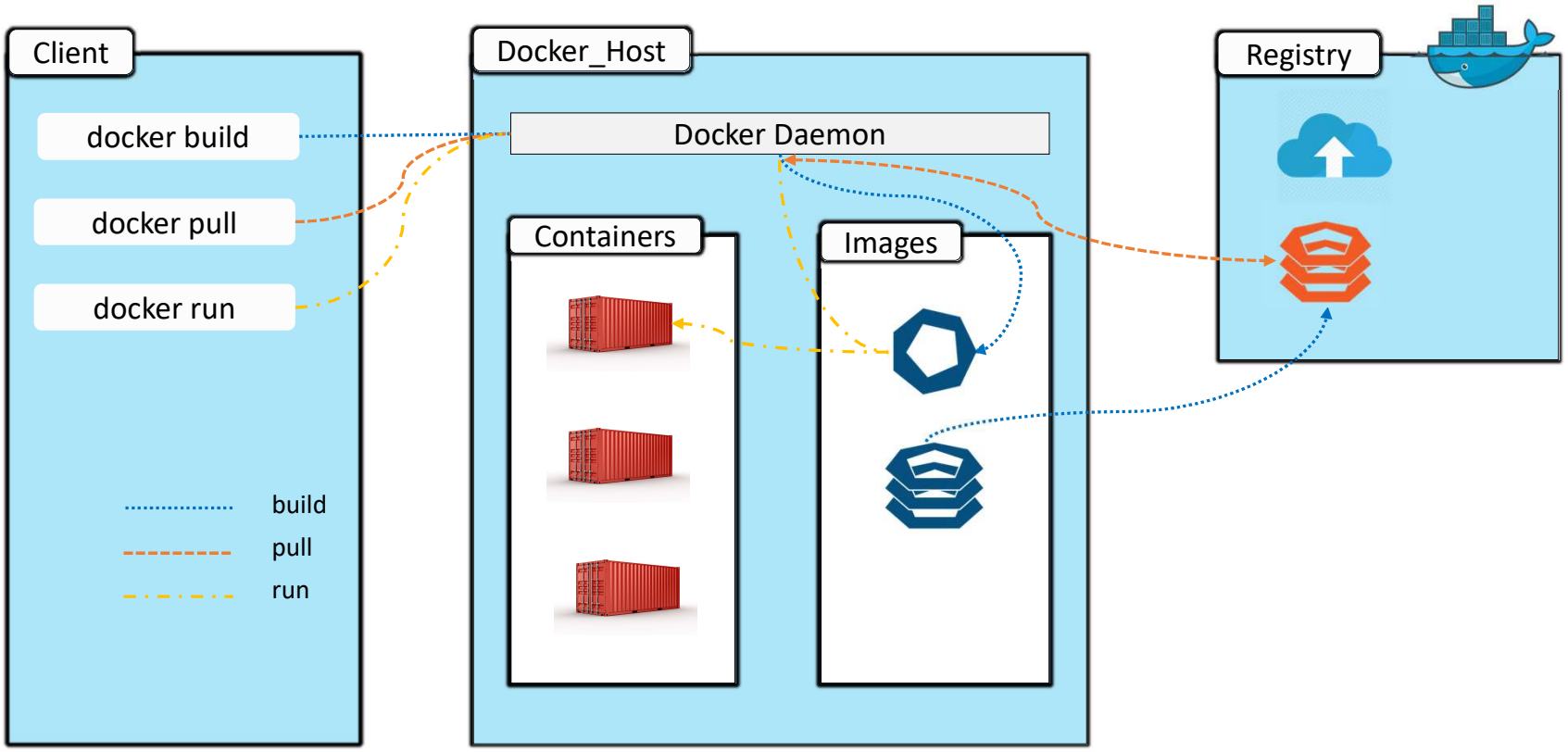
Docker Images

- Read Only Template Used To Create Containers
- Built By Docker Users
- Stored In Docker Hub Or Your Local Registry

Docker Containers

- Isolated Application Platform
- Contains Everything Needed To Run The Application
- Built From One Or More Images

Docker Architecture



Docker's Plugins And Plumbing

Docker Architecture – Plugins and Tools

The Docker engine and the Docker Hub do not in-and-of themselves constitute a complete solution for working with containers

Docker API

- ✓ API level allowing components to hook into the Docker Engine

Docker Compose

- ✓ Tool for building and running applications composed of multiple Docker containers
- ✓ Used in development and testing rather than production

Docker Machine

- ✓ Installs and configures Docker hosts on local or remote resources
- ✓ Machine also configures the Docker client, making it easy to swap between environments

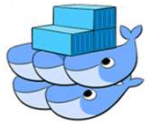
Docker Kitematic

- ✓ Kitematic is a Mac OS and Windows GUI for running and managing Docker containers

Docker Architecture – Plugins and Tools

Docker Trusted Registry

- ✓ Docker's on premise solution for storing and managing Docker images
- ✓ A local version of Docker Hub that can integrate with an existing security infrastructure and helps organizations comply with regulations regarding the storage and security of data
- ✓ Only non–open source product from Docker Inc.



Docker Swarm – Docker's Clustering Solution

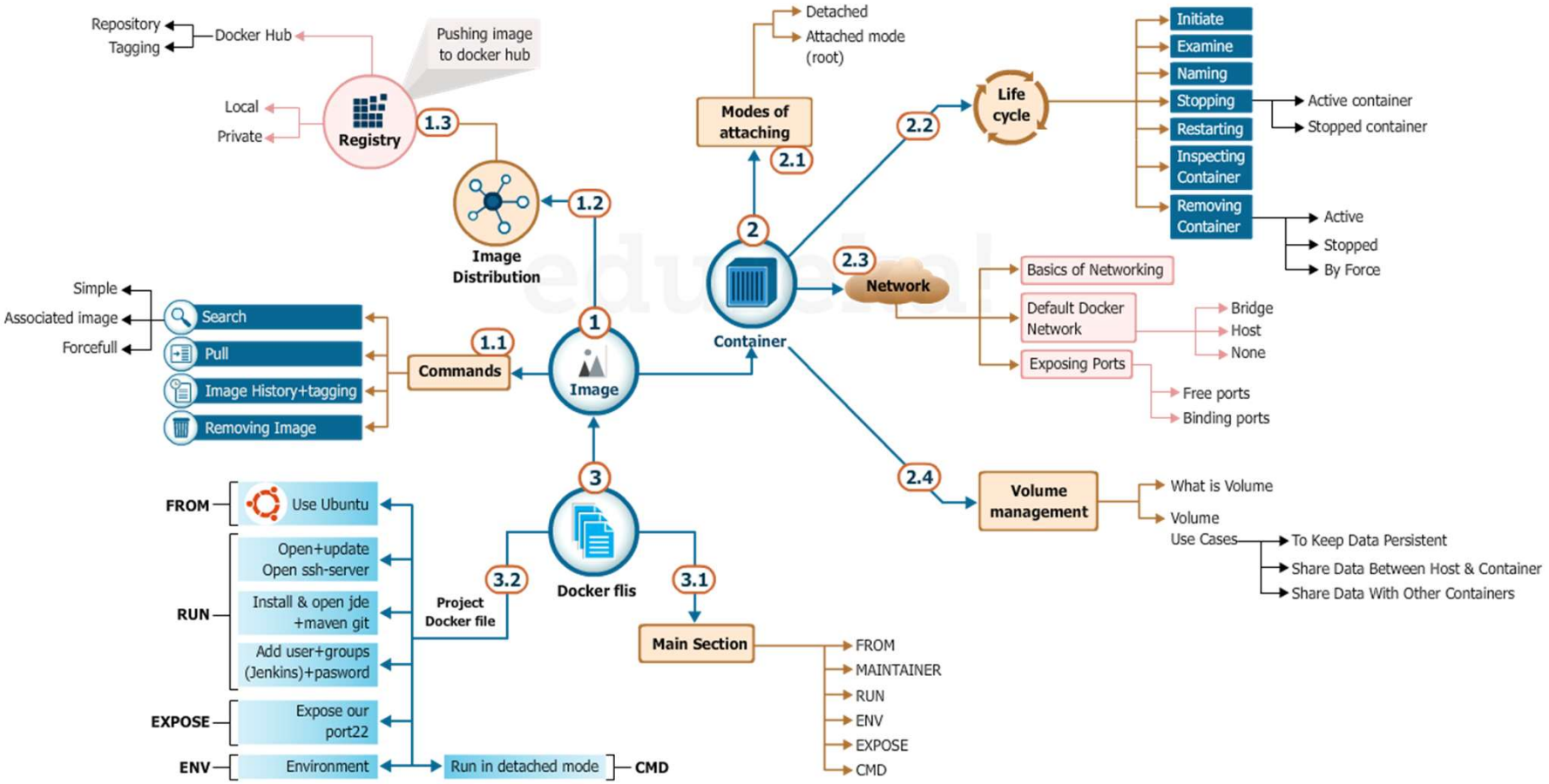
- ✓ Used to group several Docker hosts, allowing the user to treat them as a unified resource

