Docker basics

Martin Toshev

Agenda

- Docker basics
- Docker containers
- Docker administration
- Provisioning Docker

Docker basics

What is Docker ?

- A container engine that bundles applications in containers

- Containers share the OS kernel and use resource isolation features of the Kernel

- VMs on the other hand required a hypervisor (such as Hyper-V, VMWare vSphere, VirtualBox, Xen, KVM), make use of user and kernel space and virtualize hardware resources

- In that regard containers provide lesser isolation than VMs and are considered more lightweight than VMs

What is Docker ?

- Features of the Kernel used by Docker include:
  - namespaces: isolate the application's view of the operating system, processes, network, user IDs and mounted file systems
  - cgroups: provide limits on the CPU and memory being used
  - union filesystem (UnionFS) which allows files and directories of separate file systems to be overlaid (thus Docker containers can be layered)

Why Docker ?

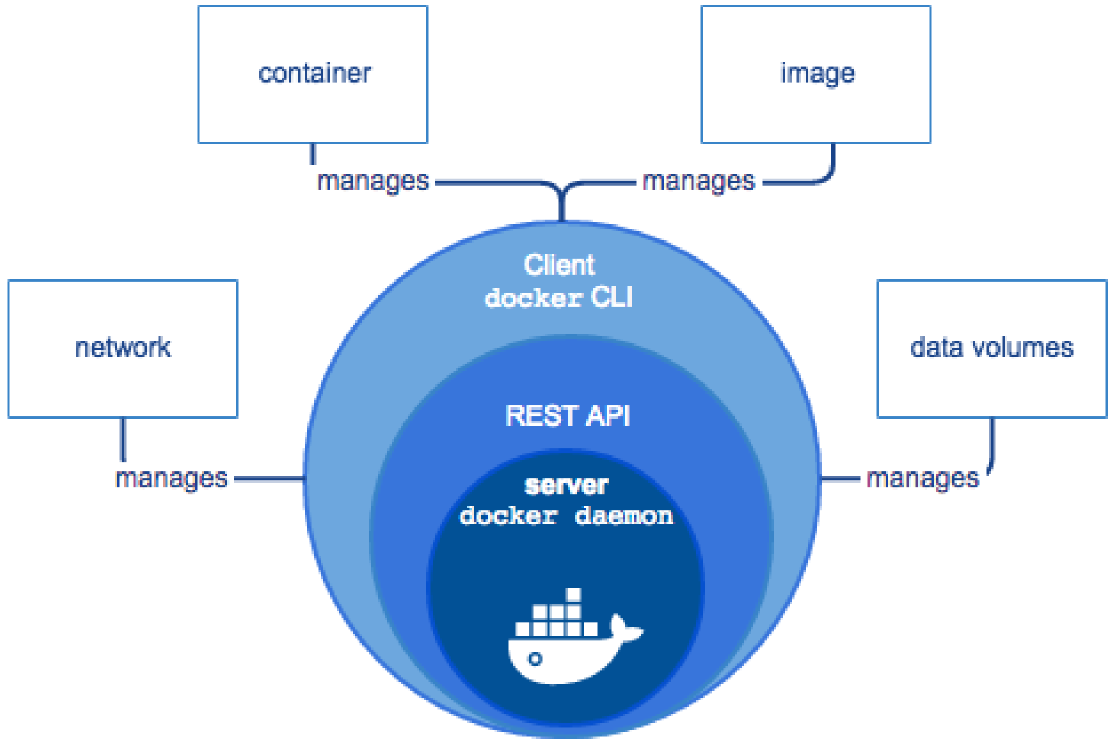
- To understand why Docker is so popular we need to understand the pros and cons of containers vs virtual machines:

Feature	**Container**	**VM**
Bundled OS	No (uses host OS)	Yes
Resource isolation	Lesser isolation based on Kernel capabilities	Strict isolation based on hardware virtualization
Startup time	Relatively fast	Slower than containers
Hardware resources	Relatively small	Higher that containers
Security	Less secure	More secure than a container
Distribution	Containers are easier to distribute via registries	Distribution of VMs is more difficult due to VM size
Automated creation	Available out-of-the box	More difficult to build or further automate the build of a VM