Orchestration and cluster management

- ✓ In large container deployments, tooling is essential in order to monitor and manage the system
- ✓ Each new container needs to be placed on a host, monitored, and updated
- ✓ The system needs to respond to failures or changes in load by moving, starting, or stopping containers appropriately.
- ✓ Several competing solutions in the area, including Kubernetes from Google

Docker Topics

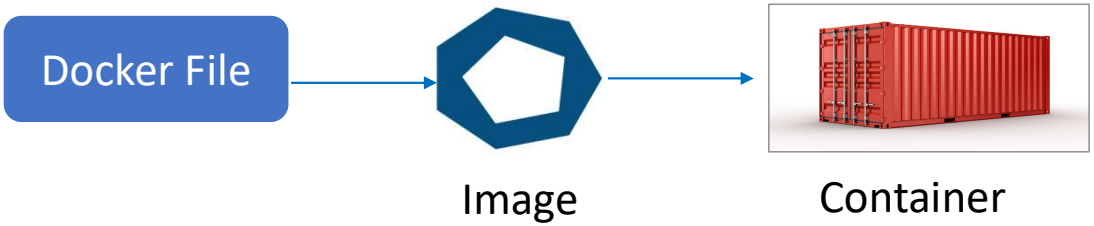
Docker Topics



Working with Docker Images

Docker Architecture - Images

- ✓ An image is basically a text file with a set of pre-written commands and saved as a file usually called as a Docker file
- ✓ Docker images are made up of multiple layers which are read-only filesystem
- ✓ A layer is created for each instruction in a Docker file and sits on top of the previous layers
- ✓ When an image is turned into a container the Docker engine takes the image and adds a read-write filesystem on top (as well as initializing various settings such as the IP address, name, ID, and resource limits)



Docker – A few Basic Commands

✓ Few Basic Commands:

- $ docker help :- Displays all the useful commands for Docker and other general help commands
- $ docker images :- Displays a list of existing images in Docker system. It also displays the following details:
  - **REPOSITORY:** Name of the repository
  - **TAG:** Every image has an attached tag.
  - **IMAGE ID:** Each image is assigned a unique ID
  - **CREATED:** The date when the image was created
  - **SIZE:** The size of the image

Docker – A few Basic Commands

✓ $ docker ps :- Displays the list of active containers. It also displays the following details:

- **CONTAINER ID:** Each container is assigned a unique ID
- **IMAGE:** Every image has an attached tag.
- **COMMAND:** Each image is assigned a unique ID
- **CREATED :** The date when the image was created
- **STATUS:** This shows whether the container is active or not
- **PORTS:** The number of exposed ports (needed for networking)
- **NAMES:** Name of container which is automatically assigned by Docker. It contains first name, "_" and last name

✓ $ docker ps – a :- Displays the list of all the container processes which are running or have run in the past

Docker – Hello World

✓ The following actions will be performed to pull the image:

- Search for images which start with “hello” word from Docker hub
- Select the image which we are going to pull
- Pull the selected image from Docker Hub
- Search for the copy of the image in our local repository
- Initiate a container based on “Hello-world” image
- Examine container Details

Docker – Hello World

- ✓ Command: $ docker search hello

```
vagrant@demo1:~$ docker search hello
NAME                                      DESCRIPTION                                     STARS     OFFICIAL
TED
hello-world                               Hello World! (an example of minimal Docker...   416       [OK]
kitematic/hello-world-nginx               A light-weight nginx container that demons...   88
tutum/hello-world                         Image to test docker deployments. Has Apac...   46
openshift/hello-openshift                 Simple Example for Running a Container on ...   22
google/nodejs-hello                                                                       16
dockercloud/hello-world                   Hello World!                                    13
karthequian/helloworld                    A simple helloworld nginx container to get...   6
hypriot/armhf-hello-world                 Hello World! (an example of minimal Docker...   5
marcells/aspnet-hello-world               ASP.NET vNext - Hello World                     4
nginxdemos/hello                          NGINX webserver that serves a simple page ...   3
crccheck/hello-world                      Hello World web server in under 2.5 MB          2
ppc64le/hello-world                       Hello World! (an example of minimal Docker...   1
yaros1av/hello-core                       Hello from ASP.NET Core!                        1
```

[OK] indicates that this is an official image from Docker

Stars provide information on most popular images

- ✓ Command: $ docker pull hello-world

```
vagrant@demo1:~$
vagrant@demo1:~$ docker pull hello-world
Using default tag: latest
latest: Pulling from library/hello-world
ca4f61b1923c: Pull complete
Digest: sha256:445b2fe9afea8b4aa0b2f27fe49dd6ad130dfe7a8fd0832be5de99625dad47cd
Status: Downloaded newer image for hello-world:latest
vagrant@demo1:~$
```

If no tag, pull latest version if not present locally.

Docker – Hello World

✓ Command: $ docker images

Result Details:

- **REPOSITORY:** This is our local repository
- **TAG:** Since we did not specify the Tag in our command, we got the latest version
- **IMAGE ID:** This is the ID of the image
- **CREATED:** This is the time when the image was created
- **SIZE:** This is the size of the image

Docker – Hello World

✓ Image History:

- Can get the history of the base image to understand how the base image was built
- Can see the different layers that were used during the build of image
- Gives ImageID, command that created particular layer and the size of the layer

✓ Command: $ docker history ImageName

```
vagrant@demo1:~$ docker history hello-world
IMAGE          CREATED        CREATED BY                                      SIZE      COMMENT
f2a91732366c   4 weeks ago    /bin/sh -c #(nop)  CMD ["/hello"]               0B
<missing>      4 weeks ago    /bin/sh -c #(nop) COPY file:f3dac9d5b1b030...   1.85kB
vagrant@demo1:~$
```

Latest version always shown first

Docker – Hello World

- Psychologically users tend to use the latest images. But sometimes we may need to use older images for following reasons.
  - Supporting clients who have not yet been migrated to newer technologies. Assume Client A is requiring Library Version 1.0 which was released around January 2013. An image built around this time is more likely to be supporting Library Version 1.0
  - History also allows to see if a particular images (esp. from a 3rd party) is even under active development/support.
    - As an example, if an image was last built in January 2012, it is unlikely we will consider this as a base image for our development.

Docker – Tagging

- ✓ Tagging:
  - Each image has a default tag associated with it
  - Default tag is set by the image maintainer
  - Command: $docker tag centos (Image ID or Image Name) Tag (Name of the desired tag)

```
vagrant@demo1:~$ docker tag hello-world sriram_hello-world
vagrant@demo1:~$ docker images
REPOSITORY                    TAG                 IMAGE ID            CREATED             SIZE
apache                        latest              eeae279ba281        2 days ago          257MB
sriram_abc                    latest              7be0f375fde8        2 days ago          221MB
sriram_image                  latest              7be0f375fde8        2 days ago          221MB
<none>                        <none>              49b8b5644fde        2 days ago          221MB
<none>                        <none>              019d32788f37        2 days ago          221MB
<none>                        <none>              00fd29ccc6f1        9 days ago          111MB
ubuntu                        14.04               67759a80360c        9 days ago          221MB
centos                        latest              3fa822599e10        3 weeks ago         204MB
hello-world                   latest              f2a91732366c        4 weeks ago         1.85kB
sriram_hello-world            latest              f2a91732366c        4 weeks ago         1.85kB
seshagirisriram/addressbook   15                  6444b79f2d4a        7 months ago        370MB
seshagirisriram/addressbook   latest              6444b79f2d4a        7 months ago        370MB
```

Docker – Tagging

- Docker uses the term tagging to refer to a label applied to an image (e.g. –t) or refer to string applied to end of image name (e.g. jenkins:latest). The latter is usually referred to as a version tag.
- For version tags, there are really no clear cut best practices
- Usually practices for versioning in GIT are applied as is to docker tagging also.
- Docker's automated builds lets a user link a "version tag" to either to a branch or a tag in the git history.
- A "branch" in this case can refer either to a different git branch **or** merely a different sub-directory.
- **Matching to a Git tag provides the most clear-cut use of the docker version-tag;** providing a relatively static version stable link. (Food for thought: Is this a good practice or not?)
- Using the version tag to indicate any other difference is a widespread practice but with no clear use case except for supporting multiple dockerfiles in same repository.

Docker – Repository

✓ What is a Repository?

- A collection of images.
- There can be three kinds of repositories
- Local: Can be saved on the system. All the images which are pulled from Public or Private repository gets saved on the local repository
- Private:
  - Can get one free from Docker
  - If you need other private repositories then you need to pay
  - Requires Username and Password
- Public (Docker Hub):
  - Need to sign up. We will be discussing this in detail

Docker – Repository

✓ There is a hierarchical system for storing images, where following terminology is used:

Image Storing:

Registry

A service responsible for hosting and distributing images. The default registry is the Docker Hub.

Repository

A collection of related images (usually providing different versions of the same application or service)

Tag

An alphanumeric identifier attached to images within a repository (e.g., 14.04 or stable)

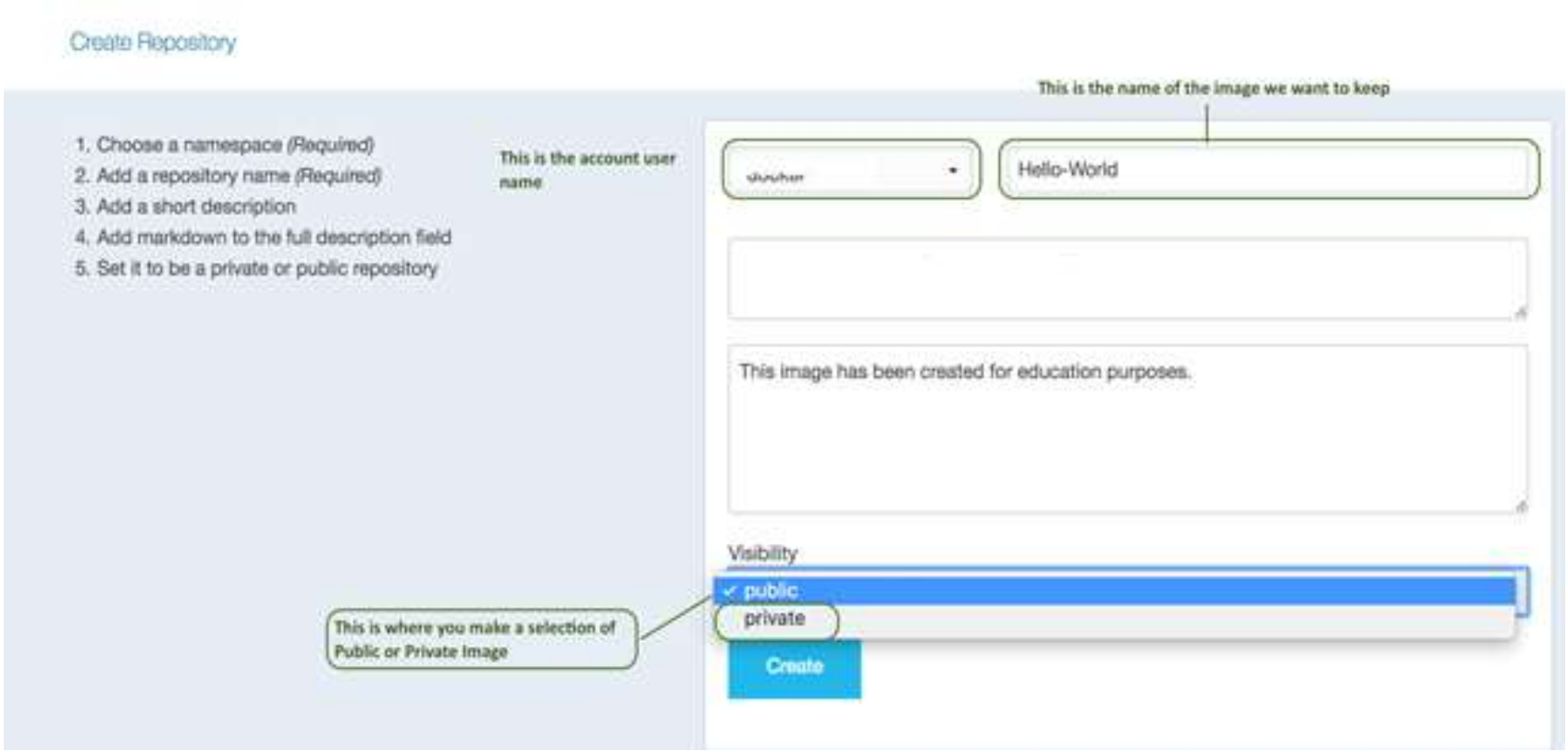
Docker – Repository

- ✓ As far as possible, use namespaces (We will be dealing with namespaces later on)
- ✓ Version tags as far as possible should be mapped to GIT branching tags and not used for indicating other differences.
- ✓ Clearly indicate for personal images, the use case for the image e.g. dev, qa, production.
- ✓ Assume you are the person responsible for pushing images to the repository. Automate the process of pushing images to the repository, which involves:
  - ✓ Push images
  - ✓ Remove images locally
  - ✓ Repull images (Sounds close enough to GIT Best Practices?)

Docker – Pushing Images

- ✓ Step 1: To push the images to Docker Hub, first login to Docker Hub
- ✓ Step 2: Create a new public repository with your name
- ✓ Step 3: Make sure your local image's name is the same as Docker repo's name
- ✓ Step 4: If it is not the same, tag your local image to give it the same repo name as the repository you created on Docker Hub using the command: $ docker tag <Local image name> <Docker Hub repo name>
- ✓ Step 4: Now, to push the Image to Docker Hub, use the command: $ docker push <Docker Hub repo name>

Docker – Pushing Images



Docker – Pushing Images

Docker – Pushing Images

- ✓ Image is pushed to Docker hub using command: $ docker push seshagirisriram(username)/sriram_hello_world(Name of image)

```
vagrant@demo1:~$ docker push seshagirisriram/sriram_hello_world
The push refers to a repository [docker.io/seshagirisriram/sriram_hello_world]
f999ae22f308: Pushed
latest: digest: sha256:48b69107095fc23bf6a4e61b0bd11f9843bf4f0e8bf4033fc3b6742b066b756e size: 524
vagrant@demo1:~$
```

If user name/password is required, you will be prompted for same.

Docker – Namespace

✓ Namespaces:

- Namespacing ensures users **cannot** be confused about where images have come from
- **Example:** If using the Centos image, it is the official image from Docker Hub and not some other registry's version of Centos image

✓ Following are three namespaces pushed Docker images, which can be identified from the image name:

- Names Prefixed With A String:
  - Names prefixed with a string and /, such as /nginx belong to the "user" namespace
  - These are images on Docker Hub that have been uploaded by a given user
  - **Example:** docker/nginx is the nginx image uploaded by the user docker

Docker – NameSpaces

- Simple Names:
  - Names such as Debian and Ubuntu, with no prefixes or /s, belong to “root” namespace
  - There are official images for most common software packages, which should be your first port of call when looking for an image to use
- Names Prefixed With Hostname or IP:
  - Names prefixed with a hostname or IP are images hosted on third-party registries (not the Docker Hub)
  - These include self - hosted registries for organizations, as well as competitors to the Hub, such as quay.io
  - Example: localhost:5000/wordpress refers to an WordPress image hosted on a local registry

Docker – Base Images

✓ Base Images

- **When creating your own images, you will need to decide which base image to start from**
- The best-case scenario is that you don't need to create an image at all you can just use an existing one and mount your configuration files and/or data into it
- This is likely to be the case for common application software, such as databases and web servers, where there are official images available
- In general, you are better off using an official image than rolling your own

Docker – Base Images

✓ Benefits of Using Base Images

- You get the benefit of other people's work and experience in figuring out how best to run the software inside a container

✓ If Base Image Is Not Available

- If there is a particular reason an official image doesn't work for you, consider opening an issue on the parent project, as it is likely others are facing similar problems or know of workarounds.

Docker – Base Images

✓ Benefits of Using Base Images

- You get the benefit of other people's work and experience in figuring out how best to run the software inside a container

✓ If Base Image Is Not Available

- If there is a particular reason an official image doesn't work for you, consider opening an issue on the parent project, as it is likely others are facing similar problems or know of workarounds.