Docker components

- Docker engine contains the following components:
  - Docker daemon (**dockerd**): runs Docker containers and manages Docker images and other objects. Implements the Docker Engine API
  - Docker client(**docker**): communicates with the Docker daemon by sending requests to the Docker Engine API
  - Docker object: container, image or service (manage a cluster of Docker daemons, also called a swarm)
  - Docker registry: a repository for Docker images

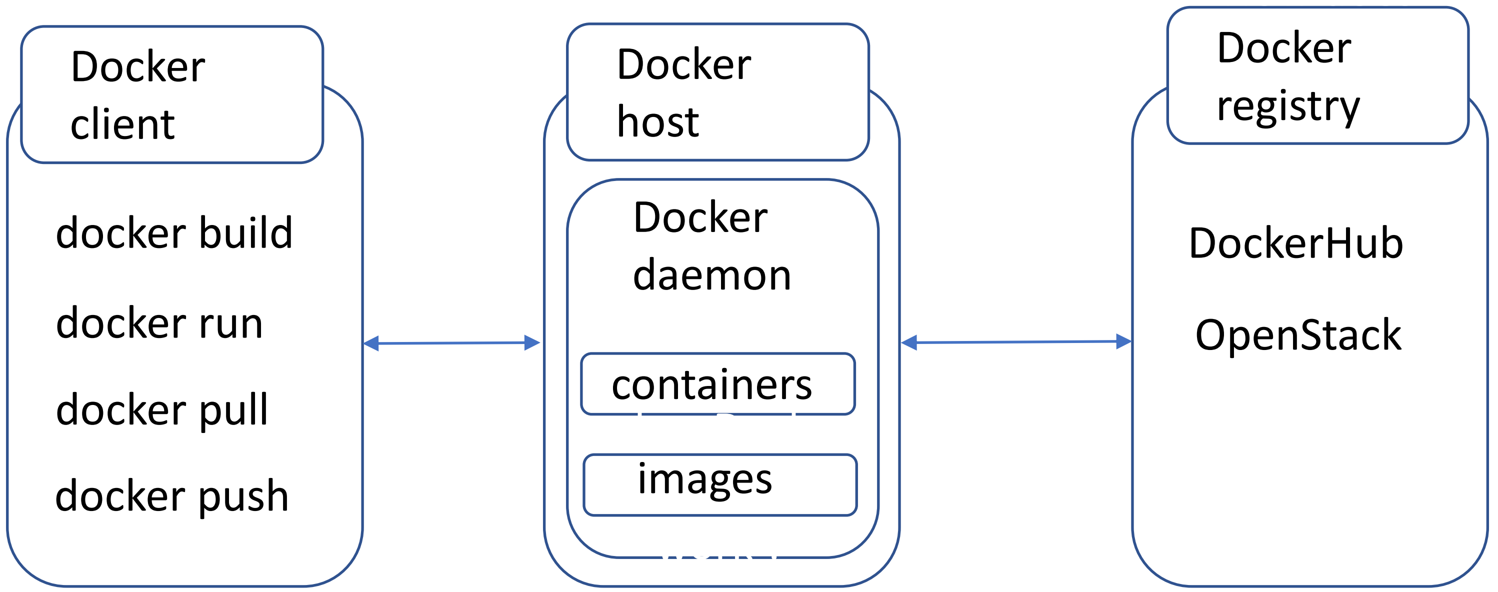
How does Docker work ?



Reference: https://docs.docker.com/engine/docker-overview/

How does Docker work ?

- Architecture:



Installation

- Although Docker engine initially leverages the Linux OS kernel for containerazation installation under Windows and Mac is also supported

- Under Windows docker is installed through Docker Desktop and can run in a Hyper-V VM or as a Windows container

- Previous versions of Docker for Windows (Docker Toolbox) Virtual Box VMs were used to run the Docker containers

Docker alternatives

- Docker is not the only container engine available:
  - BSD Jails
  - LXD
  - LXC
  - Solaris Zones
  - RKT
  - CRI-O

Docker containers

Creating images

- Images can be created by writing a **Dockerfile** with proper commands which are used to build and prepare the image
- Docker images are layered (can reference another image as a base layer)
- Images serve as templates for the Docker containers
- Docker images can be pushed to a central image repository (such as the public DockerHub)