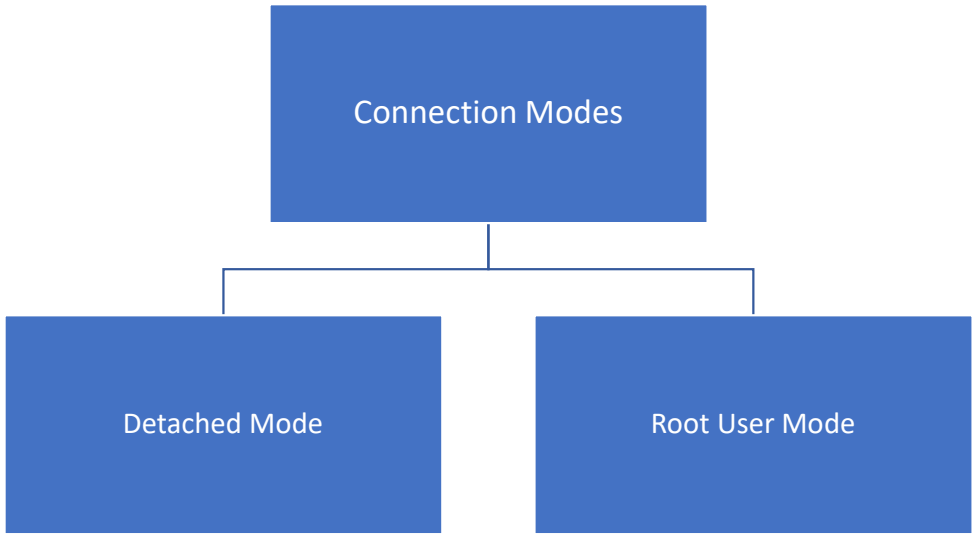
Containers

Docker – Containers

- ✓ Attaching to a container
- ✓ Container Life Cycle:
  - Initiating a container
  - Examining existing containers
  - Naming a container
  - Attaching to a container
  - Stopping a container
  - Restarting a container
  - Removing a container
- ✓ Inspect our existing containers and look for important information

Docker – Connection Modes

✓ Container can be connected in the following two modes:

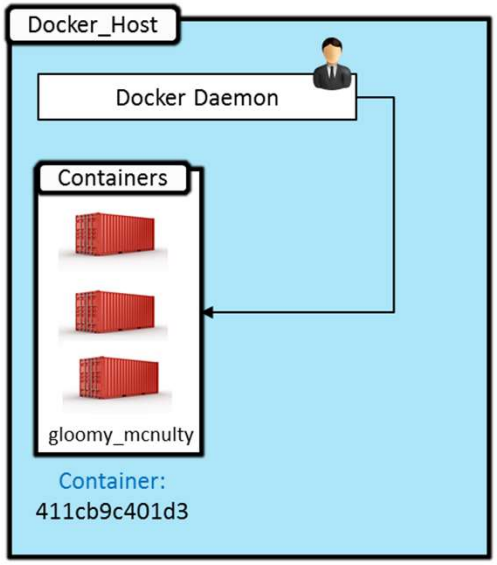


Docker – Detached vs User Mode

Detached Mode

✓Command: $ docker run –itd ubuntu:xenial

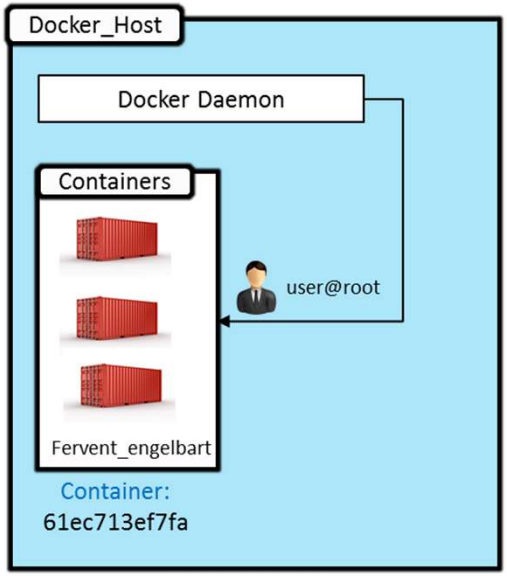
- I – Interactive
- T – Connected to terminal
- D – Detached mode



Root User Mode

✓Command: $ docker run –it ubuntu:xenial

- I – interactive
- T – connected to terminal



Docker – Detached vs User Mode

Detached Mode

- User manages from Daemon
- Container does not exit after the process within the container is over.
- Container could be stopped at a later stage though.
- Using the $docker attach command user can attach as a root user
- Allows control over other containers

Root User Mode

- User manages from the Root
- Container exits after the process within the container is over
- Container can be restarted at a later stage though
- Using the $docker exit command user can attach to the daemon
- Allows control over the container to which user is attached

Docker – Detached vs User Mode

- ✓ Containers started in detached mode exit when the root process used to run the container exits.
- ✓ A container in detached mode **cannot** be automatically removed when it stops.
- ✓ If this is not your use case and you do wish to automatically remove them, then you would not use the detached mode option.
- ✓ 2 examples that do not use detached mode is the use of Dockers plugins in Jenkins where we want to remove the containers after a Jenkins job is executed. This runs in a root user mode with attaching to the foreground and pretending to be a pseudo terminal and on completion close the container.
- ✓ Another example will be the starting of a service – where we need to start the service. However, using "service nginx start" with the –d option starts the nginx server but this cannot be used as is since container stops after the command executed.
- ✓ The detached mode is used when you want to run one off commands and (possibly) have the output of the commands send it outputs to some shared data volume for processing later.

Docker – Starting a container

- ✓ Command: docker run hello-world:latest

```
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker Hub account:
 https://hub.docker.com

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/
```

Container can be initiated using both the “Image name” as well as “Image ID” along with the required tag

Docker – Containers

- ✓ Command: docker ps

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES

- ✓ **Question:** What happened to the container we initiated from our image “hello-world?

Docker – Containers

✓ Command: docker ps

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES

Question: What happened to the container we initiated from our image “hello-world?

✓ Command: docker ps -a

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
7a45c308c3fa	hello-world	"/hello"	3 minutes ago	Exited (0) 3 minutes ago		pedantic_jepsen

Display inference:

This container with the name “pedantic_jepsen” , with CONTAINER ID (7a45c308c3fa) ,got created 3 minutes ago from the and exited 3 minutes ago as soon as its processes were executed.

Docker – Containers

- ✓ Docker assigns default names to the container. The usual format would be firstname_secondname.
- ✓ However Docker gives us the privilege to name our containers. This can be done with the use of the –name command

```
vagrant@demo1:~$ docker run  --name my_ubuntu_2 -itd ubuntu:14.04
c93ca54dc9079e5861ed7756a33a60044b366c2d1bc2934c6284d2d15824b971
vagrant@demo1:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
  NAMES
c93ca54dc907        ubuntu:14.04        "/bin/bash"         2 seconds ago       Up 1 second
  my_ubuntu_2
ab8b45a2f507        ubuntu:14.04        "/bin/bash"         20 seconds ago      Up 19 seconds
  my_ubuntu
vagrant@demo1:~$
```

Docker – Containers

- ✓ When we are running containers in Detached mode (Daemonised Mode) we can still attach to the container if required.
- ✓ Use the **docker attach** command to do so.

```
vagrant@demo1:~$ docker run  --name my_ubuntu_2 -itd ubuntu:14.04
c93ca54dc9079e5861ed7756a33a60044b366c2d1bc2934c6284d2d15824b971
vagrant@demo1:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
  NAMES
c93ca54dc907        ubuntu:14.04        "/bin/bash"         2 seconds ago       Up 1 second
  my_ubuntu_2
ab8b45a2f507        ubuntu:14.04        "/bin/bash"         20 seconds ago      Up 19 seconds
  my_ubuntu
vagrant@demo1:~$ docker attach ab8b
root@ab8b45a2f507:/#
root@ab8b45a2f507:/#
```

Docker – Containers

- When container is initiated in detached mode. It keeps running. It can be stopped in following two ways:
  - Get attached to the root and exit
  - Stop the container using the “stop” command
- Start a stopped container by using the “start” command: $ docker start

By default a container exists as soon as its processes are executed

Docker – Containers

- ✓ Inspect containers and to view important information regarding config, IP address etc. where both the running and stopped containers can be inspected by the command: $ docker inspect


```
[
    {
        "Id": "ab8b45a2f507125d8e8d7f3c61df082324ef3bfbafd4054331a0b75783b22072",
        "Created": "2017-12-24T19:12:30.588445493Z",
        "Path": "/bin/bash",
        "Args": [],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 2697,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2017-12-24T19:12:30.768009212Z",
            "FinishedAt": "0001-01-01T00:00:00Z"
        },
```

Docker – Containers

- ✓ The basic usage of inspecting containers is
  - ✓ to verify that everything is fine
  - ✓ debug issues (if any)
- ✓ The inspect command is used to inspect
  - ✓ Containers
  - ✓ Networks
  - ✓ Nodes
  - ✓ ……..
- ✓ The classical usage of these commands are in the Docker Pipeline plugin of Jenkins. This is used to work with Docker Swarm to inspect all nodes in a Swarm that can be used to run the jobs
- ✓ As a very advanced use case, the output of the docker inspect is used as an input to data visualizers to present your organization with real time data on containers, networks and nodes used in your docker infrastructure.

Docker – Removing Images

- With Associated Containers
- Without Associated Contain
- Removing by force

Removing Images: Use Cases

Use Case 1: Without Associated Containers

Use Case 2: With Associated Containers

Use Case 3: Removing by force

Docker – Removing Images

- **With Associated Container**
- Without Associated Contain
- Removing by force

✓ Removing Images with attached container:

- Docker will not allow you to remove an image if there are containers attached with the image
- The running containers will have to be removed first and then the image can be removed
- Using **–f** command, the images can be removed forcefully making the associated containers orphan which is not advisable though.

Docker – Removing Images

- With Associated Containers
- **Without Associated Contai**
- Removing by force

✓ Can use the command: $ docker rmi (Image Name/ID)

```
vagrant@demo1:~$ docker images
REPOSITORY                                TAG                 IMAGE ID
apache                                    latest              eeae279ba281
sriram_abc                                latest              7be0f375fde8
sriram_image                              latest              7be0f375fde8
<none>                                    <none>              49b8b5644fde
<none>                                    <none>              019d32788f37
<none>                                    <none>              00fd29ccc6f1
ubuntu                                    14.04               67759a80360c
centos                                    latest              3fa822599e10
hello-world                               latest              f2a91732366c
sriram_hello-world                        latest              f2a91732366c
seshagirisriram/sriram_hello_world        latest              f2a91732366c
seshagirisriram/addressbook               15                  6444b79f2d4a
seshagirisriram/addressbook               latest              6444b79f2d4a
vagrant@demo1:~$ docker rmi seshagirisriram/addressbook:15
Untagged: seshagirisriram/addressbook:15
```

Docker – Removing Images

- With Associated Containers
- Without Associated Contain
- Removing by force

✓ Docker allows to remove images with associated containers forcefully.

This is **not** recommended though.

✓ Can use the –f command to perform this action.

```
C:\Users\seshagiri sriram>docker rm jenkins-slave
Error response from daemon: cannot remove container "/jenkins-slave": container is running: stop the con
tainer before removing or force remove

C:\Users\seshagiri sriram>docker rmi jenkins-slave
Error response from daemon: conflict: unable to delete jenkins-slave:latest (must be forced) - container
 fe0f86ef10a8 is using its referenced image d1ceba282470
```

Docker – Removing Images

- **Use this very rarely and only for images that have containers.**
- As best practices, containers should be disposed off when they are done.
- If this is not possible, you will need to script to dispose of the containers before removing the image.
- **REITERATED WARNING** – use of the –f option is never a good option, unless you know that the image itself warranted removal because of
  - Poor performance
  - Security issues

Docker Networking

Docker – Networking

- ✓ When Docker is installed, it creates three networks **automatically**, which can be listed using the docker network Command : @docker ls

```
vagrant@demo1:~$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
fa9f01e86063        bridge              bridge              local
767b7ed0b7d1        host                host                local
3fa774fe21ea        none                null                local
vagrant@demo1:~$
```

Docker – Networking

✓ Bridge Network:

- The bridge network represents the docker0 network present in all Docker installations
- Docker daemon connects containers to this network by default.

✓ Host Network:

- The host network adds a container on the hosts network stack.
- You'll find the network configuration inside the container is identical to the host.

✓ None Network:

- The none network adds a container to a container-specific network stack. That container lacks a network interface.

Docker – Networking

- ✓ With the exception of the bridge network, you really don't need to interact with these default networks.
- ✓ While you can list and inspect them, you cannot remove them. They are required by your Docker installation.
- ✓ Command : $ docker network inspect bridge
- ✓ The Engine automatically creates a Subnet and Gateway to the network.
- ✓ Any new containers get added to this network

Docker – Networking

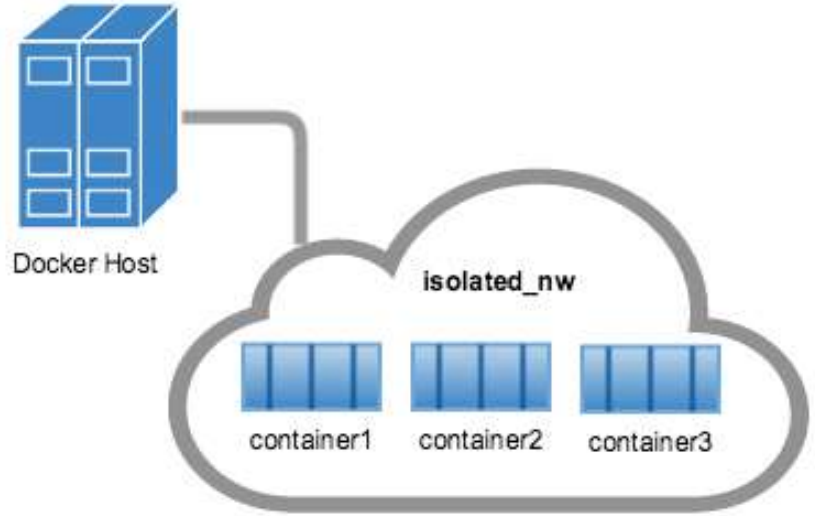

```
{
    "Name": "bridge",
    "Id": "fa9f01e86063713e8afa01e1407993f7811aa6d1f1e92ffa7446a2c85eeb9611",
    "Created": "2017-12-24T18:19:47.482394466Z",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
        "Driver": "default",
        "Options": null,
        "Config": [
            {
                "Subnet": "172.17.0.0/16",
                "Gateway": "172.17.0.1"
            }
        ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "Containers": {
        "ab8b45a2f507125d8e8d7f3c61df082324ef3bfbafd4054331a0b75783b22072": {
            "Name": "my_ubuntu",
            "EndpointID": "b0d5a30e121b92f9a417f3c7c214a615e40e038403fc459dfb68021461a4f888",
            "MacAddress": "02:42:ac:11:00:02",
            "IPv4Address": "172.17.0.2/16",
            "IPv6Address": ""
```


Automatic subnet and Gateways

IP auto assigned to container

Docker – Networking

- If you have a set of containers,
  - each providing micro-services to the other (and)
  - Should not be exposed to the external world
- The bridge network is the **best choice**.
- This is typically used with layered architectures.
- Be Aware that services in these containers are **not** exposed to outside networks.



Docker – Networking

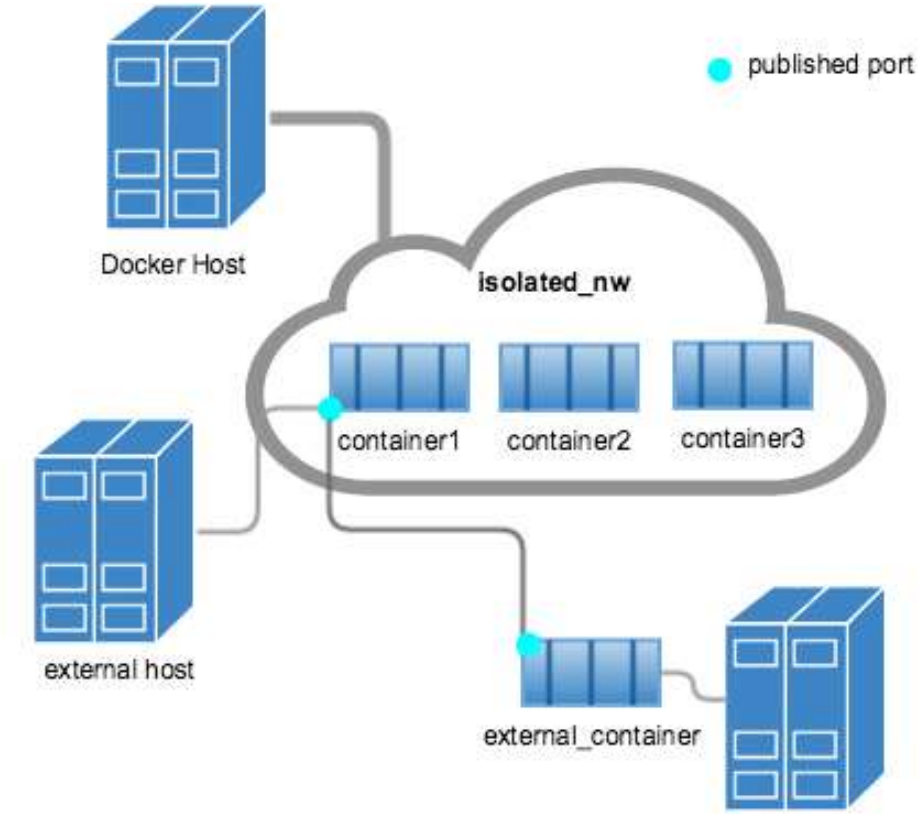
•Taking the same example from above, if you have an application that needs to expose part of the network and not all, then we use custom bridge networks that expose and publish custom ports.