Dockerfile

- Example:

```
# use AdoptOpenJDK build of JDK 13
FROM adoptopenjdk/openjdk13:centos

RUN mkdir /opt/app

WORKDIR /opt/app

COPY app.jar /opt/app

CMD ["java", "-jar", "app.jar"]
```

```
docker image build -t app:1.0.0 .
```

Dockerfile

- Dockerfiles contain lines in the format:

```
# comment
COMMAND arguments
```

- Example:

```
FROM adoptopenjdk/openjdk13:centos
```

- Dockerfile should specify at least one CMD or ENTRYPOINT command !

Dockerfile

- The following commands can be used:

ENV key=value	Declares environment variables (can be references in some commands with $env or ${env}
FROM image	Specifies base image to use or **scratch** for a base image. There can be multiple FROM commands resulting in a multistage build
RUN command	Runs a command during the build of the docker image
CMD command	Can be used to run a command when the container starts
`LABEL <key>=<value> <key>=<value>`	Adds metadata to an image
`EXPOSE <port>`	Specifies port on which the container listens (but does not expose it externally !)

Dockerfile

- The following commands can be used:

ADD <src> <dest>	Copies file or directory from <src> to <dest> in the image
COPY <src> <dest>	Does the same as ADD but with less options
ENTRYPOINT ["command", "param1", …,"paramN"]	Allows a container to behave as an executable when running 'docker run' (and passing extra parameters)
`VOLUME path`	Specifies mount point for the containers

`USER <user>[:<group>] or`
`USER <UID>[:<GID>]`
	Sets and user and (optionally) group with which to run the container

`WORKDIR <dir>`	Sets the current working directory
`ARG <name>[=<default value>]`	Defines build time variables (that can be passed with 'docker build'

Dockerfile

- The following commands can be used:

ONBUILD <instruction>	Creates a trigger instruction that will be triggered when the image is used to build another image with FROM
STOPSIGNAL <signal>	Defines the signal to be sent to the container once stopped
HEALTHCHECK [OPTIONS] CMD <command>	Performs a healthcheck using a specified command being triggered at a specified interval (by default 30s)
`SHELL ["command", "parameters"]`	Triggers a command using default shell

.dockerignore

- A .dockerignore file can be placed along the Dockerfile

- Lists files and directories that are not sent to the Docker daemon (I.e. using command such as ADD or COPY)

```
**/.git
**/.DS_store
**/node_modules
```

Docker image commands

build	Builds an image
history	Shows the history of an image
import	Imports an image from a tarball
inspect	Shows detailed information for an image
load	Loads an image from a TAR archive
ls	List images
prune	Removes unused images
pull	Pull an image or a repository to a registry
push	Push an image or a repository to a registry
rm	Removes one or more images
save	Save one or more images to a TAR archive
tag	Create an image tag

Creating containers

- A container can them be started from a created Docker images
- Containers can be started, stopped, moved or deleted
- Additional isolation can be provided for the containers in terms of network, storage and other subsystems of the host operating system

Docker container commands

ls	List containers
commit	Create a new image from a container
cp	Copies files from/to a container
create	Create a new container from an image
diff	List changes to files and directories since the container was created
exec	Execute a command in a running container
export	Exports the container's file system in a TAR
inspect	Provides detailed information for a container

Docker container commands

kill	Kills one or more running containers
logs	Fetches the logs of a container
pause	Pauses the processes of one or more containers
port	Lists port mappings of a container
prune	Removes all stopped containers
rename	Renames a container
restart	Restarts a container
rm	Removes one or more containers

Docker container commands

run	Runs a command in a new container
start	Starts a container
stats	Displays a stream of resource usage statistics
stop	Stops a container
top	Displays the top running processes of a container
unpause	Unpauses one or more processes within a container
update	Update configuration of one or more containers

The Docker CLI

- Additional operations of the Docker CLI include (many of the **image** and **container** command can be triggered directly from the Docker CLI):

app	(experimental) Provides management commands for Docker applications
context	Commands for managing the Docker context
events	Get realtime events from the Docker server
config	Manage Docker configs
volume	Manage Docker volumes
info	Show system-wide information
login/logout	Login/logout to/from a Docker registry