•Again a tiered architecture, where you want to connect to a web application, which in turn connects to backend application or database services is an example of same – which is standard web application deployment practice

•A bridge network is useful in cases where you want to run a relatively small network on a single host.

•You can, however, create significantly larger networks by creating an overlay network.

Docker – Networking



Docker – Networking

- ✓ Ports are exposed in the container so that the container can be using the container IP. We can connect to the http address of that port.
- ✓ We are going to direct the port that is listening to http on a container to an underlying port on our host
- ✓ We can redirect the exposed ports to the host ports.
- ✓ There are two ways of exposing the ports – by commands: P or p
- ✓ P: Any ports that are exposed by the container, any random port between 32768 and 65000 will be available on the host machine. This is the ports available for Docker to pick randomly
- ✓ The available ports can be seen in different ways
- ✓ Command to connect to a container: $ docker run – d – name = nginx – P nginx:latest

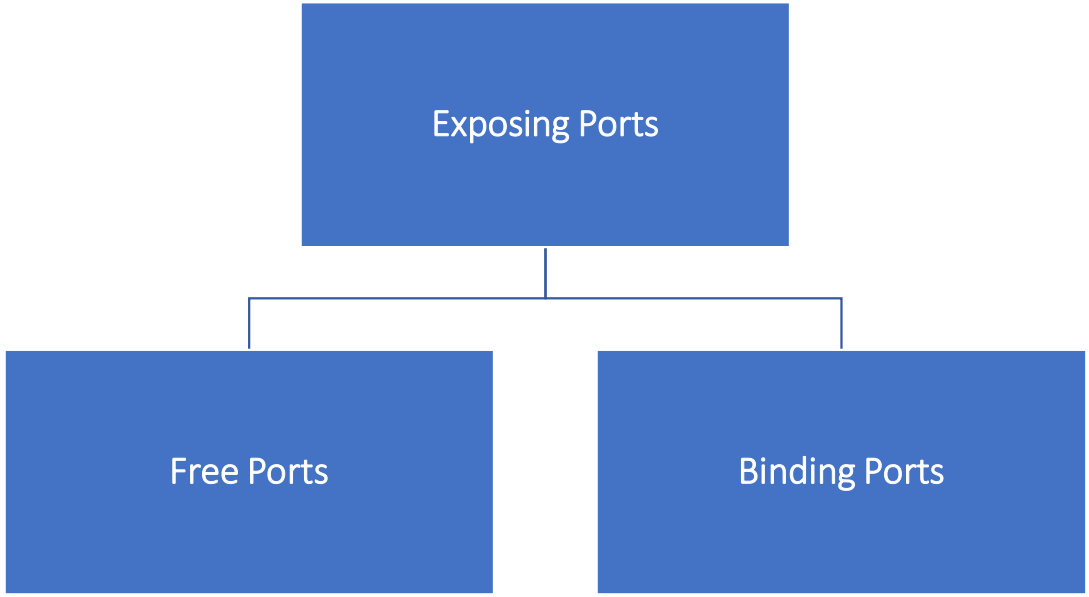
Docker – Networking

✓ Two ways to connect to a container:

- Connect if IP Address is known and I am on the host machine
- I am on the network of my host machine and the IP address and port of the container.

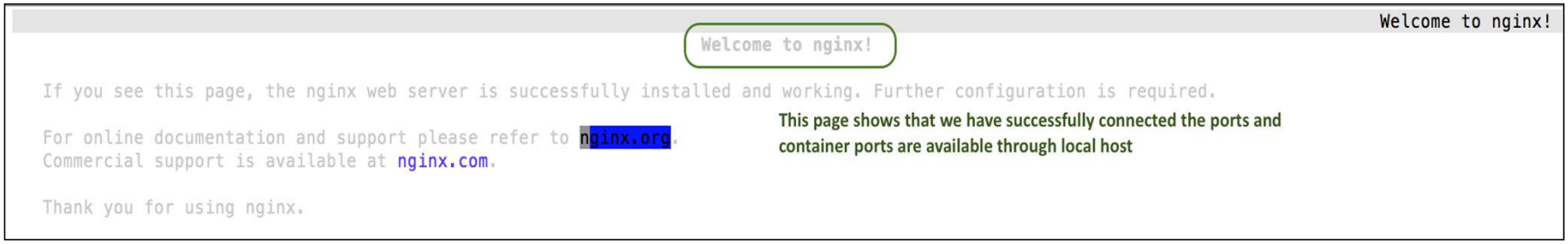
✓ It can also be obtainer through Docker port command : $ docker port nginx-demo $CONTAINERPORT

Docker – Networking



Docker – Networking

```
vagrant@demo1:~$ docker run -itd -P seshagirisriram/addressbook
c252e336be40685d198f06826af5d62a50bec0e0bb19952675972bae29b12603
vagrant@demo1:~$ docker ps
CONTAINER ID        IMAGE                          COMMAND             CREATED             STATUS              PORTS
 NAMES
c252e336be40        seshagirisriram/addressbook    "catalina.sh run"   8 seconds ago       Up 7 seconds        0.0.0.0:32768->8080/tcp
 hardcore_meitner
```



Docker – Networking

```
$ docker port nginx          $CONTAINERPORT      ← Variable to be passed
443/tcp -> 0.0.0.0:32770     ← Port 443 goes to all local addresses through port 32770
80/tcp -> 0.0.0.0:32771
```

✓ What if I want to bind it to a specific port on my server rather than any value which is picked in a range $ docker run –d –p 8080:80

```
/agrant@demo1:~$ docker run -itd -p 8080:8080 seshagirisriram/addressbook
4bf7c71a8fc4bd4a56d04ff1ca80d27b284df52533c1a36985446d206fdeb833
/agrant@demo1:~$ docker ps
CONTAINER ID        IMAGE                          COMMAND             CREATED             STATUS              PORTS
 NAMES
4bf7c71a8fc4        seshagirisriram/addressbook    "catalina.sh run"   3 seconds ago       Up 2 seconds        0.0.0.0:8080->8080/tcp
 competent_wiles
```

Docker – Networking

•It is generally preferred to use binding ports esp. for services that are external facing and do not need to change e.g. a web server needs to be exposed on port 80.

•It is also used (indirectly) as a proxy so that end users do no need to remember non-standard ports like 8081 (Technically firewall rules do not need to be tweaked much).

Docker Volumes

Docker – Volumes

- ✓ Volumes are files or directories that are directly mounted on the host and not part of the normal union file system
- ✓ Docker filesystems are temporary by default
- ✓ You can create, modify, and delete files as you wish
- ✓ If container is stopped and restarted, all the changes will be lost: any files you previously deleted will now be back, and any new files or edits you made won't be present
- ✓ Docker Data Volumes allow you to store data in separate place which can then be used as a normal folder - all changes are persistent
- ✓ Command: docker run –ti –v /hostLog:/log ubuntu
- ✓ Run second container, where volume can be shared: docker run –ti --volumes-from firstContainerName ubuntu

Docker – Volumes

- To Keep Data Persistent
- Share Data Between Host and Container
- Share Data With Other Containers

Docker Data Volumes: Use Case

Use Case 1: To keep data around, even through container restarts

Use Case 2: To share data between the host filesystem and the Docker container

Use Case 3: To share data with other Docker containers

- **To Keep Data Persistent**
- Share Data Between Host and Container
- Share Data With Other Containers

✓ There's no way to directly create a "data volume" in Docker - Instead create a data volume container with a volume attached to it. For any containers to connect to the data volume container, need to use the Docker's –volumes from option to grab the volume from this container and apply them to the current container

✓ Commands:

- # Create the data volume Container

  docker create -v /tmp --name datacontainer Ubuntu

- # The above creates a container called data container in the directory /tmp.
- # use it..

  docker run -t -i --volumes-from datacontainer ubuntu /bin/bash

- # from now on any data written to /tmp is persisted

Docker – Volumes

- To Keep Data Persistent
- **Share Data Between Host and Container**
- Share Data With Other Containers

✓ **Example:**

```
docker run -d -v ~/nginxlogs:/var/log/nginx -p 5000:80 -i nginx
```

✓ In above example, folder ~/nginxlogs on host machine is mapped to /var/log/nginx

✓ Writes to /var/log/nginx in container will be reflected to ~/nginxlogs on host

✓ Similarly writes to ~/nginxlogs on host will be reflected back to /var/log/nginx on container

Docker – Volumes

- To Keep Data Persistent
- Share Data Between Host and Container
- **Share Data With Other Containers**

- In addition to mounting a host directory in your container, some Docker volume plugins allow you to provision and mount shared storage, such as iSCSI, NFS, or FC.
- A benefit of using shared volumes is that they are host-independent.
- A volume can be made available on any host that a container is started on as long as it has access to the shared storage backend, and has the plugin installed.
- The –volumes-from option is used to share information between services.