The Docker CLI

- Additional operations of the Docker CLI include (many of the **image** and **container** command can be triggered directly from the Docker CLI):

network	Commands used to manage Docker networks
node	Commands used to manage Swarm nodes
plugin	Commands used to manage Docker plugins
registry	(enterprise) Commands used to manage Docker registries
search	Search the Docker Hub for images
secret	Used to manage Docker secrets
service	Commands to manage Docker Swarm services
system	Commands to manage Docker

Docker administration

Administration activities

- Docker administrations includes the following activities:
  - managing Docker networks
  - managing Docker volumes and mounts
  - daemon configuration
  - logging and monitoring
  - performance tuning and security
  - backing up Docker containers
  - troubleshooting running containers

Docker networks

- Docker networking system provides a mechanism to define a network for use by Docker containers

- Several types of networks can be used:

  - **bridge**
  - **host**
  - **overlay**
  - **macvlan**
  - **none**

Docker networks

- Current list of networks managed by the host can be listed with the following command:

```
docker network ls
```

- A network has a unique network ID, name, type (driver) and scope (**local**, **global** or **swarm**)

Bridge networks

- Default type of network (if none specified) but can be created explicitly with a target name
- Using the default bridge network containers can communicate only by IP (unless the legacy **--link** option is used to link containers)
- User defined networks provide automatically a DNS for communication between the containers
- Typically used when multiple applications are running in containers on the same host

By default traffic on the bridge network is not forwarded to the outside world

Bridge networks

- A user-defined bridge network can be created with the following command:

```
docker network create bridge-net
```

- A user-defined bridge network can be removed with the following command:

```
docker network rm bridge-net
```

- A user-defined bridge network can be inspected with the following command:

```
docker network inspect bridge-net
```

Bridge networks

- To connect a container during bridge network the **--network** option can be specified

```
docker create --name rabbitmq --network bridge-net \
  --publish 8080:80 rabbitmq:latest
```

- To connect a running container to a bridge network the following command can be used:

```
docker network connect bridge-net rabbitmq
```

- To disconnect a running container from a bridge network the following command can be used:

```
docker network disconnect bridge-net rabbitmq
```

Host networks

- Removes network isolation between host and the Docker container
- Used when the network stack does not need to be isolated from the host
- Host networking implies that ports exposed by the container are bound to the host operating system
- There is only one network of **host** type

```
docker create --name rabbitmq --network host rabbitmq:latest
```

Host networking works for Linux containers only

Overlay networks

- Used to connect multiple Docker daemons or Swarm services
- Overlays the host network
- Used when containers need to communicate over remote hosts

A default overlay network called **ingress** is used by Swarm services by default

Overlay networks

- Docker daemon needs to be initialized as a Swarm manager (even though Swarm is not used) before an overlay network can be created

```
docker swarm init
```

- An overlay network can be created with the **overlay** driver option

```
docker network create --driver=overlay overlay-net
```

Overlay networks require the following ports to be open to and from the Docker host:

- TCP port 2377 for cluster management communications
- TCP and UDP port 7946 for communication among nodes
- UDP port 4789 for overlay network traffic

Macvlan networks

- Provide the possibility to assign MAC addresses to containers
- Traffic is routed based on the MAC address
- Typically used when migrating from VMs to containers

```
docker network create --driver=macvlan --opt parent=eth0
     macvlan-net
```

No networking

- Networking is disabled with the **none** type of network

```
docker create --name rabbitmq --network none rabbitmq:latest
```

Network configuration

- Network configuration (such as subnet and IP range) can be specified when the network is created with **docker network create**
- For default networks that configuration can be modified in the **daemon.json** configuration file
- Network configuration (such as DNS or hostname) for Docker containers can be specified when a container is created or started (via **docker create** or **docker run**)

Storage management

- By default data in a container is stored on a writable layer of the container
- Once the container is destroyed data is gone
- The extra abstraction provided by UnionFS makes reading/writing of data slower compared to regular volumes on the host OS
- Reading/writing data from/to the container is also not convenient

Storage management

- If however data needs to be written mostly on the container (in the writable layer created for the purposes) different storage drivers can be used