Volumes – Use Cases

- To Keep Data Persistent
- Share Data Between Host and Container
- **Share Data With Other Containers**

✓ Other Commands:

```
# Example of using flocker
    docker run -d -P  --volume-driver=flocker  -v my-named-
  volume:/webapp --name web training/webapp python app.py
      # (or) Create a volume
      docker volume create -d flocker -o size=20GB my-named-volume
      # use it..
    docker run -d -P -v my-named-volume:/webapp  --name web
  training/webapp python app.py
```

Docker – Volumes

- Mounting host directories is usually used.
- Any write inconsistencies have to be managed by applications themselves – **not** DOCKER.
- For larger data volumes, docker has plugins to manage NFS or other shared storage volumes.
- The factors determining which ones to choose will depend on
  - Your volume requirements
  - Cost of solution

Docker Files

Docker – DockerFile

- ✓ Docker file is the basic building block of Docker containers
- ✓ Docker file is a file with a set of instructions written in it. It forms the basis for any image in Docker.
- ✓ Almost every time any base image is going to based upon another image. You are going to pick up a base image and build up on that image.

Docker – DockerFile

- ✓ Step 1: Create Directory
- ✓ Step 2: Create a named file subdirectory directory
- ✓ Step 3: Input the set of instructions
- ✓ Step 4: Save the file
- ✓ Step 5: Build the image

Docker File: Main Sections

✓ Following are the main sections of the Docker file:

- FROM
- How to RUN commands
- How to set up ENV (environments)
- Difference between CMD vs RUN
- How to EXPOSE ports

Docker File: FROM

FROM:

✓ Every Docker file starts with this command

✓ It shows where is the base image coming from

✓ Will pick up an image from Docker hub or some other repository and make some changes for ex: environmental changes, or expose your ports etc. and then save the file.

✓ Example: FROM debian:stable

Docker File: RUN

RUN:

✓ Set of actions you want to perform on the base image, where the modification of the base image starts.

✓ These actions have to be performed with root images

✓ Commands will be executed in the exact way you write it

Example:

```
FROM debian:stable
MAINTAINER  docker<sriram@gmail.com >
RUN apt –get update
RUN apt –get upgrade
```

Docker File: RUN

RUN:

✓ Following steps will happen when you save this file and build an image

- Step 1: It is going to pull the Debian base image
- Step 2: It will set up the MAINTAINER in the container
- Step 3: IT will update the packages
- Step 4: IT will upgrade the packages

Docker File: ENV

ENV:

✓ Can set up the environment variable

✓ By setting this up we can pass a variable that we need to pass inside the container that runs on base image

✓ Following format is required to set up this directive: ENV MYVALUE -test

✓ When you run the container this value will have to be passed using “echo $MYVALUE”

Example:

```
FROM debian:stable
MAINTAINER  docker<sriram@gmail.com >
RUN apt –get update
RUN apt –get upgrade
ENV MYVALUE -test
```

Docker File: EXPOSE

EXPOSE:

✓ Command to expose any ports to expose through your container to the underlying host operating system with mapping redirect. We can get to the containers through containers IP's.

✓ Ports are set up in the Docker file to be exposed.

✓ Following format is required to set up this directive: EXPOSE 80 (Port number which you want to expose)

✓ When you run docker ps for this container you will see the information for the ports which are exposed

Example:

```
FROM debian:stable
MAINTAINER  docker<sriram@gmail.com >
RUN apt –get update
RUN apt –get upgrade
ENV MYVALUE -test
EXPOSE 80
EXPOSE 24
```

Docker File: CMD

CMD:

✓ Command for starting up of a service of some kind

✓ Anything that is after a command is a list of things to run within any container that is initiated on a base image

✓ All the actions to run when the containers are initiated is described in this section

✓ Following format is required to set up this directive: CMD ["usr/sbin/apache2ctl"]

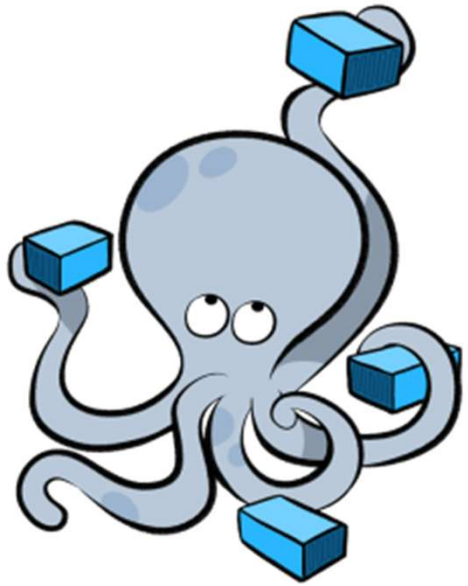
Example:

```
FROM debian:stable
MAINTAINER  docker<sriram@gmail.com >
RUN apt –get update && apt –get upgrade
ENV MYVALUE -test
EXPOSE 80
EXPOSE 24
CMD  ["usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

Questions and Queries

Should I include my code with COPY/ADD or a volume?

- You can add your code to the image using COPY or ADD directive in a Docker file.
- This is **useful** if you need to relocate your code along with the Docker image, for example when you' re sending code to another environment (production, CI, etc.).
- Prefer COPY over ADD as image size is reduced with COPY over ADD.
- You should use a volume if you want to make changes to your code and see them reflected immediately, for example when you' re developing code and your server supports hot code reloading or live-reload.
- There may be cases where you' ll want to use both. You can have the image include the code using a COPY, and use a volume in your Compose file to include the code from the host during development. The volume overrides the directory contents of the image.

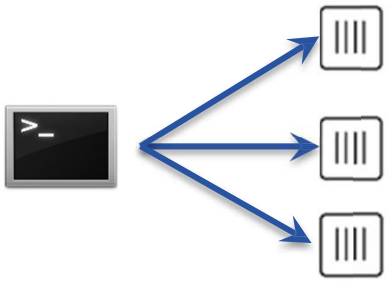


Docker Compose

Docker Compose: Multi Container Applications

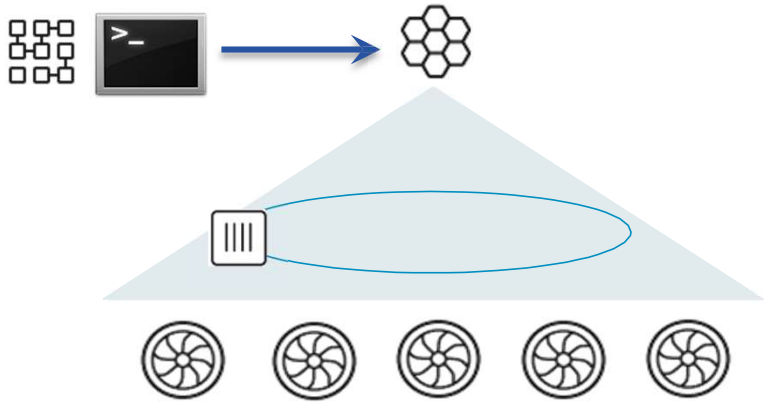
Without using Compose

- Build and run one container at a time
- Manually connect containers
- Manual Dependency Management



With Compose

- Define multi-container apps in a single file
- Single command to deploy entire apps
- Handles Dependencies
- Works with Networking, Volumes, Swarm



Docker Compose: Multi Container Applications

Without using Compose