- This is possible due to the fact that the storage mechanism on the container is pluggable (configured in **daemon.json** configuration file)

- The following types of storage drivers are supported:
  - overlay2 (preferred)
  - aufs
  - devicemapper
  - btrfs
  - zfs
  - vfs

Storage management

- There are two ways provided by Docker to store data on the host machine:
  - **volumes**: directories on the host system that are managed by Docker and should not be modified by other processes (preferred option)
  - **bind mounts**: directories on the host system not managed by Docker and can be modified by other processes
- Volumes might be named or anonymous (without a given name)
- Volumes can be shared among multiple containers

Volumes

- Docker volumes are managed with the **docker volume** set of commands

- A named volume can be created with the following command:

```
docker volume create sample_volume
```

- A volume can be inspected with the following command:

```
docker volume inspect sample_volume
```

- Volumes can be listed with the following command:

```
docker volume ls
```

Volumes are mounted under /var/lib/docker/volumes/ on the host

Volumes

- To start a container with a volume the **--volume** or **--mount** options can be used

- The **--mount** option is newer (as od Docker 17.06) and provides a different, more verbose syntax used to specify volume options as key-value pairs

```
docker create --name rabbitmq \
     --volume sample_volume:/app rabbitmq:latest
```

```
docker create --name rabbitmq \
     --mount source=sample_volume,target=/app \
     rabbitmq:latest
```

If the container directory already contains files they are first copied to the volume before it is mounted

Volumes

- When mounted volumes are read-write by default

- Volumes can be mounted as read-only

```
docker create --name rabbitmq \
     --volume sample_volume:/app:ro rabbitmq:latest
```

```
docker create --name rabbitmq \
     --mount source=sample_volume,target=/app,readonly \
     rabbitmq:latest
```

Volumes

- Different types of volumes can be mounted as well
- Support for third-party volume driver is provided by means of Docker plug-ins
- The volume driver is specified when the volume is created and mounted
- Each driver may have driver-specific options that need to be passed during the volume creation or mounting

Bind mounts

- A typical use case for using bind mounts is to share folders with existing files (configuration, source code, build artifacts etc.) with the container

- The directory on the host may not be existing and is created by Docker

```
docker create --name rabbitmq \
     --volume /app/dir:/app rabbitmq:latest
```

```
docker create --name rabbitmq \
     --mount type=bind,source=/app/dir,target=/app \
     rabbitmq:latest
```

If the container directory already contains files then they are **OBSCURED** rather than being copied as with volumes !

Security

- One of the major concerns related to Docker is security, especially due to the fact that the Docker daemon needs to run with a root user
- There is at present an experimental feature called **rootless mode** that allows running the Docker daemon with certain privileges
- Although the daemon starts with root Docker uses Linux kernel capabilities that further restrict access control on certain operations performed on the containers

Some of the main concerns related to running Docker in production are due to security and the fact Docker daemon has root access.

Automatic restarts

- Docker can provide automatic restart for containers in case the Docker daemon is restarted or the containers exits through **restart policies**

- The policy is applied with the **--restart** option when the Docker container is created or executed with any of the following options:

  - **no** (default): no restart policy is applied
  - **on-failure**: the container is restarted when exits with a non-zero return code
  - **always**: the container restarts always
  - **unless-stopped**: the container is always restarted unless explicitly stopped

```
docker run --restart unless-stopped --name auto-rabbitmq rabbitmq
```

Daemon configuration

- Docker daemon can be provided via flags passed to the Docker daemon process or a daemon.json configuration file (under /etc/docker/**daemon.json**) under Unix

```
{
  "tls": true,
  "tlscert": "/var/docker/server.pem",
  "tlskey": "/var/docker/serverkey.pem",
  "hosts": ["tcp://192.168.0.5:2345"],
  "log-driver": “somelogdriver",
  “storage-driver": “somestoragedriver"
}
```

Both flags and **daemon.json** can be used for configuration unless there is a configuration overlap

Troubleshooting

- The Docker daemon can be troubleshoot as follows:

  - Review the Daemon logs generated for the specific host OS (i.e. **/var/log/daemon.log** for Debian)

  - Enable Daemon debug mode for verbose output in **daemon.json**

```
{
  “debug": true
}
```