```
docker run -e MYSQL_ROOT_PASSWORD=<pass> -e MYSQL_DATABASE=wordpress --name wordpressdb -v "$PWD/database":/var/lib/mysql -d mysql:latest

docker run -e WORDPRESS_DB_PASSWORD=<pass> --name wordpress --link wordpressdb:mysql -p 80:80 -v "$PWD/html":/var/www/html -d wordpress
```

With Compose

```
services:
  db:
   image: mysql:5.7 volumes:
    - db_data:/var/lib/mysql environment:
   MYSQL_ROOT_PASSWORD:
    MYSQL_DATABASE: wordpress
  wordpress:
     depends_on:
        - db
     image: wordpress:latest ports:
    - "80:80"
   environment:
    WORDPRESS_DB_PASSWORD=<pass>
```

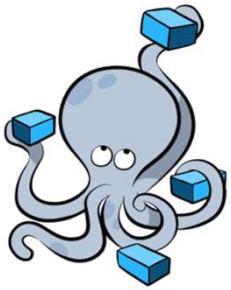
Docker Compose

1

A tool for defining and running multi-container Docker applications

2

With Compose, you use a YAML file to configure your application's services.



3

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

4

With a single command, you create and start all the services from your configuration

Docker Compose

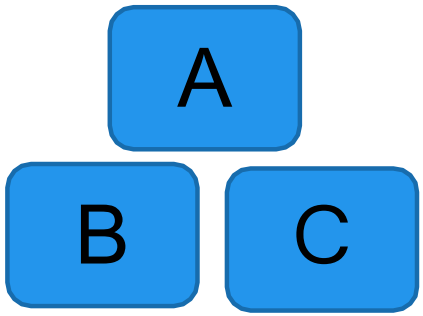
Define your app's environment with a Dockerfile

Define the services that make up your app in Docker Compose file

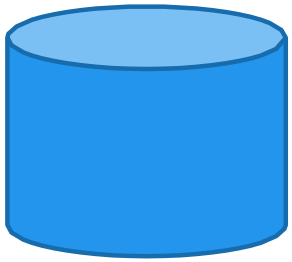
Run the CLI:

docker-compose up

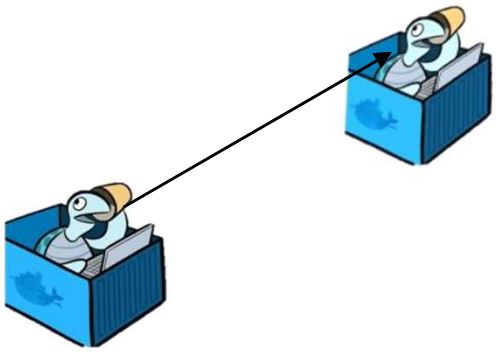
Docker Compose: Multi Container Applications



Services

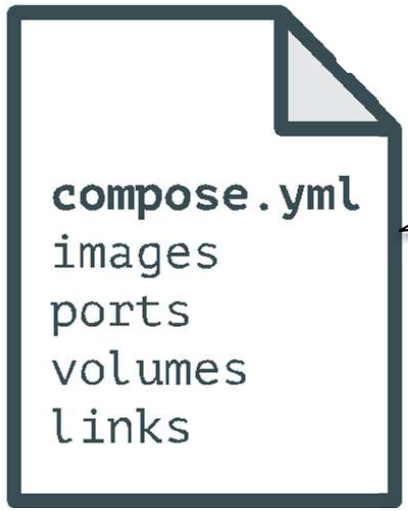


Volumes



Networking

Docker Compose: Multi Container Applications




```
containers:
  web:
    build: .
    command: python app.py
    ports:
    - "5000:5000"
    volumes:
    - .:/code
    environment:
    - PYTHONUNBUFFERED=1
  redis:
    image: redis:latest
    command: redis-server --appendonly yes
```

Docker Compose: Multi Container Applications

```
$ cat docker-compose.yml
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
    environment:
      FLASK_ENV: development
  redis:
    image: "redis:alpine"
```

Docker Compose: Multi Container Applications

Using .env file

```
$ cat .env
TAG=v1.5

$ cat docker-compose.yml
version: '3'
services:
  web:
    image: "webapp:${TAG}"
```

Docker Compose: Multi Container Applications

Commands:

docker compose up
docker compose down
docker compose run –e DEBUG=1 <services>

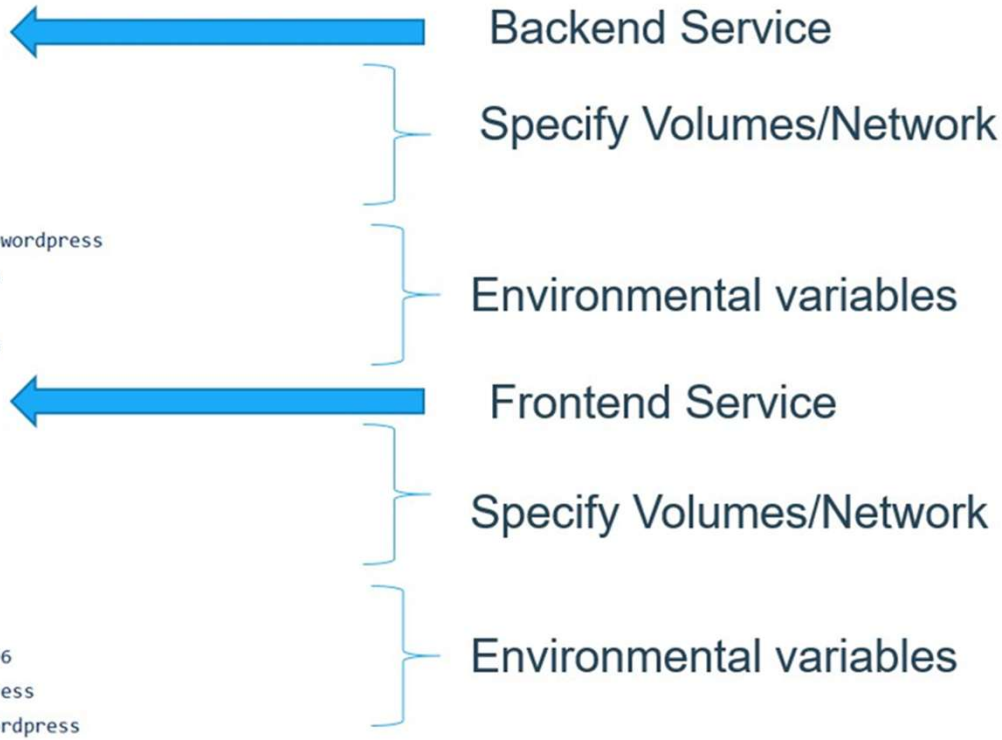
Docker Compose: Multi Container Applications

- A network called myapp_default is created.
- Name is based on directory
- A container is created using web's configuration.
  It joins the network myapp_default under the
  name web.
- A container is created using db's configuration.
- Joins the network myapp_default under the name db.

```
version: "3"
services:
  web:
    build: .
    ports:
      - "8000:8000"
  db:
    image: postgres
    ports:
      - "8001:5432"
```

Docker Compose: Multi Container Applications

```yaml
version: '3'
services:
   db:
     image: mysql:5.7
     volumes:
       - db_data:/var/lib/mysql
     restart: always
     environment:
       MYSQL_ROOT_PASSWORD: somewordpress
       MYSQL_DATABASE: wordpress
       MYSQL_USER: wordpress
       MYSQL_PASSWORD: wordpress
   wordpress:
     depends_on:
       - db
     image: wordpress:latest
     ports:
       - "8000:80"
     restart: always
     environment:
       WORDPRESS_DB_HOST: db:3306
       WORDPRESS_DB_USER: wordpress
       WORDPRESS_DB_PASSWORD: wordpress
volumes:
```



Multi-Stage Builds

- You can use multiple FROM (not just one)
- Each from a different base
- Selective Copy of artifacts from one to another
- Helps in keeping Size small

Multi-Stage Builds

- Example:
- A simple Hello World example in GoLang
- 2 Docker Images
  - One that contains goLang
  - One that only contains the executable
  - Note the **difference in sizes** (both give the same output when run)

helloall	latest	a2f57b6a1ee5	1 minute ago	1.27 GB	▷	⋮
hellomulti	latest	c3780bd7357c	11 seconds ag	3.44 MB	▷	⋮

Multi-Stage Builds

```
D:\jenkins\multi-stage-build>docker run -it helloall
hello, world

D:\jenkins\multi-stage-build>docker run -it hellomulti
hello, world
```

Multi-Stage Builds

- Classic use of such builds
  - Compile the code on an image that has the development tools (dot net compilers, JDK etc)
  - Run the resultant on an image that only has the runtime

Q &A

We're done!

Thank you for your time and participation.