  - If the Docker daemon is unresponsive a SIGUSR1 signal can be sent for retrieval of full stack trace

```
sudo kill -SIGUSR1 $(pidof dockerd)
```

Troubleshooting

- A Docker container can be troubleshoot as follows:
  - review the container logs with **docker logs**
  - review docker statistics with **docker top**
  - execute troubleshooting commands on the container with **docker exec**

Logging

- To view container logs on a container the **docker logs** command can be used
- Container logging is further controlled by logging drivers defined in the Docker daemon configuration that provide different ways to output logs
- The default driver logging file is **json-file** (logging driver can be reviewed with the **docker inspect** command)

Docker registries

- Private Docker registries for image hosting can also be deployed

- The Docker Registry is provided as a Docker container

```
docker run -d -p 5000:5000 --name registry registry:2
```

- Other repository managers like Nexus provide support for Docker Registry whereby a private/hosted/proxy Docker repository can be created and managed by Nexus

As Docker registries do not contain a management UI a third party one can be installed

Provisioning Docker

Provisioning technologies

- In most cases applications are distributed into multiple containers
- The Docker client is not very convenient at managing multiple images or containers at once
- CI/CD systems like Jenkins may be used to automate the process
- Tools like **docker-compose** and **Ansible** also provide the possibility to manage a set of containers

Docker Compose

- Uses a YML file (**docker-compose.yml**) to define application services that are managed in Docker containers
- Uses a project name to isolate services on a single host
- Preserves volume data when containers are (re)created
- Recreates containers it manages for the services only if the corresponding configuration of the service has changed

On Windows and Mac docker-compose is part of the Docker distribution and on Linux it is installed separately

Docker Compose

- Using Docker Compose a container can be built from a Dockerfile or from a Docker registry image

docker-compose.yml

```
version: '3'
services:
  app:
    build: .
  openjdk:
    image: "openjdk:latest"
  rabbitmq:
    image: "rabbitmq:latest"
    ports:
      - "15672:15672"
      - "5672:5672"
```

Docker Compose

- To build Docker images for the services in the docker compose file the **build** command can be used (a wrapper around **docker build** for each service)

```
docker-compose build
```

- To run containers for the services defined in the Docker compose file the **up** command can be used

```
docker-compose up
```

- Only certain services from the compose file can be specified

```
docker-compose build openjdk app
```

```
docker-compose up openjdk app
```

Docker Compose

- Docker Compose service containers can be stopped without removing the containers with the **stop** command

```
docker-compose stop
```

- Docker Compose service containers can be destroyed along with their associated images, networks and volumes using the **down** command

```
docker-compose down
```

- Only certain services from the compose file can be specified

```
docker-compose build openjdk app
```

```
docker-compose up openjdk app
```

Docker Compose

- Environment variables that can be references in the **docker-compose.yml** file by listing them in a **.env** file in the same location

- The .env file lists entries as key-value pairs which can them be referenced with $KEY or ${KEY} in the compose file

.env

```
APP_VERSION=123
OTHER_VARIABLE=SOME_VALUE
```

- Environment variables can also be set using the **–e** flag on **docker-compose run**

Docker Compose commands

- A number of commands provided by Docker Compose provide the equivalent of regular commands on a single container for the group of containers used for the compose services:
  - exec
  - logs
  - pause/unpause
  - ps
  - restart
  - start/stop
  - top
  - push/pull
  - run

Docker Compose file

- The following top elements may specified for a service:

build	Configuration related to the build of the service container
command	Command to be executed by the container
container_name	Specify custom name for the container
depends_on	Specify dependency to another services. Determines startup order of service containers
deploy	Configuration related to deployment of service to a swarm
dns	Specify custom DNS servers

Docker Compose file

- The following top elements may specified for a service:

entrypoint	Specifies entrypoint for the service container
environment	Adds environment variables to the container
expose	Exposes ports without publishing them
image	Specify the image used to start the container from
labels	Specifies metadata for the service container
logging	Logging configuration for the service
networks	Specifies networks to join
ports	Exposes ports for the containers
restart	Specifies restart policy for the service container
volumes	Mount host paths or volumes on the service container

networks and **volumes** can also be specified as top-level elements to configure networks and volumes

Questions ?